# Cloud/DevOps Assessment Responses

*Submitted by: Abu Bakkar*

*Date: 3/2025*

## Introduction

Thank you for the opportunity to participate in this technical assessment. I've approached each question and task drawing from my experience as a Cloud Engineer. I've focused on providing practical solutions based on real-world implementations I've worked with as far.

The following responses outline my approach to DevOps challenges, infrastructure automation, and cloud security considerations. I've endeavoured to balance technical accuracy with readability throughout.

## *Section 1: Theory Questions*

### Answer1-DevOps/DevSecOps tools I’ve worked with:

I’ve cut my teeth on **AWS** services, **Docker**, Terraform, Ansible, GitHub Actions and Jenkins. These have been my bread and butter for automating both infrastructure and application deployments.

| Category | Used Tools | How to utilize it |
|---|---|---|
| **Infrastructure as Code (IaC)** | Terraform, Ansible, AWS CloudFormation | Automated AWS infrastructure provisioning (VPC, EC2, RDS). Used Terraform with a remote backend (like S3) to prevent state conflicts. |
| **CI/CD Pipelines** | Jenkins, GitHub Actions | Built CI/CD pipelines to automate code deployment, integrated security scanning tools in pipelines |
| **Containerization & Orchestration** | Docker, Kubernetes (Basic) | Containerized applications using Docker, have a solid theoretical understanding of Kubernetes concepts like pods, deployments, and services. |
| **Monitoring & Logging** | Grafana, AWS CloudWatch | I've got monitoring and alarms set up. And when something goes a bit pear-shaped, i’m using CloudWatch Logs to figure out what's gone wrong?!. |
| **Security & Compliance** | AWS IAM, Security Groups, AWS WAF, HashiCorp Vault | So for security, we made sure everyone only had the absolute minimum access they needed, used AWS WAF to block common online nasties, and kept all the sensitive passwords and keys |

| | | |
|---|---|---|
| | | safe with Vault. |

## Technical challenge example:

We had this **hospital finance system**, deployed lovely in testing, but come month-end in production, it’d just go completely wrong. Looking at the AWS logs and Docker stuff, the containers were running out of memory when the finance team did their big reports. Sorted it by setting proper limits on the Docker containers, got the developers to make those big database queries a bit less greedy, and put some alarms in CloudWatch to warn us if things were about to go south. Now the finance lot have no more deployment problems, and those important monthly reports run perfectly.

## CI pipeline security scan troubleshooting:

When security scans pass on my machine but fail in CI, here’s how I’d tackle it:

1. Compare the environments – often it’s something daft like different versions or configurations
2. Check if the CI environment has more up-to-date vulnerability databases
3. Have a proper look at the CI logs to spot exactly which security rules are being triggered

4. Mock up the CI environment locally to recreate the issue
5. Crank up the verbosity in the security scanner logs
6. Look at recent pull requests for new packages or dependencies that might be causing trouble

## Answer 2: CI/CD Pipeline Setup

**Setting up a GitHub Actions pipeline – the simple version:**

1. **Repository Setup**

- Ensuring the code repository is properly structured
- Create a `Jenkinsfile` in the root of the repository

2. **Basic Jenkins Pipeline:** The comprehensive `Jenkinsfile` for a CI/CD pipeline with parallel test execution and credential management clipped onto the **Github**.
3. **Credential Management in Jenkins**

- Store sensitive information in Jenkins Credentials Manager
- Access credentials securely using the `withCredentials` block
- Use different credential types (username/password, AWS credentials, tokens)

4. **Parallel Test Execution**

- Use the `parallel` block to run testing stages simultaneously
- Separate unit tests, integration tests, and security scans
- This reduces overall build time significantly

## Answer 3: Security Considerations

**Keeping the cloud setup secure – the practical approach:**

**Access Control:**

1. Use of AWS IAM and stick to the principle of least privilege – only give what's needed
2. Prefer temporary access with IAM roles over permanent access keys
3. Regularly check who's got access to what and clean up old accounts
4. Turn on Multi-Factor Authentication – it's a no-brainer

**Data Protection:**

1. Encrypt the data using AWS KMS – both at rest and in transit
2. Lock down S3 buckets – public buckets are asking for trouble
3. Keep an audit trail with CloudTrail – invaluable when things go wrong
4. Use VPC endpoints to keep traffic off the public internet

