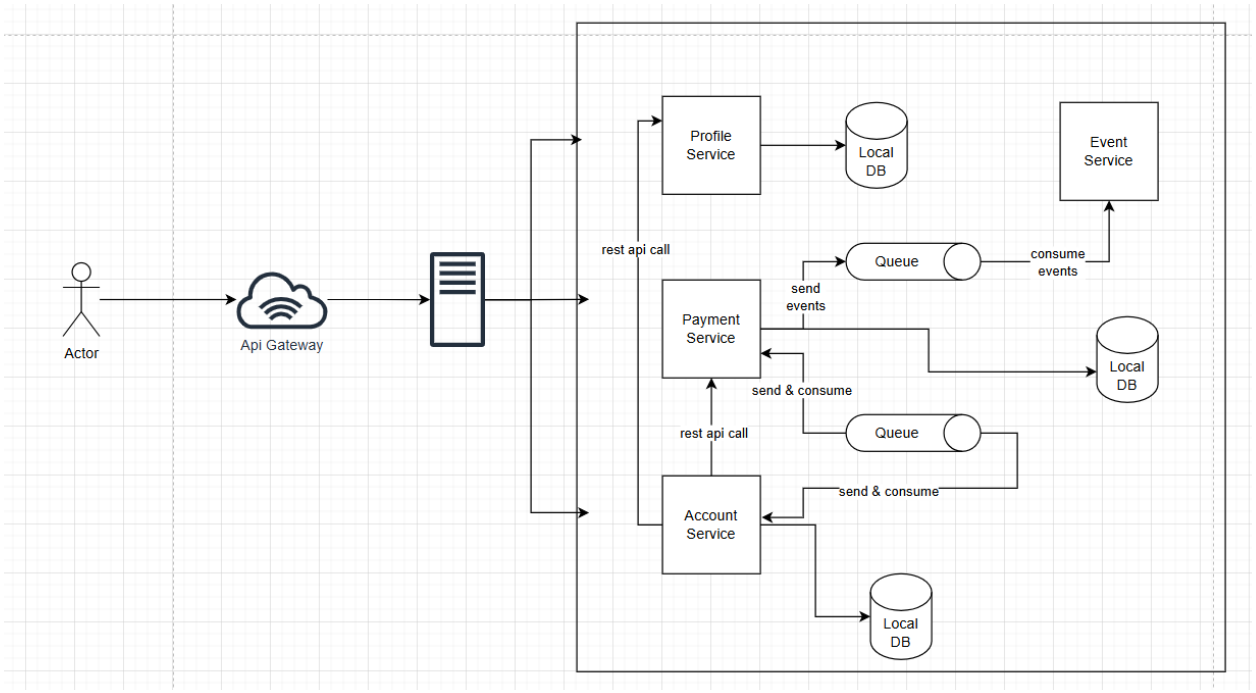
Provide a high-level architecture diagram of the system, showing how the microservices will interact with each other.



The architecture follows a microservices-based, hybrid model combining REST and event-driven communication. External requests pass through an API Gateway, which handles authentication and routing. Core services such as Profile, Payment, and Account operate independently, each with its own database. Synchronous REST APIs are used for operations requiring immediate validation, while RabbitMQ facilitates asynchronous, event-driven interactions through a Saga choreography pattern. An Event Service consumes domain events for auditing and future projection use cases.

Explain the choice of architecture, including considerations for security (e.g., encryption, authentication, authorization) and inter-service communication.

The chosen architecture is a microservices-based, event-driven architecture leveraging the Saga pattern for managing distributed transactions. This approach supports scalability, fault isolation, and flexibility in service evolution.

