Scenarios for interview - Author Pankaj Dhapade

Content

AWS

✅ Scenario 1: EC2 Instance Suddenly Became Inaccessible via SSH

🔸 **What happened:**
An EC2 instance hosting our staging backend suddenly stopped responding to SSH. Monitoring alarms also showed no CloudWatch metrics incoming from that instance.

🔸 **Root cause:**
After investigation:

- The instance had a **custom security group** with `port 22` allowed only from a specific IP
- That IP had changed (due to ISP restart)
- No fallback like Bastion or VPN was in place

🔸 **How I handled it:**

- Used **SSM Session Manager** to connect to the instance without SSH
- Verified the system was up, and networking was fine internally
- Modified the security group rule via the AWS Console to allow current IP (temporarily)
- Later:
  - Introduced Bastion Host for internal-only EC2 access
  - Configured fallback access using SSM + IAM session policies
  - Set up a recurring reminder to validate security group IP rules

🔸 **Lesson learned:**
Avoid hardcoding developer IPs in SGs for SSH. Prefer **Bastion Hosts or SSM** for safer, flexible access.

✅ **Why it's realistic:**
This happens often when teams whitelist specific IPs for SSH but don't use SSM/Bastion properly. Also lets you mention secure practices.

👉 **Cross-question prep:**

- How do you access EC2 securely in production?
- What is Session Manager, and how does it work?
- Why is whitelisting IPs risky?

✅ Scenario 2: S3 Bucket Data Was Publicly Accessible Due to Misconfiguration

🔸 **What happened:**
During an internal audit, we found an S3 bucket (`team-assets-bucket`) was publicly accessible. It stored frontend assets, but sensitive internal data was accidentally uploaded too.

🔸 **Root cause:**

- A developer set **bucket policy** to allow public read for all objects for faster testing
- No S3 Block Public Access settings were enabled
- No object-level tagging to separate public/private data

🔸 **How I handled it:**

- Used:

  ```
  aws s3api get-bucket-policy-status
  aws s3api get-bucket-acl
  ```

  to validate the exposure
- Immediately enabled:
  - S3 Block Public Access at both **account** and **bucket** level
- Moved public files to a separate static bucket for website hosting with scoped IAM policies
- Integrated **AWS Config** to monitor for future public S3 exposure
- Tagged sensitive files using `aws s3 cp ... --metadata` for access-level enforcement via bucket policy

🔸 **Lesson learned:**
S3 security needs **default deny** posture — never allow public access without a controlled use case. Use AWS Config or SCPs to enforce it.

✅ **Why it's realistic:**
This is a **classic AWS security mistake**, especially in teams working with static frontend assets. Fixing it shows security maturity.

👉 **Cross-question prep:**

- How do you secure S3 buckets?
- What is AWS Config and how does it help in compliance?
- Difference between bucket policy and ACL?

✅ Scenario 3: RDS CPU Spiked to 100%, Causing Application Timeouts

🔸 **What happened:**
During a feature rollout, our app started throwing DB timeout errors. CloudWatch showed RDS CPU stuck at 100% for over 15 minutes.

🔸 **Root cause:**

- A new query introduced in the release had an unindexed WHERE clause
- The query executed over a large table (~10M rows)
- Connection pool got exhausted, causing cascading timeouts

🔸 **How I handled it:**

- First scaled the RDS instance vertically (from `db.t3.medium` to `db.t3.large`) to buy time
- Used **Performance Insights** to identify slow query
- Worked with dev team to create a missing index and redeploy
- Added `slow_query_log = 1` and set `long_query_time = 2` in DB parameter group
- Introduced an **alarm on RDS CPU > 85% for 5 min** to catch similar issues earlier

🔸 **Lesson learned:**
Performance bottlenecks can be caused by poor queries. Use **RDS Performance Insights** + indexing best practices + autoscaling thresholds for resilience.

✅ **Why it's realistic:**
Very common issue in early-stage teams — makes you look responsible and aware of root-cause techniques.

👉 **Cross-question prep:**

- How do you troubleshoot RDS performance?
- What is Performance Insights?
- How do you handle DB-level scaling vs query optimization?

✅ Summary — 3 Realistic AWS Mid-Level Scenarios

Scenario	Real-World Takeaways
EC2 SSH inaccessible	Shows secure access practices (SSM, Bastion)
S3 bucket misconfigured	Demonstrates data classification and public access protection
RDS CPU spike	Highlights monitoring + root-cause + rollback skillset

---

Would you like to move next to **Helm**, **GitOps**, or start practicing **mock answers** using these scenarios with STAR (Situation–Task–Action–Result) format?

You've now covered **all major DevOps tools** with solid interview scenarios — this is what separates confident candidates from those faking it 🚀

Git

✅ Git Scenario 1: "PR Merge Broke the Build Due to Missed Rebase"

🔸 **What happened:**
A developer merged a feature branch into `main`, and the CI pipeline on `main` immediately failed. This broke the deployment flow and caused a delay in the sprint demo.

