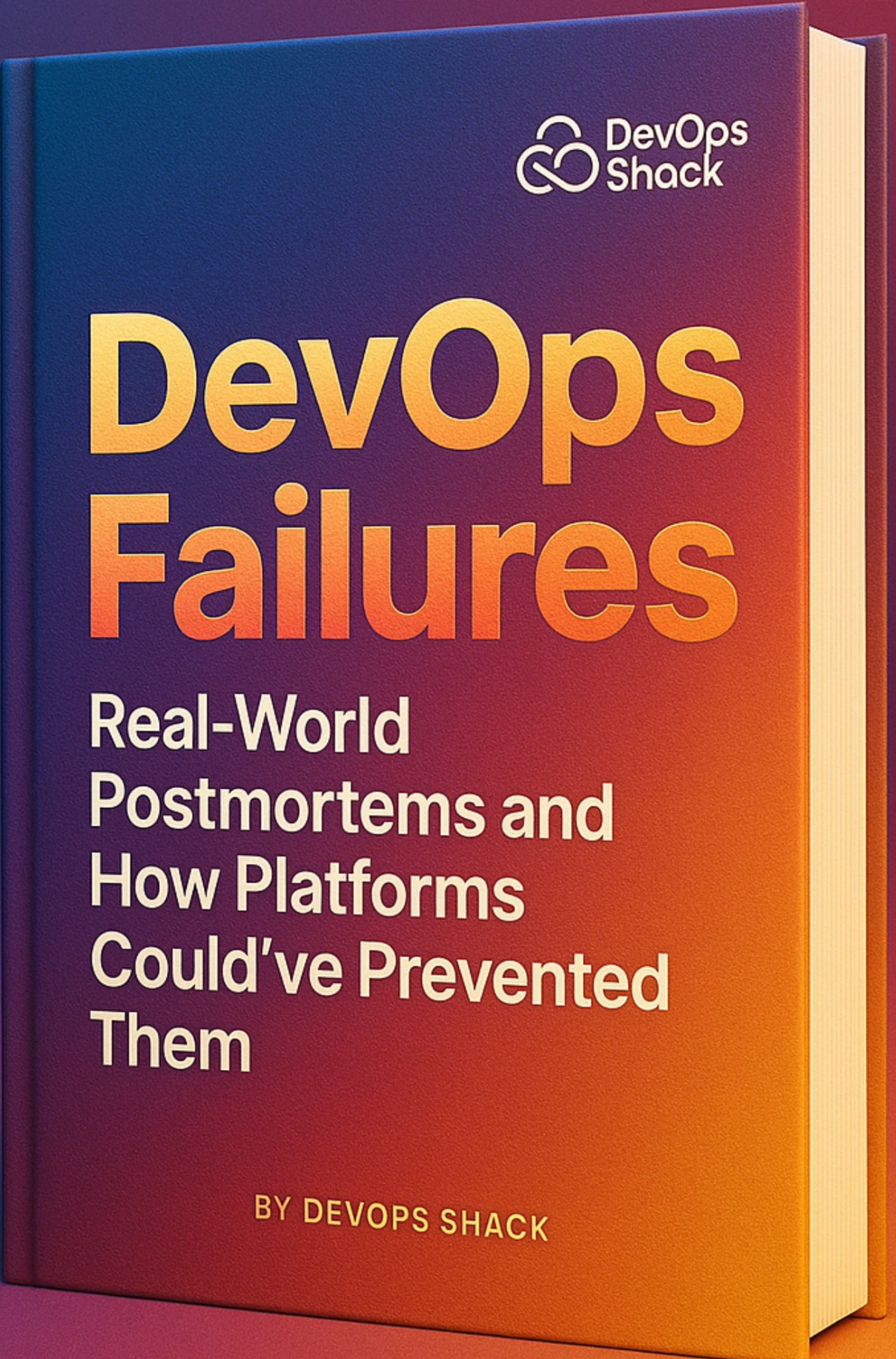

DevOps Shack

DevOps Failures

Real-World Postmortems and How Platforms Could've Prevented Them

BY DEVOPS SHACK

Click here for DevSecOps & Cloud DevOps Course

DevOps Shack

DevOps Failures:

Real-World Postmortems and How Platforms Could've Prevented Them

Table of Content.

- **3.2** Force-Push to Main – Developer overwrote good state with faulty commits
- **3.3** Incomplete RBAC Controls – Unauthorized changes merged to production branch

4. Secrets Leaked into Codebase

- **4.1** Hardcoded AWS Keys Pushed to Git – Exposed keys exploited within 10 minutes
- **4.2** Vault Misconfiguration – Secrets injected into logs via debug prints
- **4.3** Environment Variable Dump in Stack Trace – Secrets exposed in browser console

5. Containerization Blunders

- **5.1** Latest Tag Used in Production – New image introduced breaking change
- **5.2** Bloated Docker Image – Slow builds, cold starts, and security CVEs
- **5.3** Unscanned Image Pushed – Contained high CVE severity vulnerabilities

6. Misuse of Infrastructure as Code

- **6.1** Terraform Plan Not Reviewed – Destroyed live infrastructure unknowingly
- **6.2** Manual Patching Outside IaC – Configuration drift led to untraceable bugs
- **6.3** Secrets in Terraform State – Plaintext secrets stored in S3/backend

7. Downtime from Rolling Updates

- **7.1** No Readiness Probe – App served traffic before DB was ready
- **7.2** Misconfigured Liveness Probe – Caused crash loops across replicas

- **7.3** Full Pod Replacement – Stateful app crashed without graceful termination

8. Third-Party Dependency Failures

- **8.1** NPM Package Compromised – CI build passed but deployed malicious code
- **8.2** DockerHub Outage – Pipeline halted due to unavailable base image
- **8.3** External API Rate-Limiting – Service degraded due to no fallback/retry logic

9. Security & Access Control Oversights

- **9.1** Over-permissioned Service Accounts – Compromised pod accessed secrets
- **9.2** Open Ingress to the World – Exposed internal dashboard publicly
- **9.3** Missing Audit Trails – No visibility into who triggered destructive actions

10. Poor Incident Response & Communication

- **10.1** No Runbooks or Playbooks – On-call team guessed through recovery
- **10.2** Blame Culture in Postmortems – Root causes never fixed, just assigned
- **10.3** No Incident Management Tooling – Delays in escalation and response

1. CI/CD Pipeline Misconfigurations

CI/CD pipelines are the backbone of modern DevOps. However, even minor misconfigurations can lead to catastrophic failures in production, affecting thousands of users, damaging brand trust, or even leading to data loss.