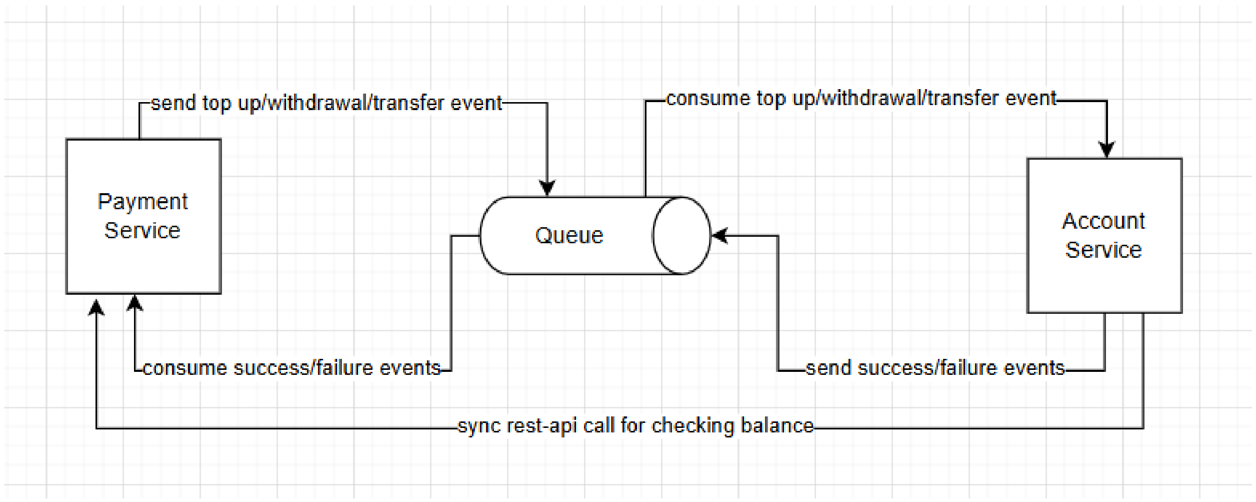
<u>Security Considerations:</u> - For authentication and authorization, I used JSON Web Tokens (JWT) combined with Role-Based Access Control (RBAC) to ensure secure and granular access management. While encryption was not implemented in the current setup, I would incorporate HTTPS to secure communication channels and encrypt sensitive configuration data using external secrets management solutions. Additionally, application logs are sanitized by redacting or hashing sensitive information to prevent leakage of personally identifiable or financial data.

<u>Inter-service Communication:</u> - Inter-service communication primarily relies on RabbitMQ, supporting an event-driven, loosely coupled architecture using the Saga pattern with choreography. This was particularly useful for coordinating actions between services such as Payment and Account. For operations requiring immediate consistence such as top-ups, withdrawals, and transfers. I also used synchronous REST API calls. REST APIs were

similarly employed between the Profile and Account services to validate customer existence before account creation, ensuring data integrity.

Discuss how you would implement data consistency across microservices.
To maintain data consistency across microservices, I implemented the Saga pattern using a choreography-based approach. This was particularly effective in coordinating transactions between services such as Payment and Account, allowing each service to manage its own local transactions while communicating state changes through events.

In addition to the Saga pattern, I also employed REST API integrations between services for synchronous operations where immediate data validation was required. This hybrid approach ensured both eventual consistency for long-running processes and strong consistency where needed.


send top up/withdrawal/transfer event
consume top up/withdrawal/transfer event
Payment Service
Queue
Account Service
consume success/failure events
send success/failure events
sync rest-api call for checking balance

Describe your approach to scaling the services, handling high availability, and disaster recovery.
<u>Scaling Services</u> - I prefer deploying services on container orchestration platforms like Kubernetes or OpenShift, ideally using Amazon EKS for managed scalability. Each node would be part of an Auto Scaling Group to ensure compute resources adjust based on demand. At the service level, I would configure Horizontal Pod Autoscalers (HPA) to dynamically scale pods based on CPU/memory utilization or custom metrics.

<u>High Availability and Disaster Recovery</u> - To ensure high availability, I would deploy services across multiple Availability Zones (AZs), leveraging load balancers to distribute traffic and minimize downtime in case of AZ failure. For databases, I would implement a master-slave (primary-replica) setup routing write operations to the primary instance and distributing read operations across replicas to enhance performance and availability. For disaster recovery, I would ensure automated backups, cross-region replication where applicable, and well-documented failover procedures.