🔸 **Root cause:**
The feature branch had passed its own CI checks, but it was created 10 days ago and not rebased. In the meantime, `main` had diverged significantly. When the feature was merged, incompatible changes in dependency versions caused the build to break.

🔸 **How I handled it:**

- Identified the root cause via Git logs and Jenkins console.
- Reverted the merge using:

```
git revert -m 1 <merge_commit_sha>
```

- Coordinated with the dev team to rebase the branch onto `main`, resolve conflicts, and re-test before merging again.
- Introduced a Jenkins policy to **auto-trigger CI on `main` after every PR merge**, even if PR had passed earlier.
- Later, we enforced a rebase-before-merge rule via GitHub PR templates and reviewer checklist.

🔸 **Lesson learned:**
Even in orgs with CI/CD, stale branches can silently pass their own checks. Always rebase regularly and test on the target branch (`main`) before merging.

🔸 **Why this is realistic:**
Many orgs use **“merge if CI passes”**, but don't enforce rebase before merging — this causes last-minute breaks. It’s a common **real-world failure point**, even in good setups.

🔸 **Red flag to avoid:**
Don’t say "We just pushed it again" — you need to show rollback, coordination, and CI improvement.

✅ Git Scenario 2: "Developer Accidentally Deleted Remote Feature Branch with Valuable Commits"

🔸 **What happened:**
A developer ran:

```
git push origin --delete feature/payment-refactor
```

thinking it was no longer needed. Later, we realized some last-minute valuable commits were only on that branch and never merged.

🔸 **Root cause:**
The developer misunderstood that a parallel branch (`payment-api`) had all commits, but they were not merged. There was no PR, and the branch wasn't protected.

🔸 **How I handled it:**

- Used GitHub's UI to search for the deleted branch and verify if PRs existed (none found).
- Since I had a local copy, I pushed it back using:

  ```
  git push origin feature/payment-refactor
  ```

- Restored the PR and resumed review. After the incident, we implemented:
  - **Branch naming conventions**
  - Expiry rules (but with review/merge status checks)
  - A rule: only delete feature branches **after merge** or archival tag

🔸 **Lesson learned:**
Avoid deleting any branch unless:

1. It's merged
2. It's tagged for archival
3. Team lead or repo owner approves

🔸 **Why this is realistic:**
In fast-moving teams, people often clean up branches. Unless there's **merge enforcement or PR-based cleanup**, this can happen even with senior developers.

🔸 **Red flag to avoid:**
Don't say "We recreated the branch" unless you can explain **how** you recovered it — from GitHub, reflog, or local backup.

✅ Git Scenario 3: "Wrong Git Tag Pushed to Production Release"

🔸 **What happened:**
We released version `v1.2.0`, but users noticed the frontend was missing some recent features. Investigating the Git logs showed that the `v1.2.0` tag pointed to an older commit that didn't include the final merged changes.

🔸 **Root cause:**
The release tag was created locally before the final PR merge was completed, and the developer pushed it without verifying the commit hash. It was an off-by-one error due to async merges.

🔸 **How I handled it:**

- First, I updated the tag locally:

```
git tag -d v1.2.0
git tag v1.2.0 <correct_commit_hash>
git push --force origin v1.2.0
```

- I sent a communication to all stakeholders and triggered a re-deploy of the correct version.
- We then enforced a rule:
  - All release tags must be created by **a CI job**, not manually.
  - Tagging only happens after PR merges & version bump validation.

🔸 **Lesson learned:**
Manual tagging is error-prone. In CI/CD-driven orgs, version tags should be created automatically after successful build, test, and artifact signing steps.

🔸 **Why this is realistic:**
This is a **classic DevOps mistake** that happens often when teams switch from manual to automated releases.

🔸 **Red flag to avoid:**
Don't say "We deleted the tag" — that's risky in teams. Say "We corrected it with team approval and enforced automation afterward."

🧠 Summary: These 3 Git scenarios are real, defensible, and cross-question-proof.

Scenario	Why it's Realistic
PR merged without rebase broke build	Happens when teams don't enforce rebase-before-merge
Deleted branch with unmerged work	Happens when branches are cleaned up manually without checks
Wrong tag pushed to production	Common in semi-automated release setups

---

Let me know if you'd like:

- **Cross-question drills** for these Git scenarios
- Move on to **Jenkins or Docker scenarios next**
- Or get a **runbook-style 1-pager** of Git incident responses for your portfolio

Your preparation is now in elite territory — keep going strong 💪.

Jenkins

✅ Scenario 1: Jenkins Build Failing Due to Missing Secrets

🔸 **What happened:**
A critical production deployment job failed at the Docker login step with `unauthorized: authentication required`. This caused a delay in deployment during a release window.

🔸 **Root cause:**
The Jenkins credential used to authenticate with DockerHub was accidentally deleted during a cleanup of unused credentials. The pipeline expected it via `withCredentials`, but it returned null.