Let’s explore 3 real-world inspired failures under this category and analyze how they could’ve been prevented using platforms like **GitHub Actions**, **Jenkins**, or **GitLab CI**, and by implementing key **DevSecOps** practices.

1.1 Accidental Deployment to Production

Incident

An engineer was testing a new feature in a feature branch. The pipeline was configured to deploy to production if a push was made to the main branch. Due to a Git mistake (git push origin feature:main), the feature branch was force-pushed to main and the CI/CD pipeline **automatically deployed to production**, breaking live user traffic.

Root Cause

- No branch protection on main.
- No approval workflow or manual gate before production deploy.
- No environment validation step.

Platform Fix

GitHub Actions / GitLab CI can include **environment protection rules** and manual approval gates.

Example Fix (GitHub Actions):

```
jobs:
  deploy-prod:
    if: github.ref == 'refs/heads/main'
    runs-on: ubuntu-latest
    environment:
      name: production
      url: https://yourapp.com
    steps:
      - name: Checkout
        uses: actions/checkout@v3

      - name: Manual Approval
        uses: hmarr/auto-approve-action@v2
        with:
          github-token: ${{ secrets.GITHUB_TOKEN }}
        # This can be replaced with an actual manual approval gate in enterprise setups

      - name: Deploy to Prod
        run: ./scripts/deploy.sh
```

☑ **GitHub Settings:**

- Enable **Branch Protection Rules** for main
- Enable **Required Approvals** for protected environments

1.2 Test Stage Skipped

Incident

In a Jenkins pipeline, a developer added a when condition to skip certain test jobs during weekend builds. Due to a typo (env.DAY == 'Saturday' instead of params.DAY), the tests **always skipped**, even during production builds. A major bug was deployed due to this oversight.

Root Cause

- Misconfigured conditional logic

- No default fallback for the skipped stage
- Lack of alerting when test stages were skipped

Platform Fix

Use **explicit failure** when test results are missing. Also, enforce mandatory test stages for all production workflows.

Example Jenkinsfile Fix:

```
pipeline {
 agent any
 stages {
  stage('Test') {
   when {
    not {
     branch 'hotfix/*'
    }
   }
   steps {
    echo 'Running Tests...'
    sh './run_tests.sh'
   }
  }
  stage('Deploy') {
   when {
    branch 'main'
   }
   steps {
    echo 'Deploying to Production'
    sh './deploy.sh'
```

```
      }
    }
  }
  post {
    always {
      junit '**/test-results.xml'
    }
    failure {
      mail to: 'devops@example.com',
           subject: 'Pipeline Failed',
           body: "Build ${env.BUILD_NUMBER} failed. Check Jenkins logs."
    }
  }
}
```

1.3 Wrong Artifact Promoted to Production

🔥 Incident

In a multistage pipeline, the staging artifact (v2.1-staging) was tagged and deployed to production by mistake. This caused features meant for internal review to go live prematurely.

🔬 Root Cause

- Inconsistent artifact tagging between stages
- No checksum verification or promotion gate
- Poor artifact metadata management

🛡 Platform Fix

Introduce **artifact promotion gates**, validate checksum, and tie artifact version to Git commit SHA or tag.

☑ **Example GitLab CI Artifact Promotion:**

```
stages:
  - build
  - test
  - staging
  - production

build:
  stage: build
  script:
    - docker build -t myapp:${CI_COMMIT_SHORT_SHA} .
    - docker save myapp:${CI_COMMIT_SHORT_SHA} > myapp.tar
  artifacts:
    paths:
      - myapp.tar

staging:
  stage: staging
  dependencies:
    - build
  script:
    - docker load < myapp.tar
    - ./deploy-staging.sh

production:
  stage: production
```

```
dependencies:
  - build
script:
  - echo "Deploying to production"
  - docker load < myapp.tar
  - ./deploy-prod.sh
when: manual
environment:
  name: production
  url: https://yourapp.com
```

🔐 **Bonus:**

- Sign your artifacts using **cosign** or **SLSA** framework
- Use **SBOMs** to verify integrity before deployment

☑ **Takeaways for Job Seekers**

1. Always implement **multi-stage validation** in CI/CD: test → staging → production.
2. Use **branch protections**, **manual approvals**, and **environmental safeguards**.
3. Learn YAML for GitHub Actions, Jenkinsfile syntax, and GitLab CI to **avoid misconfigurations**.
4. Understand the **flow of promotion** and how to enforce secure handoffs (using tags, checksums, or commit SHAs).

2. Lack of Observability & Alerting

When your systems are running live, silence is not a sign of health. It often means your **observability stack is broken**. Real-time monitoring, logging, and alerting form the **eyes and ears of DevOps**, and without them, you're flying blind.

Here are 3 real-world failure cases and how **platform-level observability practices** could have avoided them.

2.1 Silent Failures in Microservices

🔥 Incident

A Node.js-based microservice responsible for generating invoices silently failed due to a bad third-party API response. There were **no alerts**, and the service was technically "running" (the pod was up), but users weren't getting invoices for 5 days — until a customer raised a complaint.

🔬 Root Cause

- Application was not instrumented for **business-level metrics**
- Kubernetes health checks passed, but **logical failure** was not caught
- No Prometheus alert rule to detect drops in service output

🛡 Platform Fix

Instrument microservices using **Prometheus metrics** (e.g., invoices_generated_total). Use alert rules to detect abnormalities.

☑ Prometheus Instrumentation (Node.js):

```
const client = require('prom-client');
const register = new client.Registry();

const invoicesGenerated = new client.Counter({
  name: 'invoices_generated_total',
  help: 'Total number of invoices generated',
});

register.registerMetric(invoicesGenerated);
```

```
// In your service logic:
try {
  // ... generate invoice
  invoicesGenerated.inc();
} catch (e) {
  console.error("Invoice generation failed", e);
}
```

☑ **Prometheus Alert Rule:**

```
- alert: InvoiceGenerationDrop
  expr: rate(invoices_generated_total[5m]) < 1
  for: 10m
  labels:
    severity: critical
  annotations:
    summary: "Invoice generation is below expected rate"
    description: "Check the invoice microservice. Output rate is low."
```

2.2 No Logging Pipeline

🔥 Incident

A Python microservice crashed with unhandled exceptions, but logs were stored **only in the container**. By the time the team investigated, the pod had restarted, and **all logs were lost**. Root cause was unrecoverable.

