**Case Study.**

## Part 1: CI/CD and Version Control (8 minutes)

**Question 1:** How would you set up a CI/CD pipeline using Cloud Build and Cloud Deploy to manage the deployment of the containerized API service on GCP? Describe the process and tools you would use.

**Answer**

I would set up a CI/CD using the below steps

1. Create a git repository on the BitBucket with branches; dev, staging and production. Push the Java API service code to the created git repository.
2. Add a Dockerfile and a could build yaml (cloudbuild.yaml) to the repository. The docker file contains the instructions for building docker image. The cloudbuild yaml file contains CI/CD instructions
3. Create a Google Cloud project in the Google Cloud console and enable billing. Enable the Artifact Registry, Cloud Build, and Cloud Run APIs.
4. Create a service account in the IAM and grant the following roles
    - Cloud Run Admin
    - Access Container Registry
    - Logging And Monitoring
5. Create an image repository in the Google Artifact Registry
6. Create and configure Cloud Build trigger. The trigger is invoked whenever there is a new push to any branch of the source code repository.

Question 2: How would you manage version control for both the application code and the infrastructure code? Explain how you would structure the repositories in Bitbucket.

Answer
I would manage version control for both the application code and the infrastructure code by;

1. Creating and maintaining different branches for development, staging and production
2. Creating branches for new features and changes to the source code
3. Documenting the usage of the Iac code within the files
4. Using merge requests for reviewing code before merging changes to staging and production branches
5. Separating repositories for the application code and infrastructure code
6. Creating a CI/CD pipeline for each repository in cloud build

I would structure the repositories by;

1. Using Bitbucket workspace to group repositories by team.
2. Ensuring consistency in naming repositories
3. Organizing repositories based on their domain
4. Enforcing branching and using tags
5. Enforcing access control to repositories at a group level

The Java API service repository would have below structure in bitbucket
Workspaces:
  api-services
    java-api-service
      src/
      pom.xml/
      Dockerfile
      cloudbuild.yaml
      README.md

## Part 2: Containerization and API Management (8 minutes)

**Question3**: Explain how you would containerize the Java-based API service using Docker. What considerations would you keep in mind to ensure it runs efficiently in a containerized environment?

**Answer**
I would containerize the Java-based API service using docker by creating a Dockerfile in the repository. The Dockerfile would contain the following

1. An official OpenJDK slim  image as the base image
2. Working directory
3. Copy command to copy compiled JAR file into the container
4. A port on which the application will run
5. A command to run the application

I would consider the following to ensure the Java-based API runs efficiently in a containerized repository

1. Using a slim base image to reduce the image size for faster downloads and deployments.

2. Using environment variables for configuration to make the container more flexible and portable

**Question 4:** How would you utilize ApigeeX for API management? Describe the key features you would leverage and why they are important for this API service.

**Answer**

I would use Apigee X to manage the entire lifecycle of the API. Develop and manage APIs with a set of Restful operations such as deploying and securing API proxy, and monitor APIs . I would also use Apigee X to ensure the reliability, security, and observability of the API service

For this API service, I would leverage the following Apigee key features

1. Apigee proxies: - To create an API proxy for the API service. The API proxy maps the publicly available URLs to the API service. The developers will use the proxy HTTP endpoint to access the API service. This protects the service from direct exposure.
2. API Authentication - To enforce authentication using OAuth 2.0 or JWT tokens. Use API keys to control access for specific clients or applications. This ensures that only authorized users can access the API service
3. Rate Limiting and Traffic Management: To define quotas and spike arrest policies to prevent abuse or overload. This prevents the API service from DOS attacks and ensures consistent performance for all users
4. Analytics and Monitoring: Apigee X offers detailed API usage and performance analytics. Provides metrics such as latency, response times, error rates, and traffic volumes.
5. Caching: Apigee X supports response caching to reduce API service load thus enhancing API response time.
6. Developer Portal: This is used by the developers to discover the API service and use it to build an application

## Part 3: Observability, Monitoring, and Incident Management (8 minutes)

**Question 5**: What key metrics and logs would you monitor to ensure the API service is functioning correctly? How would you set up this monitoring using Cloud Monitoring and Cloud Logging?

**Answer**

I would monitor the following key metrics and logs to ensure that the API service is functioning correctly

1. Performance Metrics. These include:
    - Request Count - Measure the number of API requests per second. Provides an understanding of the traffic patterns and ensures the API scales with demand
    - Request Latencies. This metric provides the average latency for the API Response. Detects slow responses and ensures consistent performance
2. Reliability Metrics:
    - Error Rate: Track the proportion of failed requests (HTTP 4xx and 5xx responses) relative to total requests.

3. Resource Utilization:
    - CPU and Memory Usage: Monitor container-level metrics for the API service. Helps to detect resource bottlenecks.
    - Network Traffic: Tracks inbound and outbound data transfer rates. This helps identify unusual traffic patterns.
4. Security Metrics: Unauthorized Access Attempts. Track failed authentication attempts or unusual spikes in API key usage to detect potential abuse or malicious activity
5.