🔸 **How I handled it:**
I immediately switched the deployment to a hotfix backup agent (manual deploy) to reduce downtime. Then I:

- Restored the deleted credential using Jenkins configuration backup (we had `jenkins_home` backed up daily).
- Validated credential bindings in `Jenkinsfile`.
- Added monitoring in our `shared library` to fail fast if expected credentials are missing.
- Finally, I implemented RBAC and credential folder-scoping so only job-specific credentials were visible to each team.

🔸 **Lesson learned:**
Never treat credentials as disposable. Use **fine-grained access** and audit controls. Also, have a **CI/CD fallback** strategy and always test pipelines for credential presence.

🔸 **Red flag to avoid:**
Never say "I manually added the credentials again" without explaining how you verified access or what caused the loss — it sounds patchy.

👉 **Cross-question prep:**

- Where are Jenkins credentials stored on disk?
- How do you inject them into pipelines securely?
- Difference between global vs folder-level credentials?

✅ Scenario 2: Jenkins Build Slowness Due to Shared Agent Overload

🔸 **What happened:**
Developers reported that CI builds were taking over 30 minutes during peak times. This affected their

productivity and caused bottlenecks in merging PRs.

🔸 **Root cause:**
We were using a static Jenkins agent node shared across multiple jobs. During load testing or heavy PR activity, CPU/RAM bottlenecked. Jenkins wasn't scaling because the executor count was fixed and cloud autoscaling was disabled.

🔸 **How I handled it:**
I:

- Analyzed job concurrency metrics and found we had >3x usage spikes during deployments.
- Integrated Jenkins with our Kubernetes cluster using the **Kubernetes plugin** for dynamic pod-based agents.
- Updated `Jenkinsfile` to use custom agent templates with needed tools.
- Set up pod autoscaling and defined limits so jobs get isolated containers.

After that, average build time dropped from 30min → 9min.

🔸 **Lesson learned:**
Static agents don't scale. Use **dynamic, container-based agents** for CI/CD. Jenkins plugins like Kubernetes and Docker Agent make it easy to scale on demand.

🔸 **Red flag to avoid:**
Saying "We increased memory and it worked" isn't a reliable fix — it shows short-term thinking instead of architectural optimization.

👉 **Cross-question prep:**

- How does the Kubernetes plugin work in Jenkins?
- What are executors in Jenkins?
- How do you isolate builds in dynamic agents?

✅ Scenario 3: Shared Jenkins Library Broke All Pipelines

🔸 **What happened:**
We had a shared Jenkins library used across 40+ pipelines. A recent update to that library introduced a bug in a common method (`dockerPush()`), which caused all downstream jobs to fail during the deployment stage.

🔸 **Root cause:**
The shared library didn't use semantic versioning — all jobs pulled the latest version from Git automatically. There was no testing or staging step before pushing changes.

🔸 **How I handled it:**

- I rolled back the library to a stable git commit and pushed it as a versioned tag.
- Modified `Jenkinsfile` in pipelines to use the tag instead of the `master` branch.
- Created a new Jenkins pipeline just for testing changes to the shared library.
- Implemented semantic versioning (`v1.2.0`, `v1.3.0`) and changelog enforcement before merging to main.

🔸 **Lesson learned:**
Don't use "latest" or "main" in production pipelines for libraries. Always use **version-controlled shared libraries** and promote them via CI.

🔸 **Red flag to avoid:**
Don't say "I updated all Jenkinsfiles with the fix manually" — it sounds brittle and unscalable. Also, don't ignore the need for a staging pipeline for library testing.

👉 **Cross-question prep:**

- How do shared libraries work in Jenkins?
- How do you test and version them?
- Can you explain `@Library` annotation and `vars` vs `src` structure?

✅ Summary — 3 Jenkins Scenarios to Master

Scenario	Concepts Tested
Missing credentials	Secure storage, fallback strategies, RBAC
Build slowness	Agent scaling, dynamic infra, Kubernetes plugin
Shared lib breakage	Versioning, reusability, testing in CI/CD

Would you like me to go ahead with **Docker next**, or should I now give **follow-up cross-question drills** for Jenkins scenarios to simulate real interviews?

Ansible

✅ Mid-Level Ansible Scenario 1: Playbook Failed Due to Missing Become Privilege

🔸 **What happened:**
I was running a playbook to install Apache (`httpd`) on a new set of RHEL servers, and the task failed with a `permission denied` error on package installation.

🔸 **Root cause:**
The playbook didn't specify `become: yes`, and the remote user didn't have root access by default. The role used `yum` module, which of course requires privilege escalation.

🔸 **How I handled it:**

- I added `become: yes` at the play level to ensure all tasks run with proper privileges.
- Also made sure the SSH user had sudo rights by checking `/etc/sudoers.d/`.
- Re-ran the playbook successfully.

🔸 **Lesson learned:**
Even simple playbooks need proper privilege management. It's better to define `become: yes` at the play or role level instead of relying on per-task overrides.

🔸 **Why this is realistic and safe:**
Very common mistake — many engineers miss this early on, especially if playbooks are copied or reused.

👉 **Cross-question prep:**

- When do you use `become: yes`?
- Can you explain how Ansible handles privilege escalation?
- What's the difference between `become_user` and `remote_user`?

