YouTube Video URL:

https://youtu.be/nJrCylZNUJg

Q1. Describe how you manage infrastructure changes using tools like Terraform. What challenges have you encountered?

In my role as a **Site Reliability Engineer**, I've managed infrastructure across multiple AWS accounts, like dev, test, and prod. The applications we run are **containerized** and deployed into **AWS EKS clusters (Elastic Kubernetes Service)** with **managed node groups**, meaning AWS handles the heavy lifting of managing the servers.

1. Scripting Infrastructure with Terraform IaC: I write Terraform code (in HCL) to define everything:
VPCs, subnets, EKS clusters, IAM roles, security groups, etc.

2. Automation with GitLab CI: I integrated this with **GitLab CI pipelines**, so anytime a change is pushed, the pipeline automatically:

- Runs terraform plan to show what's going to change.
- Runs terraform apply after approvals to make those changes.

3. Separate AWS accounts for Dev, Test, and Prod: We deploy to different AWS accounts for **dev**, **test**, and **prod**. The same **Terraform** code works for all environments using **variables and workspaces.**

4. EKS Cluster Setup: Each environment gets its own **EKS clusters**. Terraform provisions the cluster, sets up IAM roles for Kubernetes access, and configures node groups so our applications can run smoothly.

Challenges faced and how I handled them:

1. **Deploying to multiple AWS Accounts:** Dealing with multiple AWS accounts meant I had to manage **cross-account roles and access**. I solved it by automating role assumption using Terraform's assume_role block and made account-specific configurations through GitLab CI environment variables.

2. Securely managing secrets: I used **AWS Secrets Manager** to inject secrets securely into the pipeline.

3. Verification of production changes: We used a mandatory policy to deploy each change to dev and test environments first before moving to prod. Also, we are using manual approvals to verify and approve any production change.

4. Terraform state file management: State file conflicts can happen when multiple pipelines try to modify infra at once. I stored Terraform state in **S3** with **DynamoDB for locking**, ensuring only one operation runs at a time.

Q2. What **disaster recovery strategies** would you implement in a cloud environment to ensure high availability for **critical services**?

1. Always deploy your critical services in **multiple Availability Zones**.

2. In AWS, use **Elastic Load Balancer (ELB)** across **multiple AZs**.

3. Geo-Redundancy with Multi-Region setups:

- Replicate your infrastructure and databases in **another region**.
- Use **DNS failover** (like Route 53 in AWS) to switch traffic automatically to the backup region.

4. Automated Backups & Snapshots:
Schedule daily automated backups for all databases and key storage (EBS Volumes).
Store backups in **a separate region or account** for safety.

5. Script your Infrastructure with **IaC tool like Terraform**.
Maintain **git-enabled repos** to keep **different versions** of your IaC code.
if required, have **DR scripts ready to recreate infrastructure in another region**.

6. **Simulate DR activities** at least once in every quarter.

Q3. If you notice unusual traffic patterns that could indicate a security breach, what steps would you take to investigate and mitigate the threat?

When something like this happens, **you would observe the following stuff**:

- Number **of requests hitting an application increased rapidly**
- Traffic appearing from **unusual IPs**
- **Unusual logs** which are not seen often

To identify source of this traffic:

- I will try to **validate whether the traffic is legitimate** or not by contacting business/customer
- **Use log search query tools** like Grafana Loki to review logs
- Will try to find high number **of failed login attempts**
- Will look for high number **of 500 or 403 errors**
- Also in AWS, I can check **CloudTrail trails and VPC Flow Logs** or any **firewall/WAF logs**

Solution:

- **Block suspicious IPs or IP ranges** using **firewalls** or **NACLs or allow specific IP/IP range in AWS Security Groups**
- Use **WAF rules** to **filter malicious requests**
- **Scale down** impacted services if **needed to protect data and systems**
- **Review overall security posture to avoid such incidents in future**

Q4. Describe the **blue**-green deployment strategy. What **benefits** and **potential challenges** do you see when implementing it in a cloud environment?

You have **two identical environments**:
- **Blue**: The one currently live and serving users.
- **Green**: The new version, tested and ready to be deployed to production.

Instead of replacing the code on the live server (which can cause downtime or errors), you:
- Deploy the new version (**Green**) **side by side** with the current one (**Blue**).
- Run final tests on the Green version, while users are still on Blue.
- Once you're 100% confident the **Green** version works perfectly, you redirect the traffic from **Blue** to **Green**, usually with a **load balancer** or **DNS update**.

Now, **Green is live**, and Blue is kept idle as a backup in case something goes wrong.

Benefits:
- **Zero Downtime Deployment:** The users do not experience any interruption. They move from the old to the new version instantly.
- **Quick Rollback**: If something goes wrong after going live, you can easily switch back to the old version (**Blue**) quickly.
- **Better production testing:** You can run tests on the Green version in a live environment without disturbing actual users.
- **More confident strategy:** Developers and product teams feel safer releasing features because there's always a way to undo changes.
- **Cloud deployments make it easier:** With cloud services (like AWS, Azure, GCP), you can spin up duplicate environments quickly, scale automatically, and switch traffic with load balancers or DNS changes.

Challenges:

1. **Double Infrastructure cost:** You're running **two environments at the same time**, even if temporarily which can increase cloud costs.

2. Stateful Systems are Harder: If your app uses a database, you have to make sure both Blue and Green versions work with the same schema or can handle migrations smoothly.
Otherwise, you risk data inconsistencies when switching over.

3. Traffic Switching Isn't Always Instant: DNS propagation can take time, so some users may still hit the old version briefly after the switch, unless managed properly.

4. Operational Complexity: Managing two full environments means more deployment scripts, more monitoring, more testing — so your DevOps pipeline has to be mature.

Q5. What **steps** would you take to **optimize** the **performance** of a cloud-based application that is experiencing **latency issues**?

Tools to use to investigate: Cloud Monitoring (like AWS CloudWatch), APM tools (Datadog, New Relic, Prometheus + Grafana) or a tool like Wireshark.

1. Metrics to look for:

- Response time (how long the app takes to reply)
- Error rates
- CPU, Memory, Disk, Network usage
- Database query times

2. Reproduce the issue in a dev or staging environment, if possible. Trace the whole network path and try to find out the hop where most time is being spent.

3. Identify which application component is causing the latency: Frontend, backend, DB, or is it from the network itself.

- **Frontend:** Is the browser loading slow? Too many images to load?
- **Backend:** Are API responses slow? Any service crashes or timeouts?
- **Database:** Are queries taking too long?
- **Network:** Any latency between services or from CDN (like CloudFront)?

4. Any resource bottlenecks: CPU/RAM crunch, network throughputs , Disk I/O, too many users are attempting something at the same time

Solution:

- Horizontal/vertical scaling
- caching solutions like Redis
- optimize slow DB queries

5. Application Architecture Improvement:

• **Use CDN** (Content Delivery Network) to serve static files faster (Cloudflare, AWS CloudFront).
• **Use Load Balancers** to distribute traffic evenly.
• **Introduce Caching layers** at API level (e.g., HTTP cache) and DB level (e.g., Redis/Memcached).