🔬 Root Cause

- No central log aggregation
- No Fluent Bit / Filebeat sidecar
- Logs were not persisted outside pod lifespan

Platform Fix

Integrate a **logging pipeline** (e.g., Fluent Bit → Elasticsearch → Kibana).

✅ Fluent Bit DaemonSet (EKS):

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: fluent-bit-config
data:
  fluent-bit.conf: |
    [SERVICE]
        Flush        1
        Log_Level    info

    [INPUT]
        Name         tail
        Path         /var/log/containers/*.log
        Tag          kube.*

    [OUTPUT]
        Name  es
        Match *
        Host  elasticsearch.logging.svc.cluster.local
        Port  9200
        Logstash_Format On
```

✅ Logging Benefits:

- Access logs post-failure

- Alert on error logs (via Loki + Promtail)
- Forensic root cause analysis possible

2.3 Monitoring Gaps

Incident

A Kubernetes node in ap-south-1a zone ran out of disk space. Critical pods were evicted. However, **Grafana dashboards showed everything "green"** because node-exporter was down and Prometheus wasn't scraping metrics from that node.

Root Cause

- Node-exporter on that node failed silently
- No meta-monitoring for exporter health
- No alert rules for exporter downtime

Platform Fix

Monitor the **monitoring system itself**. Use **up** metric in Prometheus to detect exporter failures.

✓ Prometheus Alert Rule:

```
- alert: NodeExporterDown
  expr: up{job="node-exporter"} == 0
  for: 2m
  labels:
    severity: warning
  annotations:
    summary: "Node Exporter is down"
    description: "Exporter for {{ $labels.instance }} is not reporting metrics."
```

✓ Bonus: Blackbox Exporter

Use the **blackbox exporter** to ping endpoints, services, and exporters for real availability check.

✅ **Takeaways for Job Seekers**

1. Learn to **instrument applications** using Prometheus-compatible metrics libraries.
2. Understand **Loki, Fluent Bit, and ELK stack** for logging pipelines.
3. Always **monitor your monitoring system**. Set alerts for exporters and data freshness.
4. Practice writing **alerting rules** in PromQL and visualize metrics via Grafana.

3. GitOps Gone Wrong

GitOps is a powerful DevOps paradigm where **Git becomes the single source of truth** for your infrastructure and application state. But with great power comes great risk—especially when auto-syncing broken configurations or bypassing safeguards.

Here are 3 real-world failures, how they happened, and how platforms like **ArgoCD**, **Flux**, and **GitHub** could've prevented them.

3.1 ArgoCD Auto-Sync Gone Wild

Incident

A developer mistakenly committed a Kubernetes manifest that **removed resource limits** and changed the replicas: 3 to 0. ArgoCD, set to **auto-sync**, immediately applied the manifest. All 3 replicas of the payment service were terminated, and the production payment gateway went offline for 20 minutes.

Root Cause

- ArgoCD auto-sync applied bad config without human review
- No resource validation or policy enforcement (like OPA/Gatekeeper)
- No Git PR review – direct push to main branch

Platform Fix

Use **manual sync with auto-drift detection**, and **OPA/Gatekeeper** to reject dangerous configurations.

✓ Recommended ArgoCD Config (disable auto-sync in prod):

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: payment-app
spec:
  project: default
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
    syncOptions:
```

```
  - CreateNamespace=true
  - ApplyOutOfSyncOnly=true
destination:
  namespace: prod
  server: https://kubernetes.default.svc
source:
  repoURL: https://github.com/org/repo
  path: k8s/payment
  targetRevision: HEAD
```

✅ **Best Practice:**

- Use **PR-based workflow** to modify manifests
- Implement **validation webhooks or OPA** to block dangerous changes
- Use **ArgoCD's app diff** tools in UI for visibility

3.2 Force-Push to Main Wipes Stable State

🔥 **Incident**

A junior developer accidentally force-pushed their local branch to main with:

```
git push origin feature/login --force
```

This overwrote the main branch history. ArgoCD detected changes and re-deployed manifests based on outdated or broken states. Production was rolled back to a 3-week-old config.

🔬 **Root Cause**

- No protection against force-push on main
- GitOps system (ArgoCD/Flux) synced blindly
- No Git commit signature enforcement or PR-based gating

🛡 **Platform Fix**

Enable **branch protection rules**, **commit signing**, and **only merge via PRs** with reviews.

✅ **GitHub Branch Protection Settings:**

- ✅ Require pull request reviews before merging
- ✅ Require status checks to pass
- ✅ Restrict who can push to branch
- ✅ Require signed commits

✅ **Bonus: GitHub Actions for PR Policy**

```
name: Enforce PR Checks


on: pull_request


jobs:
  validate:
    runs-on: ubuntu-latest
    steps:
      - name: Validate K8s Manifests
        run: |
          kubeval ./manifests/
```

3.3 Incomplete RBAC Controls in GitOps

🔥 **Incident**

A team member with write access to the Git repo (but not intended for production) pushed a manifest update that changed a database service from ClusterIP to LoadBalancer, unintentionally exposing the DB to the internet.

This got **synced automatically via ArgoCD**.

🔬 **Root Cause**

- Git repository access was not scoped per environment
- No approval process for sensitive files
- No secret detection or scanning in pull requests

🛡 **Platform Fix**

- Use **separate Git repos or branches** for environments
- Enforce **fine-grained Git permissions** (e.g., CODEOWNERS)
- Use **Gitleaks** or **Snyk** to scan PRs for secrets/misconfig

☑ **CODEOWNERS Example:**

```
# Require senior DevOps to review all prod configs
/prod/ @lead-devops @platform-team
```

☑ **Gitleaks GitHub Action:**

```
name: Detect Secrets

on: [push, pull_request]

jobs:
  gitleaks:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Run Gitleaks
        uses: zricethezav/gitleaks-action@v2
        with:
          config-path: .gitleaks.toml
```

☑ **Takeaways for Job Seekers**

1. GitOps is powerful—but dangerous when used blindly. Learn tools like **ArgoCD, Flux, and Helm**.
2. Always use **branch protection**, **PR reviews**, and **commit validation**.
3. Set up **OPA/Gatekeeper policies** to block dangerous resource changes.
4. Understand how auto-sync works in ArgoCD and when to use **manual sync** for sensitive apps.
5. Learn **Gitleaks**, **kubeval**, and **Snyk** to automate safety in Git workflows.