✅ Mid-Level Ansible Scenario 2: Playbook Failed on One Host But Passed on Others

🔸 **What happened:**
We used Ansible to configure firewall rules across 5 Linux servers. The playbook succeeded on 4 hosts but failed on one with `firewalld not found`.

🔸 **Root cause:**
That specific host was running Debian, while others were RHEL-based. The playbook used the

`firewalld` module, which is not available or supported on Debian (which uses `ufw`).

🔸 **How I handled it:**

- I used a fact-based conditional in the playbook:

```
when: ansible_os_family == "RedHat"
```

- For Debian systems, added a task using the `ufw` module.
- This allowed the same playbook to be OS-aware and safe to run across mixed infra.

🔸 **Lesson learned:**
Always account for **OS-specific differences** when writing common playbooks. Use Ansible facts like `ansible_os_family` or `ansible_distribution`.

🔸 **Why this is realistic and safe:**
Almost every DevOps engineer faces this when infrastructure isn't uniform — and fixing it with `when:` shows awareness.

👉 **Cross-question prep:**

- How do you write OS-agnostic playbooks?
- What facts does Ansible collect automatically?
- How do you test a playbook across different OS types?

✅ Mid-Level Ansible Scenario 3: Ansible Copied Wrong Config File Due to Missing Template Variable

🔸 **What happened:**
We had a task that copied a templated config file (`nginx.conf.j2`) to all web servers. After deployment, nginx failed to start on 2 out of 6 servers.

🔸 **Root cause:**
In the Jinja2 template, a variable `{{ worker_processes }}` was used, but it was not defined in `host_vars` or `group_vars` for those two hosts. The template rendered a blank value, creating an invalid config.

🔸 **How I handled it:**

- Used `--diff` and `--check` mode to confirm the rendered output.
- Added a default fallback in the template like:

```
worker_processes {{ worker_processes | default(2) }};
```

- Updated playbooks to include this variable via group_vars.

🔸 **Lesson learned:**
Templates should always include **default values** to avoid breaking deployments due to missing variables. CI tools should validate rendered configs before applying.

🔸 **Why this is realistic and safe:**
This is a classic mid-level templating mistake — and fixing it with `default()` makes you sound experienced without exaggeration.

👉 **Cross-question prep:**

- How does Jinja2 templating work in Ansible?
- What happens when a variable is missing in a template?
- How do you preview templates before applying?

✅ Summary — 3 Mid-Level, Interview-Safe Ansible Scenarios

Scenario	Key Concepts
Missing `become: yes` caused package install failure	Privilege escalation, sudo access
Playbook failed on one host due to OS mismatch	Fact-based conditions, `when`
Templating error caused service to fail	Safe defaults, variable hygiene, config testing

---

Would you like me to now generate **Docker** mid-level scenarios next, or help you practice **interview-style answers** for these Ansible examples?

Docker

✅ Docker Scenario 1: Container Failed to Start Due to Port Conflict

🔸 **What happened:**
I was deploying a containerized Node.js app on an EC2 instance during testing. The `docker run -p 3000:3000` command worked on my local machine, but on the EC2 server, the container exited immediately.

🔸 **Root cause:**
Port `3000` was already being used by another container running in the background. I found this by running:

```
sudo lsof -i -P -n | grep LISTEN
```

🔸 **How I handled it:**

- Identified the conflict using:

  ```
  docker ps
  ```

  and

  ```
  netstat -tulnp
  ```

- Either stopped the old container or re-mapped the new one using:

  ```
  docker run -d -p 3001:3000 app-image
  ```

- Also introduced **health checks** and **restart policies** in the Docker run command so services don't silently fail.

🔸 **Lesson learned:**
Always check for **port availability** before deploying containers — especially in multi-service or shared environments.

🔸 **Why this is realistic:**
This happens often in test/staging environments or when reusing EC2s for multiple services.

👉 **Cross-question prep:**

- How do you find what's running on a port?
- How do you avoid port conflicts in production environments?
- What's the difference between container port and host port?

✅ Docker Scenario 2: Image Size Was Too Large for CI/CD Pipeline

🔸 **What happened:**
Our Jenkins pipeline was timing out while pulling Docker images in the deployment stage. Investigating further showed the custom app image was over **1.5 GB** in size.

🔸 **Root cause:**
The Dockerfile used:

```
FROM ubuntu:latest
```

and included `apt install` for many build tools. No `.dockerignore` file was present, so large local files like `node_modules` and logs were being copied into the image.

🔸 **How I handled it:**

- Added a `.dockerignore` to exclude unnecessary files like `.git`, `node_modules`, `test-reports/`, etc.
- Replaced `ubuntu:latest` with `node:18-alpine`
- Used **multi-stage builds**:
  1. One stage to build the app
  2. Second stage to copy only final build artifacts

Final image size dropped from **1.5 GB → 200 MB**.

🔸 **Lesson learned:**
Optimizing Docker images is critical in CI/CD pipelines — smaller images = faster pulls, fewer failures, lower storage.

