Architecting and Managing a Critical API Service on Google Cloud Platform (GCP)

Background: Your company has recently migrated to Google Cloud Platform (GCP). As part of the Site Reliability Engineering (SRE) team, you are responsible for ensuring the reliability, scalability, and observability of a critical API service that handles user authentication. This service, written in Java, needs to be containerized and deployed to GCP. Your team uses a CI/CD pipeline, version control with Bitbucket, and infrastructure as code (IaC) practices.

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.
  Cloud Build - I would use Cloud Build for continuous integration(CI) to automate the building, testing and packaging of the containerized Java API. Automated testing would be a key integration point in order to ensure that both security and functional requirements are met at every stage.
  Cloud Build would incorporate security scanning (e.g., for vulnerabilities using tools like Container Analysis) to maintain a secure codebase.

  Cloud Deploy - I would leverage Cloud Deploy to manage continuous delivery (CD), orchestrating deployments across the staging and deployment environments, deploying the containerized service to a Cloud Run environment.

  Steps for Setting up the CI/CD pipeline
  1. Code Pull and Build - Using Cloud Build, pull the latest code from Bitbucket, run unit and integration tests, build the docker image and tag the image with a unique identifier.

  2. Image Push - Push the already built image to the Artifact Registry.

  3. Deployment - Cloud Deploy references the specified Cloud Run service configurations, and deploys to a staging environment and later to production after approvals.

- **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.

Following GitOps principles, I'd maintain separate repositories for the application and infrastructure code. This would promote scalability, security and structured collaboration across the various teams. To ensure consistency and traceability in deployments while enforcing access control, the application code repository would have a clear branching strategy which would include main, develop, feature, release and hotfixes branches. For code quality, PRs and code reviews would be enforced.

For the IaC repository, I would maintain a similar branching strategy and use Terraform. Here all the GCP resources would be defined. To ensure deployments are consistent and repeatable, I'd use tagging in both repositories to correlate the application and infrastructure code.

Part 2: Containerization and API Management (8 minutes)

- **Question 3:** 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?
  To containerize the service, I would use a Dockerfile starting with a lightweight base image to improve the container startup time, set up multi-stage builds to optimize the image size and manage resources by defining CPU and memory limits using GKE.

  Considerations:
  1. Use a lightweight base image to improve container startup time.
  2. Use official, trusted images to reduce vulnerabilities and ensure up-to-date security patches.
  3. Implement health checks to confirm the service is running correctly inside the container.
  4. Use environment variables for configurations to avoid hardcoding, making the container environment-agnostic.

- **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.
  1. Security and Access Control: ApigeeX would serve as the API gateway to enforce access control via OAuth2 and JWT, essential for protecting sensitive user authentication data.
  2. Analytics and Rate Limiting: Leveraging ApigeeX analytics helps in understanding API performance and usage patterns. Rate limiting ensures that any spikes are managed without overwhelming the service, contributing to fault tolerance.
  3. API Gateway: ApigeeX would serve as the API gateway, handling request routing, traffic control, and caching to reduce load on the backend.

Key Features:

1. Authentication and Authorization: Set up ApigeeX policies to manage access control via API keys, OAuth2, and JWT validation, ensuring only authorized requests reach the API service.
2. Rate Limiting: Define rate-limiting policies to prevent abuse and manage traffic spikes gracefully.
3. Analytics: Leverage ApigeeX's analytics to gain insights into API usage patterns, latency, and error rates.

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?

I would use Cloud Monitoring and Cloud Logging (Stackdriver) for monitoring and logging. I would set up for various key metrics and set alerts for the key metrics and log patterns to provide visibility into the system health. In addition, I'd use Cloud Monitoring and Cloud Logging to track security-related logs, error patterns, and suspicious activity, ensuring compliance and security best practices are upheld.

The key metrics I would monitor are:
API Latency - Track response times to identify performance issues.
Error rates - Monitor HTTP 4xx and 5xx status codes to detect issues in real-time.
Throughput - Observe the number of requests per second to gauge traffic patterns.

- **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.
  In case of service unresponsiveness, automated alerts would trigger the incident response. The approach would include analyzing logs, reviewing recent deployments, and validating configurations.

Incident Response:

1. Identify: Use alerts from Cloud Monitoring and Cloud Logging to detect the issue quickly.
2. Investigate: Analyze metrics, logs, and ApigeeX analytics to narrow down the problem. This may involve checking load, latency spikes, or error patterns.
3. Resolution: Based on findings, decide if a rollback, redeployment, or resource adjustment is needed.