**Network Security:**

1. Configure security groups like a proper bouncer – deny by default, allow by exception

2. Keep sensitive resources in private subnets
3. Set up AWS WAF to protect web applications from common attacks
4. Enable flow logs so you can see what’s happening on your network

### Answer 4: Staying Updated

How I keep up with the ever-changing cloud landscape:

1. I make the cloud blog part of my regular reading – usually with my morning coffee/ or in bus
2. Pop along to local AWS and DevOps meetups when I can – great for networking too
3. I'm a big fan of practical learning,
4. Subscribe to Last Week in AWS – Corey Quinn cuts through the marketing fluff
5. Set aside a few hours each sprint to play with new services – it's the best way to learn
6. Follow folks core tech on socials

# 1.1 Infrastructure Setup with Terraform

I've created Terraform code to provision the required AWS infrastructure. Below is my explanation of the approach:

**Infrastructure Components:**

1. **VPC**: Created with CIDR 10.0.0.0/16
2. **Subnets**: One public subnet (10.0.1.0/24) and one private subnet (10.0.2.0/24)
3. **Internet Gateway**: Attached to the VPC for public internet access
4. **Route Tables**: Configured to direct traffic from the public subnet to the internet
5. **Security Groups**:
    a. EC2 security group allowing SSH access
    b. RDS security group allowing MySQL access only from the EC2 instance
6. **EC2 Instance**: t2.micro in the public subnet
7. **RDS Instance**: MySQL 5.7 instance in the private subnet

## Handling State File Conflicts

To handle state file conflicts when multiple engineers run `terraform apply` simultaneously, I would implement the following strategy:

1. **Remote State Storage**:
    ○ Store the Terraform state file in AWS S3 with versioning enabled
    ○ Use DynamoDB for state locking

    **Implementation**:

    The proper Terraform code, that's all there in the **GitHub** file.

2. **Benefits**:
    ○ **State Locking**: DynamoDB provides locking to prevent concurrent operations
    ○ **Versioning**: S3 versioning allows rollback if needed

    - **Team Collaboration**: Remote state enables team members to work from consistent state data

3. **Technical Procedure:**
    - Always run `terraform init` before any other commands
    - Use `terraform plan` to preview changes before applying

# 1.2 Docker & Kubernetes

## Dockerfile for Node.js Application

I've created a Dockerfile that:

1. Uses the lightweight *Alpine-based Node.js* image.
2. Properly separates dependency installation from code copying for better layer caching.

## Kubernetes Deployment

The Kubernetes deployment manifest:

1. Creates a deployment with 3 replicas for high availability.
2. Sets resource requests and limits to prevent resource exhaustion.
3. Includes health checks via liveness and readiness probes.
4. Creates a Load-Balancer service to expose the app.

## OOMKilled Error Troubleshooting

Well, I haven’t directly encountered a pod being OOMKilled in a production Kubernetes cluster just yet. However, having done a bit of digging and looking into common resources , I understand there’s a fairly standard approach to diagnosing these sorts of incidents and preparing a RCA.

**Troubleshooting OOMKilled Error**:

1. Check **Pod Logs** (`kubectl logs <pod-name>`).
2. Check **Memory Limits** in `deployment.yaml`.
3. **Resolution steps**:-
    a. Adjust memory limits in the deployment manifest
    b. Optimize the application code for better memory management
4. **Root Cause Analysis documentation**:
    a. Record the memory usage pattern
    b. Document the specific operation that triggered high memory usage
    c. Implement memory profiling in the app.

# Real-World Scenarios

## 2.1 Scaling and Performance

**Scenario: Your web application hosted on AWS is experiencing high traffic, leading to increased response times and occasional failures.**

To address the scaling and performance issues for our web application, I would implement the following approach:

**Initial Assessment**

**Pinpoint Solutions for High Web Traffic on AWS:**

- ❖ **Identify the Bottleneck:** Use CloudWatch to figure out *what's* struggling – is it the servers (CPU, memory), the database, or the network? Check the application logs for errors too. Get more detailed monitoring if needed.