🔸 **Why this is realistic:**
Most developers build images without focusing on size. This leads to bloated images that become a problem only at deployment time.

👉 **Cross-question prep:**

- What is a multi-stage build and why use it?
- How does `.dockerignore` help?
- How do you inspect image layers?

✅ Docker Scenario 3: Application Inside Container Crashed with No Logs

🔸 **What happened:**
After running `docker run my-app`, the container exited instantly with exit code `1`, and `docker logs` showed nothing.

🔸 **Root cause:**
The Dockerfile had:

```
CMD ["npm", "start"]
```

but the Node.js app expected `node server.js`. The `npm start` script was missing from `package.json`, so the container couldn't find the command.

Since the command failed immediately and wasn't logging to stdout/stderr, no output appeared in `docker logs`.

🔸 **How I handled it:**

- Ran the container interactively:

  ```
  docker run -it my-app /bin/sh
  ```

- Manually started the app using `node server.js` and confirmed it worked.
- Updated `package.json` with the correct `start` script.
- Added error logging to `stderr` so it appears in container logs.

🔸 **Lesson learned:**
Containers should always be tested for startup logic. If using CMD/ENTRYPOINT, verify they work exactly as expected. Also, make sure logs go to stdout/stderr.

🔸 **Why this is realistic:**
Startup command issues are very common — especially when moving from local development to Docker.

👉 **Cross-question prep:**

- How do you debug a container that exits immediately?
- Difference between `CMD` and `ENTRYPOINT`?
- How can you persist logs from containers?

✅ Summary — 3 Mid-Level Docker Scenarios for Interview

Scenario	Key Concepts
Port conflict on container start	Host/container ports, netstat, conflict resolution
Image size too large	.dockerignore, multi-stage build, base image choice

Scenario	Key Concepts
Container exits silently	CMD debugging, startup validation, log routing

---

Would you like to go with **Terraform**, **CI/CD pipeline**, or **AWS** mid-level scenarios next?

Or should I now help you **practice answering one of these Docker scenarios like an interview conversation**?

Kubernetes

✅ K8s Scenario 1: Pod CrashLoopBackOff After ConfigMap Update

🔸 **What happened:**
A backend service running in Kubernetes suddenly went into a `CrashLoopBackOff` state after a config update. This affected internal APIs used by other services.

🔸 **Root cause:**
We updated a ConfigMap that had a JSON config used by the application. The updated value had a missing closing brace `}`. Since the app validated config on startup, it crashed immediately. The liveness probe kept restarting the pod.

🔸 **How I handled it:**

- Ran `kubectl describe pod` and saw the restart count increasing.
- Used `kubectl logs` to check the crash reason → config parsing error.
- Reverted the ConfigMap and re-applied using:

```
kubectl apply -f configmap.yaml
```

- Recreated the pods using `kubectl rollout restart`.

Then I:

- Introduced `kubectl diff` before applying YAML to preview changes
- Added a validation step to CI to lint/validate JSON before updating ConfigMap

🔸 **Lesson learned:**
Always validate ConfigMap contents **before applying**, especially when they affect app startup. Even small syntax errors can cause service restarts.

🔸 **Why it's realistic and safe:**
This is very common and doesn't make the engineer or team look bad — it's about handling config safely.

👉 **Cross-question prep:**

- How do you troubleshoot a CrashLoopBackOff?
- How do you safely update a ConfigMap?
- What's the difference between `kubectl apply`, `replace`, and `edit`?

✅ K8s Scenario 2: Service Not Reachable Inside Cluster

🔸 **What happened:**
One team reported that their frontend app could not connect to the backend via the service name. The backend pod was running fine, but DNS resolution failed.

🔸 **Root cause:**
The backend service was created, but the label selector didn't match the pods:

```
selector:
  app: my-backend
```

But pods had:

```
labels:
  app: backend
```

So the service pointed to **0 endpoints**, and DNS resolved but traffic had nowhere to go.

🔸 **How I handled it:**

- Used:

  ```
  kubectl get endpoints
  ```

  → found it had `0/0` endpoints.
- Described the service and pods to compare labels.
- Fixed the selector to match actual pod labels and re-applied the service.

🔸 **Lesson learned:**
In Kubernetes, **service selectors must match pod labels exactly**. Small mismatches lead to silent routing issues.

🔸 **Why it's realistic and safe:**
This is a mistake even experienced engineers make when defining YAML manually.

👉 **Cross-question prep:**

- How do you troubleshoot service-to-pod connection issues?
- What is the role of `Endpoints`?
- How do you test DNS inside a pod?

✅ K8s Scenario 3: ImagePullBackOff on Deployment

🔸 **What happened:**
We deployed a new microservice using a new container image pushed to a private Docker registry. But

the pod failed with `ImagePullBackOff`.

🔸 **Root cause:**
The image was hosted in a private registry, but we forgot to create a Kubernetes **ImagePullSecret**. So the pod couldn't authenticate and pull the image.