4. Secrets Leaked into Codebase

One of the **most frequent and high-impact failures** in DevOps workflows is leaking secrets—such as API keys, passwords, tokens, and private certs—into the codebase. Once committed, these secrets can be harvested within **seconds** by bots scanning public and private repositories.

Let's walk through real-world incidents and how platforms like **Gitleaks**, **Vault**, and **GitHub** could have helped prevent them.

4.1 Hardcoded AWS Keys Pushed to GitHub

Incident

A developer added the following to config.js for quick testing:

```
const awsConfig = {
  accessKeyId: "AKIAIOSFODNN7EXAMPLE",
  secretAccessKey: "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY",
};
```

They forgot to remove it before committing. The repository was public, and **within 6 minutes**, the credentials were exploited to spin up **EC2 Bitcoin miners**, costing over ₹1.5 lakh in under 2 hours.

Root Cause

- Secrets were hardcoded
- No secret scanning tools (like Gitleaks) in place
- No commit hook or Git pre-push validation
- Repo was public

Platform Fix

Use **Gitleaks**, **truffleHog**, or **GitHub's secret scanning** (enabled by default on public repos).

☑ Gitleaks GitHub Action:

```
name: Secret Scan


on: [push, pull_request]


jobs:
  gitleaks:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Run Gitleaks
        uses: zricethezav/gitleaks-action@v2
```

✅ **Bonus: Git Pre-commit Hook**

```
#!/bin/bash
echo "Scanning for secrets before committing..."
gitleaks protect --staged --verbose
```

✅ **Preventative Steps:**

- Enable **secret scanning alerts** on GitHub repo
- Use **.gitignore** to avoid committing .env and config files
- Rotate secrets immediately if exposure is detected

4.2 Vault Misconfiguration Leaked Secrets via Logs

🔥 **Incident**

In a Kubernetes-based setup using **HashiCorp Vault**, secrets were injected via environment variables into pods. A developer enabled DEBUG=true in a Node.js app, and one stack trace printed the environment:

```
Error: DB connection failed
Environment:
  DB_PASSWORD = mySuperSecretPassword123
```

This output was visible in the logs, which were shipped to ELK, and later visible in **Kibana** to anyone with read access.

🔬 **Root Cause**

- Secrets injected as environment variables

- No audit on debug logs
- Vault injector improperly configured
- No separation between debug and prod log levels

🛡 **Platform Fix**

Use **Vault Agent Sidecar Injector** with **file-based secret mounts** instead of environment vars.

☑ **Vault Injector Example (K8s):**

```
annotations:
  vault.hashicorp.com/agent-inject: "true"
  vault.hashicorp.com/role: "app-role"
  vault.hashicorp.com/agent-inject-secret-db: "secrets/data/db"
  vault.hashicorp.com/agent-inject-template-db: |
    {{- with secret "secrets/data/db" -}}
    DB_PASSWORD={{ .Data.data.password }}
    {{- end }}
```

☑ **Logging Best Practices:**

- Mask sensitive keys in logs
- Use centralized logging alerts for "password=", "token=", etc.
- Avoid logging ENV or full configs in error messages

4.3 Secrets Dumped in Stack Trace on Frontend

🔥 **Incident**

An Angular frontend made a REST call to a backend service. The backend hit an unexpected 500 error and dumped the entire config object—including a token and internal DB string—into the response:

```
{
  "error": {
    "message": "Something broke",
    "config": {
      "Authorization": "Bearer sk_live_super_secret_token",
      "dbConnection": "mongodb://user:pass@host"
    }
  }
}
```

Google indexed the exposed page before the issue was discovered.

🔬 Root Cause

- Poor error handling
- Exposed config in JSON response
- No frontend rate-limiting or sanitization

🛡 Platform Fix

- Use a **custom error handler middleware** to sanitize responses
- Never return full config objects in API error payloads
- Validate outgoing HTTP error responses using middleware filters

☑ Node.js Express Middleware Example:

```
app.use((err, req, res, next) => {
  console.error("Server Error:", err.message); // Internal logging only
  res.status(500).json({
    message: "Internal Server Error", // Don't leak internal info
  });
});
```

☑ Takeaways for Job Seekers

1. Always use **Vault, AWS Secrets Manager, or Azure Key Vault** to manage secrets

2. Never store secrets in .env, source code, or Git history
3. Learn and use **Gitleaks**, **pre-commit hooks**, and **CI secret scans**
4. Know how to implement **Vault injector in Kubernetes** and the **risks of ENV-based secret injection**
5. Sanitize all error messages—especially in public APIs and UIs

5. Containerization Blunders

Containers bring consistency and portability, but poor containerization practices can lead to **bloated images**, **security holes**, **broken deployments**, and **downtime**. Many DevOps failures in production stem from **how images are built, tagged, and deployed**.

Here are 3 real-world failure scenarios and how best practices and DevSecOps tooling could've avoided them.

5.1 'latest' Tag Used in Production

Incident

The Dev team pushed a new build of the frontend container and tagged it with latest:

```
docker build -t myapp/frontend:latest .

docker push myapp/frontend:latest
```

In production, the Kubernetes manifest used the image:

```
image: myapp/frontend:latest
```

Auto-deploy was triggered, and Kubernetes pulled the new (but buggy) image without any version control. UI went blank for thousands of users due to a missing environment variable in the new image.

Root Cause

- Usage of latest tag (non-deterministic, mutable)
- No version pinning or rollback strategy
- Kubernetes pulled the new image thinking it was the same

Platform Fix

Always tag images with **commit SHA**, **semantic version**, or **build number**, and **disable image pull if not changed**.

