Section 1: General Technical Knowledge

1. **What are the key security concerns when it comes to DevOps?**

ANS:

One of the major concerns is insecure code and vulnerabilities, which can lead to attacks such as SQL injection and cross-site scripting (XSS).

Another critical issue is secrets management. If secrets are not properly managed or stored, they can be exposed, allowing attackers to gain access to sensitive information and cause significant damage.

Container security is also a concern, especially when containers are running with root privileges. If containers are not properly secured, attackers can exploit this to gain root access to the application and the underlying system.

Lastly, the lack of monitoring and logging poses a security risk. Without adequate monitoring and logging, suspicious activities or potential breaches may go unnoticed, delaying response and remediation.

2. **How do you design a self-healing distributed service?**

ANS:

To design a self-healing distributed service, I will use Kubernetes and Argo CD. My Kubernetes cluster will be configured with both node auto-scaling and pod auto-scaling, allowing the system to scale based on demand and resource utilization. This ensures that failed instances are automatically replaced with new ones, maintaining availability.

For the deployment strategy, I will implement rolling updates to ensure smooth transitions when deploying new versions, hence minimizing downtime.

Regarding data replication and backup, I will use regular data replication strategies for critical data. For instance, I will enable database replication to ensure data availability even if a primary node fails, along with automated backups for easy recovery if needed.

Argo CD will facilitate automated rollbacks of my application by automatically reverting to the previous working version in case a deployment causes issues, ensuring minimal disruption to the service.

3. **Describe a centralized logging solution and how you can implement logging for a microservice architecture**

ANS:

A centralized logging solution is an approach where logs from all microservices in an architecture are collected and stored in a single, unified location. This enables efficient log aggregation, searching, analysis, and monitoring, helping teams troubleshoot and gain insights into the behavior and performance of microservices in a distributed system.

I will implement an ELK/EFK stack for centralized logging. First, I will set up Elasticsearch as a StatefulSet for scalable and persistent log storage.

Then, I'll install Fluent Bit as a DaemonSet to collect logs from all nodes and containers which my microservices run on. Fluent Bit will parse incoming logs, add metadata, and transform them into a standardized format before storing them in Elasticsearch.

To visualize and query the logs, I'll use Kibana. Additionally, I will create custom dashboards to monitor key metrics like error rates, response times, and overall service health.

4. **What are some of the reasons for choosing Terraform for DevOps?**

ANS:

I choose Terraform because it is cloud-agnostic, allowing me to use it across different platforms.

It's easy to use and supports source control management (SCM), enabling versioning of my infrastructure code.

Terraform also integrates seamlessly with CI/CD pipelines, providing full automation for infrastructure provisioning and management.

In addition, its modular approach ensures that my infrastructure code is scalable and reusable.

5. **How would you design and implement a secure CI/CD architecture for microservice deployment using GitOps? Take a scenario of 20 microservices developed using different languages and deploying to an orchestrated environment like Kubernetes. (You can add a low-level architectural diagram)**

ANS:

To implement a secure CI/CD architecture for deploying 20 microservices to a Kubernetes cluster using GitOps, the following approach will be used:

Note:

- **The Kubernetes cluster and ArgoCD tool have been provisioned using Terraform. The EKS cluster and ArgoCD are ready to deploy microservices.**
- **Architecture diagram is on the root folder of the repository. (filename is CICD-Architecture)**

a. Git Repository Organization

- Microservice Repositories: Each microservice will have its own dedicated GitHub repository to store the source code and related files.
- Infrastructure Repository: A centralized repository will be created to manage infrastructure and deployment configurations. This repository will include:
  - Helm charts for all microservices.
  - ArgoCD application manifests.
  - Terraform scripts for infrastructure provisioning.

b. CI Pipeline with GitHub Actions:

The CI pipeline will be configured using GitHub Actions to enable the Continuous Integration of applications. The pipeline will include the following stages:

- SonarCloud Stage:
- Ensures code quality by checking against predefined quality gates which deployment only proceeds to the next stage if quality gates are passed.
- SCA (Software Composition Analysis) Stage:
  - Scans the code for vulnerable packages and dependencies.
- Build Stage:
  - Builds the Docker image using the microservice's Dockerfile.
  - Pushes the built image to an artifact registry (AWS ECR).
- Test Stage:

        - Runs unit tests to validate the functionality of the code.
    - Deploy Stage:
        - Updates the values.yml file in the infrastructure repository's Helm chart with the new image tag generated during the build stage.

c. Continuous Deployment with GitOps:
    - Helm and ArgoCD Configuration:
        - Helm charts will be created for each microservice and stored in the infrastructure repository.
        - ArgoCD application manifests with complete configurations for the microservices will also be stored in this repository.
    - GitOps with ArgoCD:
        - ArgoCD will manage the deployment process by monitoring the infrastructure repository for changes.

    When a Helm chart is updated by the deploy stage of the CI, ArgoCD will automatically sync the changes to the Kubernetes cluster, then deployment to EKS cluster kicks off

d. Deployment Workflow
    - Code Push:
        - Developers push changes to a microservice repository.
    - CI Pipeline Execution:
        - The CI pipeline runs and completes the stages: SonarCloud analysis, SCA, build, test, and deploy.
        - The deploy stage updates the values.yml file in the Helm chart within the infrastructure repository with the new image tag.
    - GitOps Trigger:
        - ArgoCD detects the change in the Helm chart and triggers the deployment process.
        - The updated microservice is deployed to the Kubernetes cluster automatically.

This process is replicated for each of the 20 microservices. Each repository follows the same CI/CD pipeline, and all deployment configurations are centralized in the infrastructure repository.

6. **You notice React Native builds are failing intermittently. What's your debugging process?**

ANS:

To debug intermittent React Native build failures, I would start by reviewing the build logs to identify any errors or warnings that could be contributing to the issue.

Next, I would check for any dependency issues, ensuring there are no package compatibility problems.

I would also verify that all environment variables are correctly set. Additionally, I would analyze potential network issues, as some builds may involve downloading dependencies.

Finally, I would use the information from the failed build log to address any specific issues and tackle them accordingly.

Section 2: Coding Challenge

7. **Write Prometheus exporter in Python/Golang that connects to specified RabbitMQ HTTP API (it's in the management plugin) and periodically reads the following information about all queues in all vhosts:**

● messages (total count of messages)
● messages_ready
● messages_unacknowledged
It should then export 3 new metrics:
● rabbitmq_individual_queue_messages{host,vhost,name}
● rabbitmq_individual_queue_messages_ready{host,vhost,name}
● rabbitmq_individual_queue_messages_unacknowledged{host,v
host,name}
where the host is RabbitMQ hostname, vhost is RabbitMQ vhost and name is name of the queue.
It should use RABBITMQ_HOST, RABBITMQ_USER, and RABBITMQ_PASSWORD environment variables to run multiple deployments of this and just change the env in them.

ANS:

The code is on the root folder of the repository (filename is prometheus-exporter.py)

8. **Write a script to restart the Laravel backend service if CPU usage exceeds 80%.**

ANS:

The code is on the root folder of the repository (file is monitor-cpu.sh)

9. **A Postgres query is running slower than expected. Explain your approach totroubleshooting it**

ANS:

When troubleshooting a slow PostgreSQL query, I adopt a systematic approach that combines diagnostic tools with query optimization strategies:

- Enable PostgreSQL's Slow Query Log
  - I first enable the slow query log to capture queries that exceed a specified runtime threshold. This helps me identify which queries to focus on and provides a starting point for optimization.
- Analyze Query Execution Plans with EXPLAIN or auto_explain
  - For identified queries, I analyze the execution plan using EXPLAIN or the auto_explain module, which logs detailed execution plans for slow queries. This helps me understand inefficiencies like sequential scans, missing indexes, or suboptimal join strategies.
- Review System-Wide Query Performance with pg_stat_statements
  - Using pg_stat_statements, I get aggregated performance data, identifying high-frequency queries or I/O-intensive operations that may collectively contribute to the slowdown.
- Optimize Query Structure
  - I review the query for inefficiencies, such as selecting unnecessary columns (SELECT *), using non-indexed columns in WHERE clauses, or inefficient joins and subqueries.
- Check Indexing and Table Statistics
  - I ensure the query is supported by appropriate indexes and that table statistics are up-to-date by running ANALYZE. If needed, I create or optimize indexes.
- Monitor and Optimize Database Resources
  - I check for resource bottlenecks (CPU, memory, disk I/O) that might impact query performance and explore optimizations like increasing work memory or parallel execution.
- Consider Database Scaling Options
  - If slow queries are due to high read traffic, I consider creating a read replica to distribute the load and improve overall performance.

10. **Write a Dockerfile to containerize a Laravel application**

ANS:

The code is on the root folder of the repository (Dockerfile)

Section 3: Monitoring and Troubleshooting

11. **How would you set up monitoring for the React Native mobile app's API endpoints?**

ANS:

First, I will configure my API backend to expose Prometheus-compatible metrics. This involves creating a /metrics endpoint that Prometheus can scrape for data on API requests and responses. I will ensure the backend is instrumented to track relevant metrics, such as request counts, error rates, and response durations.

Next, I will configure Prometheus to scrape metrics from the /metrics endpoint by adding the API server to Prometheus' prometheus.yml configuration file.

Once Prometheus is scraping the data, I'll set up Grafana and add Prometheus as a data source. I will create a dashboard in Grafana, adding panels to monitor key metrics such as:

- Total API requests: api_http_requests_total
- API response duration: api_http_duration_seconds

Finally, I will configure alerts in Grafana to notify me if there are any issues, such as high error rates or increased response times, ensuring timely action when performance degrades.

12. **Explain how you would debug high latency in the Node.js microservices.**

ANS:

To debug high latency, I will start by reviewing the application logs to identify any potential clues about the source of the issue.

Next, I'll check the resource utilization (CPU, memory, etc.) to ensure the service has sufficient resources to run efficiently.

I will also examine the database query performance to see if slow queries are contributing to the latency, and implement Database Query Caching with Redis to reduce the time spent on frequent queries.

Additionally, I will inspect the networking aspects to rule out any firewall or network-related issues affecting communication.

Finally, I will set up real-time monitoring and alerts using Prometheus and Grafana to track latency and error rates, ensuring that any performance degradation is detected promptly.

Section 4: Behavioral

13. **Describe a time you improved the performance of an infrastructure system. What challenges did you face?**

ANS:

The challenge I faced involved services hosted on an EKS cluster experiencing latency and resource bottlenecks. To address this, I started by analyzing system metrics using AWS Prometheus, which helped me identify the underperforming services and any resource constraints they were facing.

To improve performance, I scaled the infrastructure by configuring cluster autoscaling with Karpenter and pod autoscaling using Horizontal Pod Autoscaler (HPA). This allowed resources to be dynamically adjusted based on demand, ensuring that my services remained stable.

After implementing these changes, the issue was resolved, and it did not recur.

14. **How do you prioritize tasks when multiple urgent issues arise?**

ANS:

When multiple urgent issues arise, I prioritize tasks based on their impact and urgency. First, I assess the potential consequences of each issue, such as whether it affects critical systems, customer-facing services, or business operations.

Next, I consider the resources required to address each issue and the time sensitivity. For instance, if a production system is down, that will take precedence over other tasks,

and I will work on it first to minimize downtime. If the issues are equally urgent, I break them down into smaller, manageable steps to address them concurrently.

I also ensure effective communication with stakeholders to manage expectations and keep them updated on progress. I may delegate tasks when appropriate, ensuring that the team is aligned on priorities and working efficiently.

Once immediate issues are resolved, I conduct a post-mortem to understand the root cause and implement preventive measures to avoid similar situations in the future.