🔸 **How I handled it:**

- Confirmed the error using:

  ```
  kubectl describe pod
  ```

  → saw `image pull failed: unauthorized`.
- Created a secret using:

  ```
  kubectl create secret docker-registry regcred \
    --docker-server=<registry-url> \
    --docker-username=<user> \
    --docker-password=<pass> \
    --docker-email=<email>
  ```
- Added the secret to the Deployment spec:

  ```
  imagePullSecrets:
    - name: regcred
  ```
- Re-deployed successfully.

🔸 **Lesson learned:**
For private registries, always create and bind `imagePullSecrets` — and validate access before deployment.

🔸 **Why it's realistic and safe:**
Happens often in real-world clusters — especially with self-hosted registries like Harbor or GitHub Container Registry.

👉 **Cross-question prep:**

- What causes `ImagePullBackOff`?
- How do you authenticate with private registries in K8s?
- What is `imagePullPolicy` and when do you use `Always`?

✅ Summary — 3 Mid-Level Kubernetes Scenarios You Can Defend Confidently

Scenario	Real-World Insight
CrashLoopBackOff due to bad ConfigMap	Shows YAML troubleshooting and probe awareness
Service unreachable due to label mismatch	Demonstrates understanding of service discovery
ImagePullBackOff from private registry	Proves you can debug and fix pod image issues

---

Would you like me to continue with **Terraform**, **AWS**, or **CI/CD pipeline** mid-level scenarios next?

Or would you prefer I help you practice one of these in **mock interview format** — like “Tell me about a time when your pod failed…”?

Terraform

✅ Terraform Scenario 1: Manual Change in AWS Caused Drift

🔸 **What happened:**
One of my teammates manually updated the instance type of an EC2 in the AWS Console from `t3.micro` to `t3.medium` for performance testing. Later, when I ran `terraform apply`, Terraform **downgraded** it back to `t3.micro` unexpectedly.

🔸 **Root cause:**
Terraform didn't know about the manual change — so during apply, it treated the change as a drift and enforced the desired state (`t3.micro` from code).

🔸 **How I handled it:**

- I ran `terraform plan` and saw the instance marked as "~" (change in-place).
- Discussed with the teammate, and confirmed the `t3.medium` change was still needed.
- Updated the Terraform code:

```
instance_type = "t3.medium"
```

- Re-applied to make the infrastructure consistent with intent.

🔸 **Lesson learned:**
Never manually change Terraform-managed infra from the AWS Console. We enforced this with tagging + CloudTrail alerts + documentation.

🔸 **Why it's realistic and safe:**
Manual drift is common in real environments. This answer shows you understand how Terraform treats the infrastructure as *the source of truth*.

👉 **Cross-question prep:**

- What is drift in Terraform?
- How do you detect and prevent it?
- What tools can help detect out-of-band changes?

✅ Terraform Scenario 2: State File Conflict Between Two Team Members

🔸 **What happened:**
During a sprint, I applied changes to a shared Terraform module while another teammate unknowingly did the same. This led to a **state mismatch**, and my changes were overridden.

🔸 **Root cause:**
We were using a remote backend (S3 + DynamoDB), but state locking was not properly enabled. The `dynamodb_table` used for state lock was misconfigured — so both of us could run `apply` simultaneously.

🔸 **How I handled it:**

- Diagnosed that both applies happened within minutes.
- Fixed the locking issue by enabling DynamoDB state locking properly:

```
backend "s3" {
  bucket         = "my-tf-state"
  key            = "env/dev/infra.tfstate"
  region         = "ap-south-1"
  dynamodb_table = "terraform-lock"
  encrypt        = true
}
```

- Documented usage rules: always use `terraform plan` + review before apply.
- Enabled CI pipeline to handle `terraform apply` instead of doing it manually.

🔸 **Lesson learned:**
State locking is critical in team environments. Even small misconfigurations can cause invisible bugs.

🔸 **Why it's realistic and safe:**
Most DevOps teams hit this at some point — especially when starting out with collaboration.

👉 **Cross-question prep:**

- How does Terraform handle state locking?
- What happens if two people run `apply` at the same time?
- How do you set up a team-safe Terraform workflow?

✅ Terraform Scenario 3: Accidentally Destroyed a Non-Terraform Resource

🔸 **What happened:**
I used `terraform import` to bring an existing RDS database under Terraform control. Later, someone removed it from the code, thinking it was unused. On the next `terraform apply`, it was destroyed.

🔸 **Root cause:**
Terraform treats resources not present in the code as candidates for deletion. Since the DB was imported but not yet fully defined in code, it lacked lifecycle protection.

🔸 **How I handled it:**

- Restored from automated snapshot (RDS backup).
- Updated the DB config in code properly.
- Added:

  ```
  lifecycle {
    prevent_destroy = true
  }
  ```

  to critical resources like RDS, S3, and ALBs.

We also made it mandatory to use:

```
terraform plan -out=tf.plan
terraform show tf.plan
```

before every apply — to **review what will be destroyed**.

🔸 **Lesson learned:**
Imported resources need full config in code + lifecycle protection. Otherwise, accidental deletion can occur.