✓ Docker Build Example (Using Git SHA):

```
GIT_SHA=$(git rev-parse --short HEAD)

docker build -t myapp/frontend:$GIT_SHA .

docker push myapp/frontend:$GIT_SHA
```

✓ Kubernetes Manifest (Avoid latest):

```
containers:

 - name: frontend
```

```
  image: myapp/frontend:1.2.3
  imagePullPolicy: IfNotPresent
```

☑ **Bonus: Use ArgoCD/Flux with Image Updater Controllers**

5.2 Bloated Docker Images

Incident

A Java Spring Boot application was containerized using a full Ubuntu base image with Maven, Git, and unused packages. The final image was **2.1GB**, causing:

- Long build times in CI
- Slow cold-starts in Kubernetes
- Higher storage and network costs
- More CVEs in image scans

Root Cause

- No multi-stage build
- Used heavy base image (ubuntu instead of distroless or alpine)
- Included dev dependencies in prod image

Platform Fix

Use **multi-stage builds** and minimal base images like alpine or distroless.

☑ **Multi-Stage Dockerfile for Java:**

```
# Build Stage
FROM maven:3.9.1-eclipse-temurin-17 AS builder
WORKDIR /app
COPY . .
RUN mvn clean package -DskipTests

# Final Stage
```

```
FROM eclipse-temurin:17-jre

WORKDIR /app

COPY --from=builder /app/target/app.jar ./app.jar

ENTRYPOINT ["java", "-jar", "app.jar"]
```

✅ **Result**: Image size reduced from **2.1GB → ~300MB**

5.3 Unscanned Image Pushed to Production

🔥 **Incident**

A Python app was containerized and pushed to production without image scanning. Later, a security audit revealed the image included a **Python 3.6 base with a known high-severity CVE** in urllib3, which allowed remote code execution via SSRF. The app was **exploitable** for weeks.

🔬 **Root Cause**

- No container scanning before push/deploy
- Used outdated base image
- No CI integration with scanners like **Trivy**, **Grype**, or **Snyk**

🛡 **Platform Fix**

Integrate container scanning in your CI/CD pipeline using **Trivy**.

✅ **GitHub Actions + Trivy Example:**

```
name: Scan Docker Image


on: push


jobs:
  trivy-scan:
    runs-on: ubuntu-latest
    steps:
```

```
- name: Checkout
  uses: actions/checkout@v3

- name: Build Docker Image
  run: docker build -t myapp:latest .

- name: Scan with Trivy
  uses: aquasecurity/trivy-action@master
  with:
    image-ref: myapp:latest
    format: table
    exit-code: 1
    ignore-unfixed: true
```

☑ You can also enforce **policy-as-code** using tools like **OPA** to reject builds with critical CVEs.

☑ **Takeaways for Job Seekers**

1. **Never use latest tag in production**. Always pin image versions.
2. Use **multi-stage builds** to keep images lean and production-ready.
3. Understand and use image scanners like **Trivy, Grype, or Snyk** in CI.
4. Know how to use **Alpine**, **distroless**, or **slim** base images.
5. Use GitHub Actions, Jenkins, or GitLab CI to **automate security scans** pre-deploy.

6. Misuse of Infrastructure as Code (IaC)

Infrastructure as Code (IaC) empowers teams to define infrastructure through code. But when used incorrectly, it can be as dangerous as running rm -rf / on a production server. A small mistake can bring down entire environments or expose critical infrastructure.

Let's explore 3 real-world misuses of IaC and how tools like **Terraform**, **Pulumi**, **CloudFormation**, and CI/CD checks can help prevent disaster.

6.1 Terraform Plan Not Reviewed – Production Infrastructure Destroyed

🔥 Incident

A developer accidentally changed the backend config of a Terraform script from:

```
backend "s3" {
  bucket = "dev-iac-state"
}
```

to:

```
bucket = "prod-iac-state"
```

They then ran terraform init and apply in the **dev environment**, which reused the **prod state**, resulting in the **entire production infrastructure being destroyed and replaced** with dev configurations.

🔬 Root Cause

- No separation of state files
- terraform apply run without peer review
- No plan validation or approval before apply

🛡 Platform Fix

Use **CI/CD for terraform plan**, manual approval for apply, and isolate state files per environment.

☑ GitHub Actions Example:

```
name: Terraform Plan


on:
  pull_request:
    paths:
      - 'infra/**'
```

```
jobs:
  terraform-plan:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Terraform Init
        run: terraform init -backend-config=envs/prod.tfbackend
      - name: Terraform Plan
        run: terraform plan -out=tfplan
```

☑ Store prod.tfbackend and dev.tfbackend separately to avoid confusion.

☑ **Bonus:**

Use terraform workspace or separate backend buckets for isolation:

```
terraform workspace new dev
terraform workspace select prod
```

6.2 Manual Patching Outside IaC – Configuration Drift

Incident

A cloud engineer patched a Kubernetes node manually using the CLI for a quick fix:

```
kubectl label node node-1 foo=bar
```

This change **was never reflected in the Terraform or Helm chart**, and when the cluster was later re-provisioned using IaC, the patch was lost. The node stopped receiving workloads, breaking affinity-based services.

Root Cause

- Manual changes outside IaC caused **configuration drift**
- No drift detection mechanism

- Terraform or Helm didn't have full ownership of state

🛡 **Platform Fix**

- Run **terraform plan regularly** to detect drift
- Treat IaC as the **only source of truth**
- Disallow manual cloud resource changes in production (enforce through policies)

☑ **Best Practice:**

- Use tools like **Terraform Cloud Drift Detection**
- Adopt **InfraMon** or **OPA policies** that restrict non-IaC changes
- Build guardrails: make kubectl access read-only for most users in prod

6.3 Secrets Stored in Terraform State Files

🔥 **Incident**

A junior DevOps engineer added database credentials directly into Terraform:

```
resource "aws_db_instance" "main" {
  username = "admin"
  password = "SuperSecret123!"
}
```

Terraform stored the value in terraform.tfstate, which was then synced to an **S3 bucket with misconfigured public access**, exposing all credentials.

🔬 **Root Cause**

- Secrets hardcoded in Terraform
- No Vault or parameter store used
- tfstate file exposed without encryption/access control

🛡 **Platform Fix**

Store secrets in **Vault**, **AWS SSM Parameter Store**, or **Azure Key Vault**, and reference them using **data sources**.