**Scaling Strategy**

**Compute Layer:**

1. Implement **Auto Scaling Groups** to automatically adjust EC2 capacity based on traffic patterns
2. Move from individual EC2 instances to an **ECS** cluster for containerized applications
3. Configure appropriate scaling policies based on CPU utilization, network traffic, and request count metrics
4. Set minimum, maximum, and desired capacity with appropriate scaling thresholds

**Load Balancing:**

1. Deploy **Application Load Balancer** to distribute traffic across multiple instances
2. Enable sticky sessions if the application requires session persistence
3. Configure health checks to route traffic only to healthy instances
4. Implement path-based routing for microservices architecture

**Database Layer:**

1. For databases offload those reads to **RDS Read Replicas**
2. Consider **Amazon Aurora** for better speed and scaling.
3. Implement **connection pooling** to efficiently manage database connections
4. For NoSQL workloads, use **DynamoDB** with appropriate provisioned capacity

**Caching:**

- Implement **ElastiCache (Redis)** to cache frequent database queries
- Use **CloudFront** as a CDN to cache static assets closer to users
- Set up **API Gateway** caching for frequently accessed API endpoints

**High Availability Implementation**

- Deploy across **multiple Availability Zones (AZs)** with **Route 53 for DNS failover**.
- Store static content in **S3** for high availability.
- Configure **Multi-AZ replication** for RDS.

**Monitoring and Alerting**

- Set up **CloudWatch Alarms** for early warning of performance issues
- Implement **X-Ray** for distributed tracing to identify bottlenecks
- Create a **CloudWatch Dashboard** for real-time monitoring of key metrics
- Configure **SNS** notifications for critical alerts

## 2.2 Cost Management

**Scenario: As a DevOps engineer, you are tasked with reducing the monthly cloud expenditure of your company by 20% without compromising performance.**

To reduce our AWS costs by 20% while maintaining performance, I would implement the following practical strategies:

★ **Assessment and Visibility**

1. **Cost Analysis:**
    - Use **AWS Cost Explorer** to identify highest expense services and resources
    - Review **AWS Cost and Usage Reports** for detailed spending patterns
    - Set up **AWS Budgets** to track spending against targets
2. **Resource Utilization:**
    - Analyze **CloudWatch metrics** to identify underutilized resources
    - Review **Trusted Advisor** recommendations for cost optimization
    - Generate **resource utilization reports** to identify waste

- **Optimization Strategies**

1. **Right-sizing Resources:**
    - **Downsize over-provisioned EC2 instances** based on actual CPU and memory usage
    - Convert to **T3/T4g instances** with burstable performance for variable workloads
    - Use **EC2 Auto Scaling** to match capacity with demand patterns
2. **Storage Optimization:**
    - Implement **S3 Lifecycle Policies** to move infrequently accessed data to cheaper storage tiers
    - Delete unused **EBS snapshots** and orphaned volumes
    - Convert **gp2 volumes** to **gp3** for better performance at lower cost
3. **Reserved Instances & Savings Plans:**
    - Purchase **Reserved Instances** for predictable workloads
    - Implement **Savings Plans** for EC2 and Fargate usage
    - Consider **Spot Instances** for non-critical, fault-tolerant workloads
4. **Architectural Improvements:**
    - Implement **serverless architecture** using Lambda for appropriate workloads
    - Use **Application Load Balancer** instead of multiple Classic Load Balancers
    - Migrate to **managed services** that reduce operational overhead
5. **Database Optimization:**
    - Convert underutilized **RDS instances** to smaller sizes
    - Use **Aurora Serverless** for variable workloads
    - Implement **Multi-AZ** only for critical production databases