🔸 **Why it's realistic and safe:**
Many teams import existing infra — and this incident shows you understand the risk and took real-world precautions.

👉 **Cross-question prep:**

- What does `prevent_destroy` do?
- How does `terraform import` work?
- What should you check before running `terraform apply`?

✅ Summary — 3 Mid-Level Terraform Scenarios for Interview

Scenario	Real Insight
Manual change in AWS caused drift	Demonstrates state awareness & GitOps thinking
State file conflict	Shows teamwork, locking, and safe collaboration
Resource deletion after import	Explains lifecycle protections and recovery process

---

Let me know if you'd like to:

- Proceed with **AWS**, **CI/CD**, or **Prometheus + Grafana** mid-level scenarios next

- Or practice **mock interview responses** for these Terraform scenarios one by one

You're building a solid runbook library now — you’re almost interview battle-ready 💪

CI/CD

✅ CI/CD Scenario 1: Deployment Pipeline Skipped Tests and Broke Production

🔸 **What happened:**
A developer accidentally disabled unit tests in a Jenkinsfile while experimenting with a branch. The pipeline passed and deployed a buggy build to production.

🔸 **Root cause:**
The `test` stage in Jenkinsfile was wrapped with:

```
when {
  branch 'main'
}
```

But the commit was merged into `main` without updating the condition, so tests were skipped even in the main branch.

🔸 **How I handled it:**

- Investigated via Jenkins pipeline history and confirmed `test` stage was skipped.
- Reverted the code in production using the previous deployment artifact.
- Updated the pipeline to include:

  ```
  when {
    not { branch 'hotfix' }
  }
  ```

- Added a `failFast` check: if tests are skipped, mark the build as failed.
- Enforced **code review for Jenkinsfile changes**.

🔸 **Lesson learned:**
CI pipelines are also code — they need review, test coverage, and protection from bypass. Use stage guards and fail-safes.

🔸 **Why it's realistic:**
Many teams overlook their Jenkinsfile changes during fast merges. This is a production-safe, explainable failure.

👉 **Cross-question prep:**

- How do you enforce test stages in pipelines?
- How can you prevent accidental pipeline stage skips?

- Should Jenkinsfiles be protected?

✅ CI/CD Scenario 2: Artifact Deployed Was Not Matching Code in Git

🔸 **What happened:**
After a Jenkins build, the app deployed to staging didn't include a recent bug fix that was merged to `main`. The Git log had the commit, but the running app didn't reflect the change.

🔸 **Root cause:**
A race condition in the pipeline caused Jenkins to pull the old Git commit due to shallow cloning (`--depth 1`) and caching issues. The node reused an outdated workspace.

🔸 **How I handled it:**

- Used `git rev-parse HEAD` inside the Jenkins workspace to confirm commit mismatch.
- Added a `cleanBeforeCheckout()` block in the Jenkins pipeline:

```
checkout([$class: 'GitSCM', cleanBeforeCheckout: true, ...])
```

- Also removed the shallow clone flag so Jenkins fetches the full history.
- Implemented version tagging based on Git commit hash for traceability.

🔸 **Lesson learned:**
Ensure every build pulls fresh code from Git. Don't trust workspace cache in CI unless explicitly managed.

🔸 **Why it's realistic:**
CI tools like Jenkins and GitHub Actions often reuse workspaces — unless you clean them, you can easily get stale artifacts.

👉 **Cross-question prep:**

- How do you ensure artifact integrity in CI/CD?
- What's the risk of using cached workspaces?
- How do you verify if your deployment matches Git?

✅ CI/CD Scenario 3: Rollback Failed Due to Missing Previous Artifact

🔸 **What happened:**
A production deployment failed due to a misconfigured environment variable. We decided to roll back — but couldn't find the previous artifact in the Jenkins artifact repository.

🔸 **Root cause:**
Jenkins was configured to keep artifacts for only the last 3 builds due to disk constraints. Since the pipeline was heavily used during the sprint, the previous stable build had been deleted.

🔸 **How I handled it:**

- Searched for the previous artifact in Nexus/S3 → not found.
- Re-built from the last known good Git tag, using:

  ```
  git checkout tags/v1.3.2
  ```

  and triggered a new CI build with `BUILD_FROM_TAG=true`.
- Modified Jenkins to:
  - Archive stable artifacts for 30 days
  - Upload all production artifacts to S3 with version tagging
- Documented the rollback process with artifact links.

🔸 **Lesson learned:**
Rollback must be fast and predictable — always retain at least **one previously successful release artifact** in a central place.

🔸 **Why it's realistic:**
Artifact retention misconfigurations are common — especially in fast-moving projects or with self-hosted Jenkins instances.

👉 **Cross-question prep:**

- How do you manage rollback in your CI/CD pipeline?
- Where are artifacts stored, and how long are they retained?
- How do you link a deployed app back to its Git commit?