☑ **Using AWS SSM Secure Parameter:**

```
data "aws_ssm_parameter" "db_password" {
  name = "/prod/db/password"
  with_decryption = true
}

resource "aws_db_instance" "main" {
  password = data.aws_ssm_parameter.db_password.value
}
```

☑ **Protect State Files:**

- Encrypt state file (e.g., SSE-S3 or KMS)
- Restrict access using IAM policies
- Store state in remote backends with locking (e.g., S3 + DynamoDB for locking)

☑ **Takeaways for Job Seekers**

1. IaC is powerful—but dangerous when misused. Always peer-review terraform plan.
2. Keep state files **isolated per environment**. Never reuse prod state in dev/test.
3. Avoid manual infra changes. Detect and fix **configuration drift** regularly.
4. Store secrets in **Vault**, **SSM**, or **Key Vault**—never in code or state files.
5. Master IaC tools like **Terraform, Pulumi, and CloudFormation**, and integrate them securely into CI/CD pipelines.

7. Downtime from Rolling Updates

Rolling updates are designed to **avoid downtime** during application deployments by updating pods one by one. But without proper configuration, they can cause **availability issues**, **crash loops**, or even full-scale service outages.

Let's look at 3 real-world failures caused by rolling update misconfigurations and how Kubernetes + CI/CD tools could've helped prevent them.

7.1 No Readiness Probe – App Served Before It Was Ready

🔥 Incident

A new version of a Go-based web API was deployed using a rolling update. The pod started quickly, but it took 10 seconds for the database connection to initialize.

Kubernetes marked the pod as "ready" immediately (because no readiness probe was configured), and the **load balancer routed live traffic** to it. Users saw 500 errors for the first few seconds.

🔬 Root Cause

- No readinessProbe defined
- App marked as ready too early
- Traffic routed before DB or cache was ready

🛡 Platform Fix

Use **readiness probes** to delay traffic until the pod is truly ready to serve.

✅ Kubernetes YAML:

```yaml
readinessProbe:
  httpGet:
    path: /healthz
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 5
  failureThreshold: 3
```

✅ Pro tip: Implement a **custom /healthz route** that checks DB connection, cache status, and app health.

7.2 Misconfigured Liveness Probe – Caused Crash Loop

🔥 Incident

A Node.js app had a liveness probe pointing to /health, but the route was behind an auth middleware. Since the probe didn't pass credentials, it always returned 401.

Kubernetes assumed the container was **unhealthy**, and restarted it continuously, creating a **CrashLoopBackOff**.

🔬 **Root Cause**

- Liveness probe hit a protected route
- No exception for health checks
- Misleading status led to infinite restarts

🛡 **Platform Fix**

Use **dedicated unauthenticated health routes**, and monitor liveness properly.

☑ **Fix:**

1. Create a **public /liveness endpoint**
2. Don't include auth, DB calls, or heavy logic
3. Keep it lightweight and isolated

☑ **Liveness Probe Example:**

```
livenessProbe:
  httpGet:
    path: /liveness
    port: 8080
  initialDelaySeconds: 10
  periodSeconds: 10
  timeoutSeconds: 2
```

☑ In backend code:

```
app.get('/liveness', (req, res) => res.sendStatus(200));
```

7.3 Full Pod Replacement – Stateful App Crashed

Incident

An update to a Redis StatefulSet mistakenly removed the volumeClaimTemplates, causing the new pods to start **without persistent storage**. During the rolling update, **all previous pods were replaced**, resulting in complete **data loss**.

The app booted with empty state, and users saw missing carts, sessions, and queue data.

Root Cause

- Volume definition removed from StatefulSet
- No pod-level backup or rollback
- Rolling update replaced all pods without protection

Platform Fix

- Always use rollingUpdate with partition for **gradual rollout**
- Use **PodDisruptionBudgets (PDB)** to avoid full replacements
- Add **volumeMount checks** in readiness/liveness probes

☑ StatefulSet Rolling Update Strategy:

```yaml
updateStrategy:
  type: RollingUpdate
  rollingUpdate:
    partition: 1  # Only one pod updated at a time
```

☑ PodDisruptionBudget:

```yaml
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: redis-pdb
spec:
  minAvailable: 2
```

```
selector:
  matchLabels:
    app: redis
```

☑ Back up PVCs using velero or snapshot tools before stateful updates.

☑ **Takeaways for Job Seekers**

1. Learn how readinessProbe, livenessProbe, and startupProbe work.
2. Never deploy production apps without health probes in place.
3. Understand the difference between **stateless vs stateful apps** and how updates affect each.
4. Use **PodDisruptionBudgets**, maxUnavailable, and partition settings to prevent full downtime.
5. Practice writing and testing probes locally before deploying to staging/prod.

8. Third-Party Dependency Failures

Modern applications rely heavily on third-party services—like external APIs, open-source packages, container registries, and cloud services. When these dependencies fail, and we haven't planned for failure, **our systems crash too**.

Here are 3 real-world examples where reliance on third-party services caused failures, and how to build **resilient, dependency-aware DevOps pipelines** to mitigate them.

8.1 Compromised NPM Package Deployed to Production

Incident

An app used the ua-parser-js NPM package. One day, the team deployed a build, and hours later found that **their app was injecting malware** on client browsers. Turns out, a malicious actor had gained access to the maintainer's account and **pushed a trojanized version** of the package to NPM.

It went unnoticed until a Reddit thread exposed it, but thousands of users were already compromised.

Root Cause

- No lockfile validation (package-lock.json)
- No software supply chain scanning
- No SBOM or SLSA validation

Platform Fix

- Use **dependency lockfiles**, verify package signatures
- Integrate **Snyk**, **OWASP Dependency-Check**, or **Trivy SBOM** in your pipeline
- Track dependencies using **Software Bill of Materials (SBOM)**

✓ GitHub Action with Snyk:

```
jobs:
 snyk:
  runs-on: ubuntu-latest
  steps:
   - uses: actions/checkout@v3
   - uses: snyk/actions/node@master
    env:
     SNYK_TOKEN: ${{ secrets.SNYK_TOKEN }}
```

✓ Generate SBOM with Trivy:

```
trivy sbom --format cyclonedx --output sbom.json .
```

✅ Upload SBOM to registry or store for audit.

