DevOps Troubleshooting

Real-World Scenarios & Solutions

10 Critical Issues Every DevOps Engineer Must Know

From Terraform State Locks to Kubernetes Debugging

Sandhya | DevOps Engineer

Building resilient cloud infrastructure

Scenario 1: Terraform Concurrent Apply ⚠️

What Happens When Two Engineers Run terraform apply Simultaneously?

The Problem

🚨 **Real-World Scenario**

Engineer A starts `terraform apply` to add a new EC2 instance. Simultaneously, Engineer B runs `terraform apply` to modify a security group. What happens?

❌ **Without State Locking**

Disaster Scenarios:

- **State Corruption:**

  Both writes compete, state file becomes invalid
- **Duplicate Resources:**

  Same resource created twice with different IDs
- **Lost Changes:**

  One engineer's changes overwrite the other's
- **Inconsistent Infrastructure:**

  Actual vs. state mismatch
- **No Recovery:**

  Corrupted state may be unrecoverable

The Solution: State Locking

✅ **With State Locking Enabled**

How It Works:

- **Engineer A**

  runs `terraform apply`
- Terraform acquires a

  lock

  on the state
- **Engineer B**

  runs `terraform apply`
- Terraform attempts to acquire lock →

  FAILS
- Engineer B sees error: "State locked by Engineer A"
- Engineer A completes → releases lock
- Engineer B can now proceed safely

Implementation

```
# What happens internally (simplified)
Engineer A reads state → Plans changes
Engineer B reads state → Plans changes (same version!)
Engineer A writes state
Engineer B writes state ← OVERWRITES A's changes!
# Result: State file is now inconsistent
```

Why It's Dangerous

Race Condition

Both engineers see the same initial state, make different changes, and write back - last write wins, first write is lost forever

```
# backend.tf - S3 + DynamoDB for state locking
terraform {
  backend "s3" {
    bucket = "my-terraform-state"
    key = "prod/terraform.tfstate"
    region = "us-east-1"

    # State locking with DynamoDB
    dynamodb_table = "terraform-state-lock"
    encrypt = true
  }
}
# DynamoDB table structure:
# Primary Key: LockID (string)
# Contains: Who locked, when, from which host
```

Best Practices

Practice	Why
Always use remote backend	Enables locking + collaboration
Enable encryption at rest	Protect sensitive state data
Use separate states per env	Isolate prod from dev changes
Enable versioning on S3	Recover from corrupted states

⚠️ Important Note

Even with locking, always communicate with your team before applying changes to production. State locking prevents technical conflicts, not poor coordination!

Scenario 2: Pods Running, Users Suffering 🐛

Why Kubernetes Shows "Running" But Application Fails

The Deceptive "Running" Status

🚨 Real-World Scenario

`kubectl get pods` shows all pods as "Running". Yet users report 502 errors, slow responses, or complete failures. Your monitoring dashboard shows green. What's going on?

⚠️ The Truth About Pod Status

"Running"

only means:

- ✅ Container process is alive (PID exists)
- ✅ Container hasn't crashed
- ❌ Does NOT mean application is healthy
- ❌ Does NOT mean it's ready to serve traffic
- ❌ Does NOT check actual functionality

Root Causes

1. Failed Readiness Probes

Pod is Running but **not Ready**. Service doesn't route traffic to it, but kubectl shows Running

Debugging Checklist

```
# 1. Check ACTUAL pod readiness
kubectl get pods -o wide
# Look for READY column: 1/1 vs 0/1
# 2. Check pod events
kubectl describe pod <pod-name>
# Look for Warning events, failed probes
# 3. Check container logs
kubectl logs <pod-name> --tail=100
# Look for connection errors, timeouts
# 4. Check readiness probe status
kubectl get pod <pod-name> -o json | \
  jq '.status.conditions[] | select(.type=="Ready")'
# 5. Test pod connectivity
kubectl exec -it <pod-name> -- /bin/sh
# Try: curl database-service, nslookup, telnet
```

The Solution: Proper Health Checks

2. Network Issues

Pod is alive but can't reach dependencies (DNS, database, external APIs). Container runs but fails every request

3. Downstream Dependency Latency

Pod is healthy but waiting on slow database queries or external API calls. Appears running but timing out

4. Partial AZ Failure

Some pods in one availability zone can't reach resources, while others work fine. Intermittent failures

Implement Both Probe Types

```yaml
spec:
  containers:
  - name: app
    livenessProbe:
      httpGet:
        path: /health
        port: 8080
      initialDelaySeconds: 30
      periodSeconds: 10

    readinessProbe:
      httpGet:
        path: /ready # Check DB, dependencies
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 5
```

Probe Comparison

Aspect	Liveness	Readiness
Purpose	Is container alive?	Can it serve traffic?
On Failure	Restart container	Remove from Service
Check	Basic /health	Deep /ready + deps
Timeout	Can be longer	Should be quick

Scenario 3: Intermittent Pipeline Failures 🔄

What Causes CI/CD Pipelines to Fail Intermittently?