4. Post-Incident Review: Conduct a post-incident analysis to determine root causes and define preventive measures. This review would include security incident response considerations to bolster resilience against future incidents.
5. Document the incident, including root cause, impact, and resolution steps. Run a blameless post mortem to identify ways 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?
  Terraform would be my tool of choice to provision and manage Google Cloud resources because it simplifies the creation and maintenance of infrastructure at scale due to its declarative syntax.
  Steps to I'd use to set up and manage GCP Infrastructure Using Terraform:
  1. Define the Google Cloud Provider. This enables authentication and interaction with the GCP services by specifying project IDs, regions and zones.
  2. Create Infrastructure Modules: Breakdown infrastructure components into reusable modules for core resources.
  3. Provision Resources with Terrafoam Code: Write Terraform configuration files to define resources needed for the service. I would define a GKE cluster for containerized deployments, load balancers for distributing traffic and Cloud SQL for persistent storage. I would also configure IAM roles and policies to enforce secure access to resources.
  4. Use Remote State Management: I would store the Terraform state files in Cloud Storage to maintain consistency and enable collaboration. To ensure safety and predictable deployments, I'd enable state locking to prevent concurrent changes.
  5. Automation and Version Control Integration: I would integrate Terraform with Cloud Build to automate infrastructure changes and allow for changes to be deployed only after peer review ensuring quality control. In addition, I'd store the Terraform configuration in Bitbucket to maintain a clear history of infrastructure changes and enable easy rollbacks when need be.

  Benefits to the SRE Team are:

  1. Consistency and Reproducibility: Terraform ensures that infrastructure is consistently provisioned across environments, reducing drift and configuration errors.
  2. Scalability and Flexibility: With modularization, the SRE team can quickly scale resources, add environments, or make changes by updating the codebase.

3. Auditability and Security: All infrastructure changes are version-controlled, providing an audit trail for compliance. Additionally, by automating infrastructure provisioning, the team can enforce security best practices at scale.
4. Ease of Collaboration: Remote state management and code review processes streamline team collaboration and reduce the risk of misconfiguration.

- **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?

To ensure the API service infrastructure is resilient and scalable, I would adopt a multi-layered architectural approach with a focus on high availability, fault tolerance, and automated scaling.

Key Architectural Strategies:

1. Use Managed Services for Core Infrastructure:
   Google Kubernetes Engine (GKE): Deploy the containerized API on GKE for automated scaling and resource management. Kubernetes handles load balancing and self-healing, which increases resilience.
   Cloud Load Balancing: Use a global load balancer to distribute incoming traffic across multiple instances or clusters, ensuring high availability and low latency for end-users.
2. Implement Auto-scaling:
   Configure horizontal auto-scaling in GKE to automatically adjust the number of replicas based on CPU/memory usage or request load. This ensures the service can handle traffic spikes while optimizing costs during low-traffic periods.
   Similarly, use Cloud SQL's auto-scaling features if the service relies on a database, which enables automated performance scaling as demand changes.
3. Set Up Multi-zone or Multi-region Deployments:
   Deploy the API service across multiple zones within a region to ensure availability even if a specific zone goes down. For more critical services, use multi-region deployments to achieve geographic redundancy and improve latency for global users.
   Terraform can be configured to deploy resources across multiple regions, further increasing resilience and ensuring that failover options are in place.
4. Optimize with Caching and CDN:
   For static assets and frequently accessed resources, configure Cloud CDN to reduce latency and offload traffic from the backend services.

For data caching, consider using Redis or Memorystore to cache frequent queries and API responses, enhancing response times and reducing strain on backend resources.

5. Monitoring and Alerting:
   Use Cloud Monitoring and Cloud Logging to set up alerting and track key metrics such as request latency, error rates, and CPU/memory utilization.
   Set up health checks on instances and pods to ensure proactive identification of potential failures. Automatic failover and automated restarts are configured in GKE to handle service restarts without human intervention.
6. Disaster Recovery and Backups:
   Implement automated backups for stateful resources, like databases, and store them in multi-region Cloud Storage buckets. This ensures quick recovery in case of data loss or regional outages.
   Conduct regular disaster recovery drills to validate that recovery objectives meet the business needs.