8.2 DockerHub Outage Stalled CI/CD Pipelines

🔥 **Incident**

A DockerHub outage took down image pull operations globally. The CI/CD pipeline was trying to pull node:14-alpine, nginx:latest, and a few custom base images.

All builds failed, and deployments were halted for hours.

🔬 **Root Cause**

- No image mirroring or caching
- Heavy reliance on DockerHub without fallback
- latest tag used (no deterministic cache)

🛡 **Platform Fix**

- Use **private image registries** or mirror DockerHub images
- Use **versioned tags** and **cache layers** in CI
- Host frequently used base images in **ECR, GCR, or Harbor**

✅ **Example: Pull from your own registry**

```
image: registry.example.com/base/node:14.21.3-alpine
```

✅ **Mirror Script (basic):**

```
docker pull node:14-alpine

docker tag node:14-alpine my-registry.com/node:14-alpine

docker push my-registry.com/node:14-alpine
```

✅ Use **Artifactory**, **Harbor**, or **GitHub Container Registry (GHCR)** for reliability.

8.3 External API Rate Limiting Crashed Payment Flow

🔥 **Incident**

The backend service for checkout used a **third-party currency conversion API** on every transaction. During a flash sale, the API rate-limited the app (HTTP 429). Since there was **no retry, circuit breaker, or fallback**, the checkout failed for thousands of users.

🔬 Root Cause

- Tight coupling to 3rd-party API
- No retry logic or circuit breaker
- No caching of recent responses

🛡 Platform Fix

- Use **resilience libraries** like **resilience4j**, **Hystrix**, or **retry decorators**
- Implement **circuit breakers** and **exponential backoff**
- Cache static API responses for high-traffic operations

☑ Node.js Retry with Axios:

```
const axiosRetry = require('axios-retry');
axiosRetry(axios, { retries: 3, retryDelay: axiosRetry.exponentialDelay });


axios.get('https://api.currency.com/rates')
 .then(...)
 .catch(err => {
   // fallback logic
 });
```

☑ Python Retry with Backoff:

```
import requests
import backoff


@backoff.on_exception(backoff.expo, requests.exceptions.RequestException,
max_tries=3)
```

```
def get_rate():
    return requests.get("https://api.currency.com/rates")
```

✅ Combine with **Redis caching** or **sidecar proxy (e.g., Envoy)** to buffer requests.

✅ **Takeaways for Job Seekers**

1. Always scan third-party dependencies—npm, pip, Maven, Docker—with tools like **Snyk**, **Trivy**, or **OWASP Dependency-Check**.
2. Use **SBOMs and supply chain metadata** to track and verify dependencies.
3. Host **frequently-used base images** internally to avoid external outages.
4. For external APIs, use **retries**, **fallbacks**, and **circuit breakers** to handle failure gracefully.
5. Use **cache-first** strategies for predictable API responses in high-traffic zones.

9. Security & Access Control Oversights

Security is often an afterthought in fast-paced DevOps environments. But the **lack of proper access control**, **over-permissioned accounts**, and **exposed endpoints** can lead to **data breaches**, **service outages**, and **regulatory violations**.

Here are 3 real-world examples of security failures and how platforms like **Kubernetes RBAC**, **IAM**, **OPA/Gatekeeper**, and **Zero Trust** principles could have prevented them.

9.1 Over-Permissioned Kubernetes Service Account Compromised

Incident

A compromised microservice container (through an RCE vulnerability) was running with a **default service account** that had **cluster-admin** privileges. The attacker used the service account token to:

- List all secrets in the cluster
- Deploy a crypto-mining container
- Exfiltrate credentials via a sidecar

The cluster was **fully compromised** and had to be rebuilt from scratch.

Root Cause

- Default service account not scoped
- RBAC rules were overly permissive
- No pod-level security restrictions

Platform Fix

- Apply **least privilege** to service accounts using RBAC
- Use **PodSecurityContext**, **PSP**, or **OPA Gatekeeper**
- Disable default token mounting if not needed

☑ Kubernetes RBAC Example:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: dev
  name: read-pods
rules:
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["get", "list"]
```

```
---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods-binding
  namespace: dev
subjects:
  - kind: ServiceAccount
    name: myapp-sa
    namespace: dev
roleRef:
  kind: Role
  name: read-pods
  apiGroup: rbac.authorization.k8s.io
```

☑ Also use automountServiceAccountToken: false when tokens are not needed.

9.2 Open Ingress Exposed Internal Dashboard

Incident

An internal dashboard (e.g., Jenkins or Prometheus) was exposed via an Ingress like:

```
host: jenkins.company.com
```

But **no auth or IP whitelisting** was configured. A Google dork ("intitle:Jenkins") led attackers to the dashboard. They accessed the console and ran a build job that deployed a reverse shell container.

Root Cause

- Ingress exposed to public internet

- No authentication/authorization layer
- No WAF or IP restrictions

Platform Fix

- Always protect dashboards behind **auth layers** (OAuth, Basic Auth, SSO)
- Add **IP whitelisting**, **firewall rules**, or **private ingress**
- Use tools like **oauth2-proxy** + Ingress NGINX

Secure Ingress Example:

```
annotations:
  nginx.ingress.kubernetes.io/auth-url:
"https://oauth.example.com/oauth2/auth"
  nginx.ingress.kubernetes.io/auth-signin:
"https://oauth.example.com/oauth2/start?rd=$request_uri"
```

✓ Use **external-dns + private zones** for internal-only services

✓ Enforce ingress policies using **OPA Gatekeeper**:

```
deny[msg] {
  input.kind.kind == "Ingress"
  not input.spec.rules[*].host == "internal.company.local"
  msg := "Ingress must use internal-only domains"
}
```

9.3 No Audit Trails or Change Logging

Incident

An engineer manually updated a production Kubernetes Deployment via kubectl edit, changing the image from v1.5 to v2.0-beta.

The deployment failed, but **no one knew who made the change**. There was no audit trail, no GitOps flow, and no approval record. It took hours to identify and revert.

Root Cause

- Manual edits using kubectl
- No centralized audit logging
- No Git history of change

◐ **Platform Fix**

- Enforce GitOps workflows (ArgoCD/Flux)
- Use **audit logging** (enabled by default in Kubernetes 1.13+)
- Block direct kubectl apply in production using **OPA policies**

☑ **ArgoCD GitOps Flow:**

- All changes go through Git PRs
- No in-cluster config drift allowed
- Automatic rollback or drift sync

☑ **Kubernetes Audit Log Sample:**