The Problem

🚨 **Real-World Scenario**

Your pipeline passes on retry. Tests fail randomly. Same code, different results. "Just run it again" becomes the team motto. This is NOT normal.

❌ **Potential Consequences**

Why It's Bad:

- **Wasted Time:**

  Repeated runs delay deployments
- **False Positives:**

  Bugs slip to production
- **Team Frustration:**

  Erodes trust in CI/CD
- **Delayed Feedback:**

  Slows development velocity
- **Hidden Issues:**

  Masks real problems

```
# Typical log snippet
[ERROR] Test failed: Timeout waiting for DB connection
# Rerun:
[SUCCESS] All tests passed
# What changed? NOTHING!
```

Root Causes

1. Flaky Tests

Race conditions, time-dependent code, or unmocked external dependencies cause random failures

2. Dependency Version Drift

Version ranges (^1.2.3) lead to different versions on different runs. No lockfile committed

3. Shared Runner Resource Exhaustion

Other jobs consume CPU/memory on shared runners, causing timeouts/OOM

4. Race Conditions in Parallel Jobs

Shared test DB or file system conflicts between concurrent pipeline jobs

The Solution: Deterministic Pipelines

✅ **Stable Pipelines Require Determinism**

Fix Strategies:

- **Mock Dependencies:**

  Use test doubles for external services

- **Fix Versions:**

  Use exact dependencies + lockfiles

- **Isolated Runners:**

  Use dedicated runners or containers

- **Test Isolation:**

  Unique test DBs per job

- **Flake Detection:**

  Tools like pytest-rerunfailures

Implementation

```
# GitHub Actions example: Lock versions
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Install dependencies
        run: npm ci  # Uses package-lock.json
      - name: Run tests
        run: npm test -- --retry 0  # No retries
# For flaky tests:
npm install -D jest-circus
# jest.config.js
testRunner: 'jest-circus/runner',
testTimeout: 10000
```

⚠️ **Important Note**

Intermittent failures are symptoms, not the problem. Investigate until root cause is found. "It works now" is not a fix!

Scenario 4: Unnecessary Redeployments 😩

How to Prevent Unnecessary Redeployments in CI/CD?

The Problem

🚨 **Real-World Scenario**

A small frontend change triggers full backend redeployment. Pipelines run for hours on unrelated changes. Resources wasted, deployments delayed. Why redeploy everything?

❌ **Common Issues**

Consequences:

- **Slow Pipelines:**

  Long waits for simple changes
- **Wasted Resources:**

  Unnecessary builds/tests
- **Production Risk:**

  Unchanged code redeployed
- **Team Bottlenecks:**

  Queued jobs delay merges
- **Cost Increase:**

  Higher cloud runner bills

Root Causes

1. Monolithic Pipelines

All services built/deployed together regardless of changes

2. No Change Detection

Pipelines run on every push without checking modified files

3. Coupled Stages

Build, test, deploy tightly coupled without conditions

4. Poor Monorepo Structure

No proper path-based triggers in monorepos

The Solution: Smart Pipelines

```yaml
# Typical GitHub Actions - everything runs always
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Build everything
        run: npm build  # Builds ALL microservices
```

✅ **Use Proper Change Detection**

Techniques:

- **Path Triggers:**

  Run jobs only if paths changed
- **Separate Stages:**

  Build/test/deploy independently
- **Artifact Caching:**

  Reuse unchanged builds
- **Monorepo Tools:**

  Nx/Bazel for change detection
- **Conditional Steps:**

  If file changed, then run

Implementation

```yaml
# GitHub Actions with path triggers
on:
  push:
    branches: [main]
    paths:
      - 'backend/**'  # Only if backend changes
jobs:
  backend:
    runs-on: ubuntu-latest
    steps:
      - name: Build backend
        run: cd backend && npm build
      - name: Test
        run: npm test
      - name: Deploy
        run: aws deploy --only-changed
# For monorepos with Nx
npx nx affected:build  # Only build changed projects
npx nx affected:deploy
```

⚠️ **Important Note**

Scenario 5: Infrastructure Drift 😱

Why is Infrastructure Drift Dangerous?

The Problem

🚨 **Real-World Scenario**

An engineer makes a quick "temporary" change in AWS console. Weeks later, terraform apply tries to "fix" it, deleting production resources. Chaos ensues.