✅ Summary — 3 Mid-Level CI/CD Scenarios for Interviews

Scenario	Real Insight
Test stage skipped, buggy release pushed	Stage enforcement, pipeline review
Deployed artifact didn't match Git	Workspace caching, Git integrity
Rollback failed due to missing artifact	Artifact storage, retention policies

---

Would you like the next batch on **AWS**, **Prometheus + Grafana**, or shall we start practicing **mock interview delivery** for these scenarios?

You're now building an enterprise-grade runbook — very few candidates do this. Let's keep it sharp 💼💪

Prometheus+Grafana

✅ Scenario 1: Prometheus Alert Didn't Fire During a Real Downtime

🔸 **What happened:**
Our team faced a backend outage — the app was down for over 5 minutes, but no alert was fired. We only realized it via user escalation.

🔸 **Root cause:**
Prometheus was collecting data using a `/healthz` endpoint exposed by the app. But that endpoint continued returning `200 OK` even during actual app failures (e.g., DB connection issues).

So, **Prometheus thought the app was healthy**, and alert rules were never triggered.

🔸 **How I handled it:**

- Modified the `/healthz` endpoint logic to return non-200 when dependencies like DB or Redis were down.
- Updated alert rule to:

```
- alert: AppDown
  expr: up{job="my-app"} == 0
  for: 2m
  labels:
    severity: critical
```

- Added **blackbox exporter** probes to monitor end-to-end HTTP response instead of relying only on internal metrics.
- Reviewed all other service health checks for meaningful error signaling.

🔸 **Lesson learned:**
Never rely on a shallow `/healthz` endpoint. Design health checks to reflect **true service health**, not just uptime.

✅ **Why it's realistic:**
This happens often when developers stub `/health` just to return 200. Many teams don't realize Prometheus will never catch logic-level failures without meaningful metrics.

👉 **Cross-question prep:**

- What's the difference between liveness/readiness and real health?
- How can Prometheus alert on app-level issues?

- When would you use blackbox_exporter?

✅ Scenario 2: Grafana Dashboard Showing N/A for All Panels

🔸 **What happened:**
Team members reported that all panels in a key Grafana dashboard were showing `N/A` or “No data”. Prometheus itself was up and seemed to be scraping targets.

🔸 **Root cause:**
The `prometheus` data source in Grafana was misconfigured after a redeployment. The service name changed (`prometheus-service` → `prometheus-monitoring`) but the data source URL was still pointing to the old one.

So, Grafana couldn't query any metrics.

🔸 **How I handled it:**

- Verified Grafana’s data source via Settings → Data Sources
- Updated the Prometheus URL in Grafana to:

```
http://prometheus-monitoring:9090
```

- Tested using the **Explore** tab to confirm metric visibility
- Added a **Grafana provisioning YAML** to automate data source config in future deploys

🔸 **Lesson learned:**
Grafana dashboards can silently break if the data source moves. Always automate dashboard and data source configuration using provisioning or Terraform.

✅ **Why it’s realistic:**
Happens frequently in Kubernetes where service names or namespaces change. Also common in disaster recovery restores.

👉 **Cross-question prep:**

- How do you troubleshoot `N/A` in Grafana panels?
- Where is Grafana data source config stored?
- What is Grafana provisioning?

✅ Scenario 3: Prometheus Consuming High Memory and Restarting Frequently

🔸 **What happened:**
Prometheus pods in staging started crashing repeatedly. The pod memory spiked up to 2GB+ and hit Kubernetes memory limits, causing restarts.

🔸 **Root cause:**

- Prometheus was scraping **too many time series**, including:
  - All container-level metrics across `kube-system`
  - High-cardinality labels like `pod_name`, `container_id`, `task_id`
- No limits were set on the scrape intervals or label filtering.

🔸 **How I handled it:**

- Used the `/status/tsdb` endpoint and `promtool` to inspect active time series
- Added `metric_relabel_configs` to drop high-cardinality labels:

```
metric_relabel_configs:
  - source_labels: [container_id]
    action: drop
```

- Reduced scrape interval for less-critical exporters (e.g., node-exporter from 15s to 60s)
- Tuned `resource.requests` and `limits` for Prometheus in its Helm chart

🔸 **Lesson learned:**
Prometheus is powerful but memory-hungry. Always monitor cardinality and avoid collecting everything blindly.

✅ **Why it's realistic:**
Cardinality bloat is a classic Prometheus issue — it's a mistake many DevOps teams only learn after running it in production.

👉 **Cross-question prep:**

- What is time-series cardinality?
- How do you tune Prometheus performance?
- How do you debug resource usage in Prometheus?

✅ Summary — 3 Mid-Level Prometheus + Grafana Scenarios

Scenario	Real Insight
Alert didn't fire due to shallow health check	Shows you understand what metrics matter
Grafana panels showed N/A	Data source debugging, service misconfig recovery
Prometheus crashing due to high memory	Resource tuning, cardinality control

---

Would you like me to proceed next with **AWS monitoring incidents**, **Helm scenarios**, or help you **practice how to speak about one of these examples in an interview-friendly way**?

You're very close to having a fully production-ready runbook library — and this kind of prep makes you stand out in interviews 🧠💥