Microservices Architecture Proposal

Objective: Transform the existing monolithic backend architecture into a scalable, maintainable, and resilient microservices architecture using Google Cloud Platform (GCP) services like App Engine, Cloud Functions, Cloud Datastore, and other relevant services. The architecture will be designed to handle user management, interaction tracking, and fraud detection while addressing challenges related to data consistency, API versioning, and fault tolerance.

Proposed Microservices Architecture

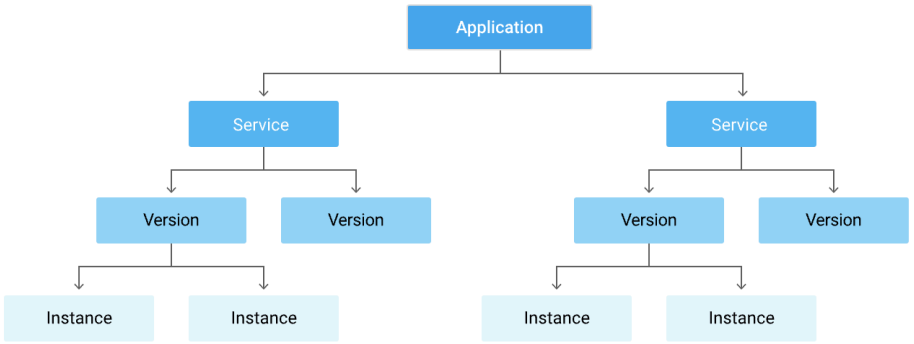
1. **User Service:**
    - **Responsibilities:** Manage user profiles, including user creation, updates, and retrieval.
    - **APIs:**
        - `POST /users`: Create a new user.
        - `GET /users/{userId}`: Retrieve user profile by ID.
        - `PUT /users/{userId}`: Update user profile.
        - `DELETE /users/{userId}`: Delete user profile.
    - **Data Storage:** Use Cloud Datastore for storing user profiles, which supports scalability and flexible data models.
    - **Deployment:** Deploy on App Engine Flexible Environment to handle variable workloads and auto-scaling.
2. **Interaction Service:**
    - **Responsibilities:** Record and manage user interactions such as likes and visits.
    - **APIs:**
        - `POST /interactions/likes`: Record a new user like interaction.
        - `POST /interactions/visits`: Record a new user visit interaction.
        - `GET /interactions/likes/{userId}`: Retrieve likes for a user.
        - `GET /interactions/visits/{userId}`: Retrieve visits for a user.
        - `GET /interactions/visitors/{targetUserId}`: Retrieve users who visited a specific profile.
    - **Data Storage:** Use Cloud Datastore for storing interaction records to support high throughput and scalability.
    - **Deployment:** Deploy on App Engine Flexible Environment for consistent performance and scaling.
3. **Fraud Detection Service:**
    - **Responsibilities:** Detect and handle fraudulent activities based on user interaction patterns.
    - **APIs:**
        - `POST /fraud/detect`: Trigger fraud detection for a user.
        - `GET /fraud/alerts`: Retrieve fraud alerts or detected issues.

- **Data Storage:** Use Cloud Datastore or Bigtable for storing fraud detection logs and alerts.
- **Deployment:** Use Cloud Functions for serverless fraud detection tasks, providing scalability and cost efficiency.

An overview of App Engine

An App Engine application consists of a single application resource that includes one or more services. Each service can be configured with different runtimes and performance settings. Within each service, you can deploy multiple versions, and each version runs on one or more instances based on the traffic handling configuration.

Hierarchy



When you create an App Engine application in your Google Cloud project, you establish a top-level container that encompasses all your app's resources, including services, versions, and instances. This application is set up in a specific region, where it houses your app code, configuration settings, credentials, and metadata. The App Engine application is made up of various resources, which can be either standard or flexible, depending on your requirements and the chosen region. Each App Engine application must have at least one service, known as the default service, which can host multiple versions based on your billing status.