❌ **Consequences of Drift**

Risks:

- **Unpredictable Applies:**

  Terraform reverts manual changes
- **Data Loss:**

  Could delete modified resources
- **Outages:**

  Unexpected destructions during apply
- **Compliance Violations:**

  Undocumented changes
- **Team Conflicts:**

  "Who changed this manually?!"

```
# terraform apply output
aws_instance.prod_server: Refreshing state... [id=i-0abc123]
# Drift detected!
aws_instance.prod_server: Modifying... [id=i-0abc123]
  machine_type: "t2.micro" → "t3.micro" # Reverts manual change
```

Root Causes

1. Manual Console Changes

Emergency fixes or "quick tweaks" without updating IaC

2. External Modifications

Auto-scaling, external tools, or other teams change resources

3. Incomplete IaC

IaC doesn't cover all resource attributes

4. No Drift Detection

No regular terraform plan checks in CI/CD

The Solution: Drift Prevention

✅ **Detect and Prevent Drift**

Strategies:

- **Daily Drift Checks:**

  CI job runs terraform plan

- **IaM Policies:**

  Restrict console changes

- **Sentinel Policies:**

  Enforce via Open Policy Agent

- **Import Resources:**

  Bring manual changes into IaC

- **Read-Only Mode:**

  For sensitive resources

Implementation

```yaml
# GitHub Actions drift check
name: Drift Detection
on:
  schedule:
    - cron: '0 0 * * *'  # Daily
jobs:
  drift:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Terraform Plan
        run: terraform plan -detailed-exitcode
      - name: Alert if Drift
        if: ${{ steps.plan.exitcode == 2 }}
        run: echo "Drift detected!" | slack-notify
```

⚠️ **Important Note**

Drift is inevitable in large teams. Focus on detection and correction, not blame. Make IaC easy to update after emergencies.

Scenario 6: Secret Sharing Nightmare 🔒

How Do You Safely Share Secrets Across Environments?

The Problem

🚨 **Real-World Scenario**

API keys committed to repo. Env vars copied via email. Prod secrets in dev environments. One leak, and your entire infrastructure is compromised.

❌ **Common Mistakes**

Risks:

- **Repo Exposure:**

  Secrets in code or .env files
- **Manual Copy:**

  Emails, Slack - easy leaks
- **Env Mixing:**

  Prod secrets in dev/test
- **No Rotation:**

  Compromised keys live forever
- **Over-Privileged:**

  Secrets with too much access

```
# ❌ BAD: Hardcoded secret
provider "aws" {
  access_key = "AKIAIOSFODNN7EXAMPLE"
  secret_key = "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
}
# Git commit → Exposed forever in history
```

Root Causes

1. Convenience Over Security

Hardcoding "just for testing" and forgetting to remove

2. Lack of Tools

No secret manager - fallback to env vars/files

3. Poor Access Control

Everyone has access to all secrets

4. No Auditing

Can't track who accessed what secret when

The Solution: Secret Managers

✅ **Use Dedicated Secret Managers**

Tools:

- **AWS Secrets Manager:**

  Rotate + audit
- **HashiCorp Vault:**

  Dynamic secrets
- **Azure Key Vault:**

  RBAC integration
- **GCP Secret Manager:**

  Versioning
- **GitHub Secrets:**

  For CI/CD

Implementation

```
# Terraform with AWS Secrets Manager
data "aws_secretsmanager_secret_version" "db" {
  secret_id = "prod/db-credentials"
}
# Access in resource
resource "aws_db_instance" "prod" {
  username =
jsondecode(data.aws_secretsmanager_secret_version.db.secret_string)
["username"]
  password =
jsondecode(data.aws_secretsmanager_secret_version.db.secret_string)
["password"]
}
# In CI/CD (GitHub Actions)
env:
  AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
```

⚠️ **Important Note**

Implement least privilege: Secrets accessible only to services that need them. Rotate regularly and monitor access logs.

Scenario 8: Mystery Latency ⏱️

How to Debug High Latency When CPU and Memory Look Normal?

The Problem

🚨 **Real-World Scenario**

Users complain of slow app response. Prometheus shows low CPU/memory. No errors in logs. Requests take 5s instead of 200ms. Where's the bottleneck?