Key logs to monitor

1. Request Logs: Capture all incoming requests, including metadata such as method, endpoint, client IP, and response time.
2. Error Logs: Log all application-level exceptions and stack traces. Identify and resolve bugs in the Java service.
3. Authentication Logs: Record successful and failed login attempts, token generation, and invalidation. Monitor for security breaches or suspicious behavior
4. System Logs: Collect container logs, JVM logs (e.g., garbage collection), and infrastructure logs. Diagnose issues at the system or runtime level

Setting Up Monitoring in Cloud Monitoring

1. Integrating Metrics to Enable Metrics Collection
2. Create a custom dashboard to visualize metrics such as Latency over time. Request throughput (RPS). Error rate trends, CPU, memory, and network usage,
3. Set SLIs/SLOs:
    - Define Service Level Indicators (SLIs) such as latency and availability.
    - Create Service Level Objectives (SLOs) (e.g., 99.9% uptime, <200ms latency).
4. Set up alerts for critical metrics:
    - Latency exceeds thresholds (e.g., >500ms for 95th percentile).
    - Error rate exceeds thresholds (e.g., >5% over 5 minutes).
    - CPU usage >80% or memory usage >75%.
    - Use incident grouping to consolidate related alerts
5. Notification Channels: Configure notification channels such as email, Slack, PagerDuty, or SMS for incident alerts.

Setting Up Logging in Cloud Logging

1. Enable Cloud Logging
2. Create metrics from logs, such as the count of 5xx responses or failed authentication attempts.
3. Log Analysis: Use Log Explorer to query logs with filters
4. Set appropriate retention policies for logs to balance cost and utility

**Question 6:** If an incident occurs where the API service becomes unresponsive, how would you approach troubleshooting and resolving the issue? Explain your incident management process.

**Answer**

I would approach troubleshooting and resolving the issue by

1. Checking on the alerts received from cloud monitoring for high latency, error rate and unavailability.
2. Checking on logs in the cloud logging to identify if there errors in the logs
3. Responding to alerts or errors from the logs
4. Notify the SRE team

Incident Management Process

1. Acknowledge the incident
2. Identify the issue through alerts, monitoring, or user reports
3. Assess the severity and impact of the incident
4. Limit the damage to affected systems or users
5. Investigate the root cause of the problem
6. Implement fixes
7. Analyze the incident to prevent recurrence

## Part 4: Infrastructure as Code (IaC) and Architecture Mindset (6 minutes)

**Question 7** How would you use Infrastructure as Code (IaC) to provision and manage the infrastructure for the API service in GCP? Which tools would you use, and how would this approach benefit the SRE team?

**Answer**

How I would use Infrastructure as Code (IaC) to provision and manage the infrastructure for the API service in GCP.

I would use the Terraform provider for Google Cloud to provision and manage Google Cloud infrastructure the infrastructure for the API service in GCP.

I would use the following tools

1. Terraform: to configure and manage Google Cloud infrastructure using code
    - HashiCorp Terraform is an IaC tool that lets you define resources in cloud and on-premises in human-readable configuration files that you can version, reuse, and share. You can then use a consistent workflow to provision and manage all of your infrastructure throughout its lifecycle

2. Infrastructure Manager
    - To automate the deployment of your Terraform configuration

3. Terraform Cloud and Terraform Enterprise
    - Terraform Cloud is a software as a service (SaaS) application that runs Terraform in a stable, remote environment and securely stores state and secrets.

Adopting IaC with tools like Terraform standardizes the management of the API service infrastructure. It enhances the SRE team's efficiency by providing consistent, automated, and auditable processes, ensuring the infrastructure can meet the API service's performance and reliability needs

**Question 8**. Describe your approach to ensuring the infrastructure is resilient and scalable. How would you architect the solution to handle varying loads and potential failures?

**Answer**
The approach to ensuring the infrastructure is resilient and scalable is by creating a robust architecture that can handle varying loads and gracefully recover from failures

Resilience
Ensure the system can withstand and recover from failures with minimal impact on availability and data integrity.

Resilience Strategies
1. High Availability. Deploy the API service across multiple zones within a region using Cloud Run.
- Use zonal distribution to ensure that a single zone failure does not affect the service.
- Multi-Region Failover: For global users, set up failover mechanisms using Cloud Load Balancer with backend services in multiple regions.
2. Redundancy. Load Balancers: Use a Cloud HTTP(S) Load Balancer to distribute traffic across multiple instances or zones.

Scalability
Design to handle traffic spikes and scale down during low-demand periods to optimize costs.

Scalability Strategies
Horizontal Scaling: Containerized Services:
- Deploy the API with autoscaling enabled based on CPU, memory, or custom metrics.

Vertical Scaling: Use instance groups with autoscaling to adjust resource allocation dynamically for VMs.

Traffic Management: Use Global Load Balancer to automatically distribute traffic across regions.

I would architect the solution to handle varying loads and potential failures by

1. Use Cloud Load Balancer for incoming requests
2. Deploy the API service with autoscaling and redundancy.
3. Logging and Monitoring: Centralize logs in Cloud Logging and set up dashboards in Cloud Monitoring.
4. Autoscaling