```
{
 "user": {
  "username": "devops@example.com"
 },
 "verb": "patch",
 "objectRef": {
  "resource": "deployments",
  "name": "frontend"
 },
 "responseStatus": {
  "code": 200
 }
}
```

☑ Store logs in **CloudTrail**, **SIEM**, or **Grafana Loki**

✅ Enforce commit-signing and PR-based deployments only

✅ **Takeaways for Job Seekers**

1. Understand Kubernetes RBAC, service accounts, and principle of least privilege
2. Always secure your Ingress with **auth**, **TLS**, and **network policies**
3. Avoid manual changes to production—use GitOps, audits, and automation
4. Learn tools like **OPA Gatekeeper**, **oauth2-proxy**, and **CloudAudit**
5. Security is not optional. You need to **bake it into your CI/CD and platform architecture**

10. Poor Incident Response & Communication

A well-prepared DevOps team isn't just measured by uptime—it's measured by **how fast and effectively they respond when things break**. Poor incident handling, lack of communication, and blame culture often make minor issues spiral into **major disasters**.

Here are 3 real-world scenarios where poor incident response and communication caused more damage than the original issue—and how to prevent it.

10.1 No Runbooks or On-Call Playbooks Available

🔥 **Incident**

At 2:30 AM, the alerting system triggered high CPU usage on the database. The on-call engineer had **no documentation or SOPs**, didn't know where logs were, and couldn't identify which microservice was spamming the DB.

It took **3 hours to resolve**, causing major service degradation and customer complaints. The issue? A new background job introduced in the previous release.

🔬 Root Cause

- No incident response documentation
- No observability dashboards linked to services
- No handover between shifts/on-call engineers

🛡 Platform Fix

- Create and maintain **runbooks/playbooks per service**
- Link **alerts to dashboards**, logs, and remediation steps
- Use tools like **Incident.io**, **FireHydrant**, or **PagerDuty Runbooks**

☑ Example Runbook Entry (Markdown):

```
### Incident: High DB CPU Usage


**Alert Name:** Postgres CPU > 90%

**Dashboards:** [Grafana Link](https://grafana.company.com/postgres)

**Logs:** View in [Loki](https://logs.company.com/postgres)


**Checklist:**

- [ ] Check slow queries

- [ ] Identify service causing load (`top_queries.sql`)

- [ ] Restart problematic job using `kubectl rollout restart`


**Owner:** @infra-team
```

✅ Store these in Git, linked to Grafana/PagerDuty alerts

10.2 Blame Culture in Postmortems – No Real Fixes

Incident

A service went down due to a bad deployment. In the postmortem, the focus was on **"who broke it"** instead of **"what broke and why."** The junior developer who pushed the code was publicly blamed in Slack. The same incident repeated a month later—this time from someone else.

Root Cause

- No blameless postmortems
- No RCA or follow-ups
- Psychological safety was compromised

Platform Fix

- Conduct **blameless postmortems**
- Focus on systems, not people
- Use tools like **Jellyfish**, **Rootly**, or **Google RCA templates**

✅ **RCA Template:**

```
## Blameless Postmortem


### Summary:

Deployment caused downtime on payment-service


### Impact:

- 20 min of failed checkouts

- ~₹3.5L revenue impact


### Timeline:
```

- 3:20 PM - Deploy initiated
- 3:22 PM - Alert fired
- 3:40 PM - Rollback initiated
- 3:55 PM - System stable

Root Cause:

- Feature flag default was "on" in production config

Action Items:

- [ ] Add feature flag validation in CI
- [ ] Update deployment checklist
- [ ] Review flag defaults before release

☑ Share postmortems internally & track if action items are closed

10.3 No Incident Management Tooling – Escalation Delays

🔥 **Incident**

A major outage was reported by users. The SRE team had no central tool to track incidents. Alerts came via Slack, email, and calls. No one knew:

- Who's on-call
- What severity the incident was
- Who's leading the response

Resolution took 4 hours longer than necessary.

🔬 **Root Cause**

- No incident commander
- No war room or comms channel
- No automated alert → response flow

🛡 **Platform Fix**

- Use incident management platforms: **PagerDuty**, **Opsgenie**, **FireHydrant**, **Incident.io**
- Set up **Slack integrations** for auto-alerts and war rooms
- Use **SRE Incident Roles**: Commander, Communicator, Scribe, Resolver

☑ **Example Workflow (PagerDuty + Slack):**

1. Alert fires in Prometheus
2. Alertmanager sends to PagerDuty
3. PagerDuty triggers Slack bot:

🚨 Incident #2432 triggered

- Service: Checkout API

- Severity: P1

- Commander: @oncall

- War Room: #incident-2432

☑ Start tracking incident timeline and updates in real time

☑ **Takeaways for Job Seekers**

1. Learn how to **write and follow runbooks**, and where to keep them
2. Understand **incident lifecycle**: detection → triage → mitigation → postmortem
3. Always practice **blameless RCA**. Focus on fixing systems, not blaming people
4. Learn how to use incident response platforms (PagerDuty, Opsgenie, FireHydrant)
5. Incident handling is a **key DevOps maturity signal**—study real postmortems from Google, Netflix, GitHub, etc.

Conclusion:

DevOps isn't just about automation, pipelines, or Kubernetes YAMLs. It's about building **resilient systems** that fail gracefully, **alert the right people**, and recover quickly without finger-pointing.

The real-world failures covered in this guide—from botched deployments to leaked secrets, from insecure infrastructure to broken incident response—highlight a simple truth:

⚠ **Every DevOps mistake is a teaching moment.** The only real failure is not learning from it.

In this document, we examined **10 categories of DevOps failures**:

- From **CI/CD misconfigurations** to **rolling update disasters**
- From **GitOps misuses** to **third-party dependency outages**
- From **secrets exposure** to **lack of observability**
- From **IaC gone wrong** to **security oversights**

- Ending with **poor incident response & communication**

Each failure taught us:

- The **root cause**
- The **cost** (downtime, breach, broken trust)
- The **platform-level fix** (tools, configs, pipelines)
- And most importantly — the **lesson** for every DevOps engineer and job seeker