❌ **Common Misconceptions**

Average Metrics Hide Truth:

- **Averages Lie:**

  p50 ok, but p99 terrible
- **Infra OK ≠ App OK:**

  Problem in code/dependencies
- **No Errors:**

  Timeouts/retries are silent killers
- **Local Fine:**

  Prod has real traffic/load

```
# Prometheus query shows "normal"
http_request_duration_seconds{quantile="0.5"} = 0.2s
# But:
http_request_duration_seconds{quantile="0.99"} = 4.8s
# 1% of requests are SLOW!
```

Root Causes

1. Tail Latency

p99/p95 high due to rare slow requests amplifying

2. Downstream Dependencies

Slow DB queries, API calls, or queue backlogs

3. Network Issues

Retries, packet loss, DNS resolution delays

4. Code Inefficiencies

N+1 queries, unoptimized loops, blocking I/O

The Solution: Deep Observability

✅ **Debug High Latency**

Tools & Techniques:

- **Distributed Tracing:**

  Jaeger/Ziplock - see span times
- **Percentile Metrics:**

  Monitor p90/p99, not averages
- **Profiling:**

  Flame graphs for code hotspots
- **Dependency Monitoring:**

  Track external call latencies
- **Query Optimization:**

  EXPLAIN ANALYZE slow queries

Implementation

```
# Jaeger tracing example (Node.js)
const { initTracer } = require('jaeger-client');
const tracer = initTracer({ serviceName: 'app' });

app.get('/api', (req, res) => {
  const span = tracer.startSpan('api-call');
  // Do work
  db.query().then(() => {
    span.finish();
  });
});
```

⚠️ **Important Note**

Average latency is useless. Always look at percentiles. 99% good + 1% terrible = unhappy users!

Scenario 9: Scaling Dilemma 📈

When Should You Scale Vertically vs Horizontally?

The Problem

🚨 **Real-World Scenario**

App hits performance limits. Do you bump up instance size (vertical) or add more instances (horizontal)? Wrong choice leads to outages or high costs.

❌ **Vertical Scaling Pitfalls**

Issues:

- **Single Point Failure:**

  One big instance down = total outage
- **Hard Limits:**

  Can't scale beyond max instance size
- **Downtime:**

  Often requires restart/resize
- **Cost Inefficiency:**

  Overprovision for peaks

```
# Vertical scale AWS EC2
aws ec2 modify-instance-attribute \
  --instance-id i-1234567890 \
  --instance-type {"Value": "m5.2xlarge"}
# Requires stop/start - DOWNTIME!
```

Root Causes for Wrong Choice

1. Stateful Apps

Horizontal scaling hard for stateful services (DBs, sessions)

2. Cost Miscalculation

Vertical cheaper short-term, horizontal better long-term

3. Performance Myths

"Bigger instance always faster" - ignores network/DB limits

4. No Auto-Scaling

Manual scaling misses peaks/valleys

The Solution: Right Scaling Strategy

✅ Vertical vs Horizontal

Aspect	Vertical	Horizontal
When	Stateful apps, quick fix	Stateless, high availability
Pros	Simple, no distribution	Resilient, auto-scaling
Cons	Downtime, limits	Complexity, state management

Implementation

```yaml
# Horizontal with Kubernetes HPA
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: app-hpa
spec:
  scaleTargetRef:
    kind: Deployment
    name: app-deployment
  minReplicas: 3
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

⚠️ Important Note

Horizontal scaling improves availability and resilience. Vertical scaling is simpler but has limits and restart impact. The choice depends on statefulness and failure tolerance.

Scenario 10: Outage Panic 🚨

What's the Biggest Mistake Engineers Make During Outages?

The Problem

🚨 **Real-World Scenario**

Production down. Team in war room. Someone restarts services without checking impact. Outage worsens. "Fixes" create new problems.

❌ **Biggest Mistakes**

Common Errors:

- **Blind Restarts:**
  Without understanding blast radius
- **Symptom Fixing:**
  Treating symptoms, not root cause
- **No Rollback:**
  Changes without undo plan
- **Poor Communication:**
  No incident commander
- **Overloading:**
  Too many "fixes" at once

```
# ❌ During outage:
kubectl restart deployment/app
# Worsens: Loses in-flight requests, no root cause fixed
# Better:
kubectl logs -f pod/app-abc  # Investigate first
```

Root Causes of Bad Decisions

1. Pressure

Panic leads to hasty actions without thinking

2. Lack of Process

No incident response playbook

3. Poor Visibility

Inadequate monitoring/logs

4. Hero Culture

"I know the fix" without verification

The Solution: Structured Response

✅ **Incident Response Best Practices**

Steps:

- **Assign Commander:**

  One leader coordinates
- **Contain First:**

  Isolate impact
- **Investigate:**

  Logs, metrics, traces
- **Hypothesis Test:**

  Small changes with rollback
- **Communicate:**

  Status updates to stakeholders

Playbook Template

```
# Incident Playbook
1. Acknowledge: "Outage detected at [time]"
2. Assess Impact: Users affected? Scope?
3. Gather Data: Logs, metrics, recent changes
4. Hypothesis: "DB overload causing latency?"
5. Test Fix: Scale DB, monitor
6. Verify: Issue resolved?
7. Post-Mortem: Root cause, prevention
```

⚠️ **Important Note**

During outages, slow is fast. Methodical fixes prevent escalation. Practice fire drills to build muscle memory.