Data Flow and Communication

1. **User Service and Interaction Service:**
   - **Communication:** Use kafka for communication. Interaction Service will call User Service APIs to validate users during interaction recording.
   - **Data Flow:** When a user interaction occurs, Interaction Service checks the validity of the user via User Service, records the interaction, and updates any relevant data.
2. **Interaction Service and Fraud Detection Service:**
   - **Communication:** Kafka (or RabbitMQ) for asynchronous messaging.
   - **Data Flow:** Interaction Service publishes interaction events to Kafka topics, which the Fraud Detection Service subscribes to for processing.

3. **User Service and Fraud Detection Service:**
    - **Communication:** Messaging for fraud alerts related to user interactions. Fraud Detection Service may need to query User Service for additional user details.

Fault Tolerance Mechanisms

1. **Circuit Breakers:**
    - **Implementation:** Use libraries like Resilience4j to implement circuit breakers that prevent cascading failures and handle service outages gracefully.
2. **Retries and Fallbacks:**
    - **Implementation:** Configure retries with exponential backoff for transient errors. Define fallback mechanisms to provide default responses or alternative actions during service failures.

Message Broker Integration

- **Kafka** is a distributed streaming platform designed for handling large-scale data streams and real-time analytics, known for its high throughput and durability.
- **RabbitMQ** is a message broker that facilitates the exchange of messages between systems, excelling in transactional messaging with its versatile exchange types and routing capabilities.

Kafka vs. RabbitMQ Comparison

1. **Performance**:
    - **Kafka**: Handles up to 1 million messages per second.
    - **RabbitMQ**: Manages between 4K-10K messages per second.

2. **Message Retention**:
    - **Kafka**: Retains messages based on policies, such as 30-day retention.
    - **RabbitMQ**: Retention is acknowledgment-based.

3. **Data Type**:
    - **Kafka**: Suitable for operational data.
    - **RabbitMQ**: Ideal for transactional data.

4. **Consumer Mode**:
    - **Kafka**: Operates with a dumb broker but smart consumer model.
    - **RabbitMQ**: Uses a smart broker with dumb consumers.

5. **Topology**:
    - **Kafka**: Publish/subscribe architecture.
    - **RabbitMQ**: Supports various exchange types like Direct, Fan out, Topic, and Header-based.

6. **Payload Size**:
    - **Kafka**: Default 1MB limit per message.
    - **RabbitMQ**: No payload size constraints.

7. **Usage Cases**:
    - **Kafka**: Best for massive data and high throughput scenarios.
    - **RabbitMQ**: Suitable for simpler use cases.

Given our requirements for handling large-scale data with high throughput, **Kafka** is the more appropriate choice.

Data Consistency and API Versioning

1. **Data Consistency:**
    - **Solution:** Utilise eventual consistency where appropriate. Ensure that services handle eventual consistency by processing events asynchronously and updating databases as needed.
2. **API Versioning:**
    - **Solution:** Implement API versioning using URL paths (e.g., `/v1/users/{userId}`) to manage changes and maintain backward compatibility.

Compatibility with App Engine Environment

- **Deployment:** Use App Engine Flexible Environment for User Service and Interaction Service for scalability and ease of management.
- **Serverless Functions:** Deploy Fraud Detection Service as Cloud Functions to handle fraud detection logic in a scalable and cost-effective manner.
- **Data Storage:** Use Cloud Datastore for user profiles and interactions, and Bigtable for large-scale fraud detection logs.

Monitoring and Access Management

1. **Cloud Monitoring:**
    - **Setup:** Implement Cloud Monitoring to track the performance, availability, and error rates of microservices. Set up dashboards and alerts for proactive management.

2. **Cloud IAM:**
    - **Setup:** Use Cloud IAM to manage access control and permissions for microservices. Define roles and policies to ensure secure and appropriate access to resources.

Summary

The proposed microservices architecture breaks down the monolithic backend into distinct services handling user management, interactions, and fraud detection. It leverages GCP services like App Engine, Cloud Functions, and Cloud Datastore for scalable, reliable, and maintainable service deployment. The architecture addresses key challenges of data consistency, API versioning, and fault tolerance, while ensuring compatibility with App Engine's deployment environment and incorporating robust monitoring and access management practices.