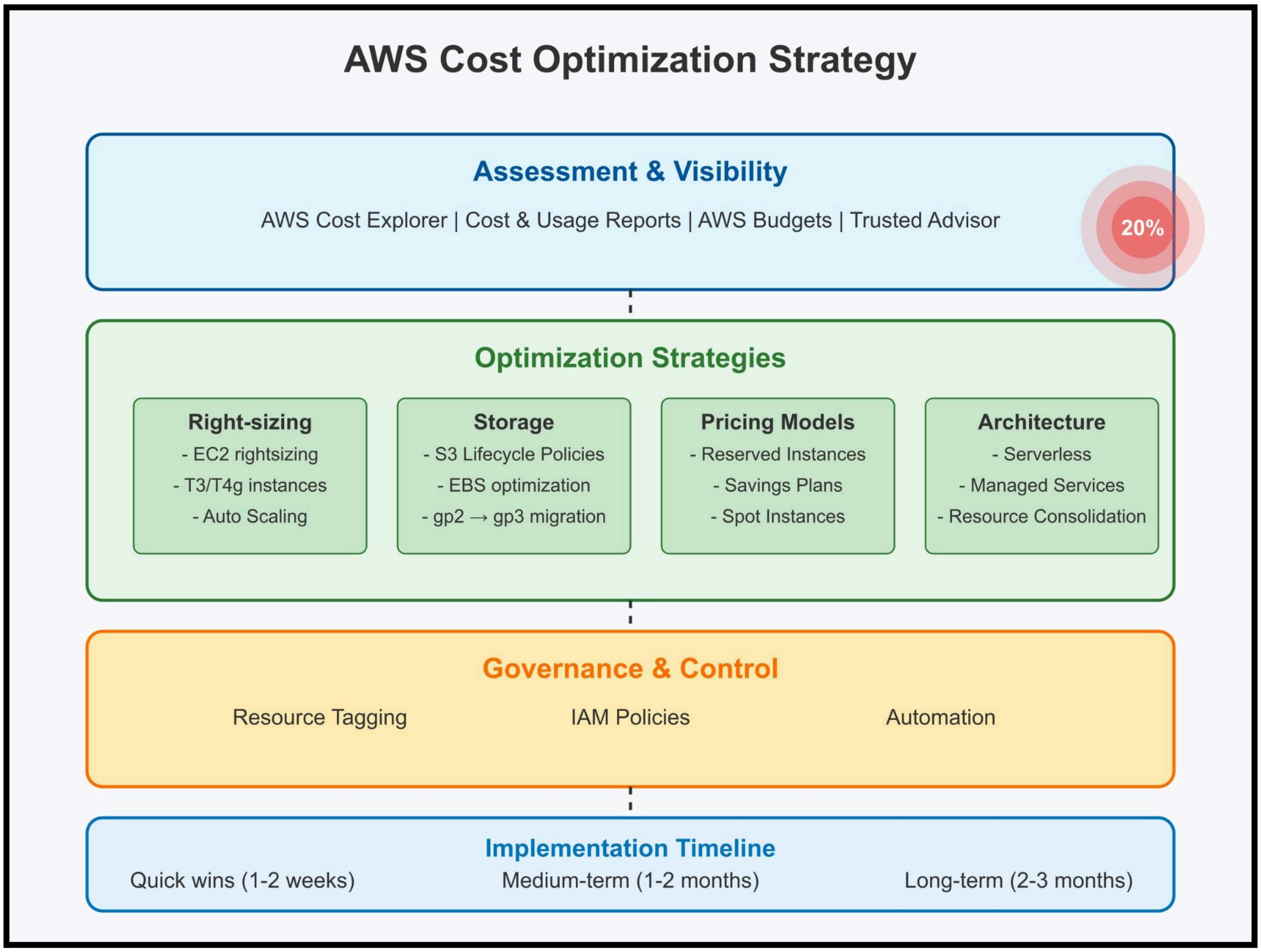
### Governance and Control

1. **Implement tagging strategy:**
    - Enforce **resource tagging** for cost allocation
    - Create **tag-based access policies** to control resource creation
2. **Set up cost controls:**
    - Configure **IAM policies** to restrict expensive resource types
    - Implement **Service Control Policies** at the organization level
3. **Automation:**
    - Schedule **automatic start/stop** of development environments
    - Implement **automatic cleanup** of unused resources
    - Create **Lambda functions** to identify and remediate cost anomalies

### Implementation Plan

1. **Quick wins** (1-2 weeks):
    - Clean up unused resources

    - Implement basic tagging
2. **Medium-term** (1-2 months):
    - Purchase Reserved Instances/Savings Plans
    - Optimize database resources
3. **Long-term** (2-3 months):
    - Refactor for serverless where appropriate
    - Implement comprehensive governance
    - Establish cost-awareness culture



*The diagram above illustrates my comprehensive approach to AWS cost optimization, showing the key components and their relationships.*