AAA ANSA prep - Generic payment design

System Overview: The payment system is designed as a platform that enables merchants to process payments, manage customer accounts, and offer loyalty programs. It supports various payment methods, including credit cards, debit cards, and ACH transfers, and integrates with popular payment gateways like **Stripe** and **Authorize.Net**.

Key Components:

1. **Merchant API:** A **RESTful** and **GraphQL** API that allows merchants to create and manage customer accounts, process payments, issue refunds, and retrieve transaction history. The API is secured using **API keys** and **JWT tokens** for authentication and authorization.
2. **Customer Accounts:** Each customer has a unique account associated with a merchant, storing information such as customer ID, name, email, and encrypted payment method details. Customer accounts maintain a **balance ledger** for tracking funds added, spent, and refunded.
3. **Payment Processing:** The system integrates with payment gateways (e.g., **Stripe**, **Braintree**) to securely process payments. When a customer makes a purchase, the system creates a **charge request** to the payment gateway, handles the response, and updates the customer's balance accordingly. The system also supports refunds and void transactions.
4. **Loyalty Programs:** Merchants can create and manage loyalty programs, such as points or cashback, through the **Merchant API**. The system tracks **loyalty rewards** earned by customers and allows merchants to define redemption rules.
5. **Fraud Detection:** The system incorporates fraud detection mechanisms (e.g., **Sift**, **Ravelin**), such as velocity checks, IP geolocation, and machine learning models, to identify and prevent fraudulent transactions. Suspicious activities are flagged for manual review or automatically rejected based on predefined rules.
6. **Notifications:** The system sends transactional notifications to customers and merchants via email (**SendGrid**), SMS (**Twilio**), or push notifications, confirming payments, refunds, and balance updates.
7. **Reporting and Analytics:** Merchants have access to a **reporting dashboard** that provides insights into transaction volumes, customer behavior, and loyalty program performance. The system generates real-time and historical reports to help merchants make data-driven decisions.
8. **Load Balancer: NGINX** and **HAProxy** distribute incoming traffic across multiple instances of services, ensuring high availability and reliability. They also handle **SSL/TLS** termination to secure data in transit.
9. **Circuit Breaker: Hystrix** and **Resilience4j** are used to monitor and limit failures in the system, providing resilience against cascading failures and ensuring service availability during partial outages.
10. **PCI DSS Compliance:** Ensures the system adheres to **PCI DSS** standards for securely storing, processing, and transmitting cardholder data. This includes implementing **security policies**, **encryption**, and **access controls**.

11. **Payment Gateway: Stripe** and **Authorize.Net** handle the routing of transactions to banks. They provide the necessary interfaces for processing various types of payments and managing transaction statuses.
12. **Database:** A distributed database system (e.g., **MySQL**, **PostgreSQL**) stores customer accounts, transaction data, and loyalty program information, ensuring high availability, fault tolerance, and optimized query performance.
13. **Caching: Redis** and **Memcached** are employed to cache frequently accessed data, such as customer balances and transaction history, improving response times and reducing database load.
14. **Message Queue: Kafka** and **RabbitMQ** are used to decouple services and ensure reliable asynchronous processing of tasks. This includes sending notifications and updating loyalty rewards, among other tasks.
15. **Encryption: AES-256** and **TLS** are used to encrypt data at rest and in transit, respectively, ensuring the security and confidentiality of sensitive information.

Here are the top ten keywords to use during your interview with Ansa, based on the provided context:

1. Payments: Discuss your experience with payment systems, particularly your work at Shipt integrating Avalara Tax Compliance and Stripe's Payments API.
2. Scalability: Highlight your experience in improving system performance and scalability, such as your migration from Ruby on Rails to Golang at Shipt.
3. Reliability: Mention your implementation of a highly redundant distributed database using CockroachDB and Kubernetes, which improved transaction reliability.
4. Integration: Talk about your experience with seamless integrations, like the one you orchestrated during Shipt's acquisition by Target.
5. Leadership: Discuss your leadership experience, such as leading a team of student support specialists at Humboldt State University.
6. Innovation: Showcase your innovative projects, like the open-source continuous integration platform you authored at Humboldt State University.
7. Adaptability: Highlight your ability to work on diverse projects and technologies, as demonstrated by your experience at RetailGo and True Market LLC.
8. Collaboration: Express your enthusiasm for working closely with Ansa's product designer and go-to-market roles to develop new product initiatives.
9. Growth Mindset: Convey your excitement about the opportunity to rapidly grow and scale Ansa's payment infrastructure to millions of dollars in processed volume.
10. Culture Fit: Emphasize your alignment with Ansa's values and your appreciation for their tight-knit, in-person team culture.

By incorporating these keywords and discussing your relevant experiences, you'll effectively demonstrate your skills, expertise, and fit for the role at Ansa.

Architecture:

The payment system follows a **microservices architecture**, with each component (e.g., **Merchant API**, **Payment Processing**, **Fraud Detection**) implemented as a separate service. The services communicate

through well-defined APIs and **message queues**, ensuring loose coupling and scalability.

The system uses a **distributed database** (e.g., **MySQL**, **PostgreSQL**) to store customer accounts, transaction data, and loyalty program information, ensuring high availability and fault tolerance. The database is designed with proper indexing and partitioning strategies to optimize query performance.

Caching mechanisms (e.g., **Redis**, **Memcached**) are employed to improve response times for frequently accessed data, such as customer balances and transaction history.

Security:

The payment system adheres to **PCI DSS compliance** requirements, ensuring the secure storage, processing, and transmission of sensitive cardholder data. All data is encrypted at rest and in transit using industry-standard encryption algorithms (e.g., **AES-256**, **TLS**).

Secure coding practices, such as input validation, parameterized queries, and CSRF protection, are followed to prevent common web application vulnerabilities.

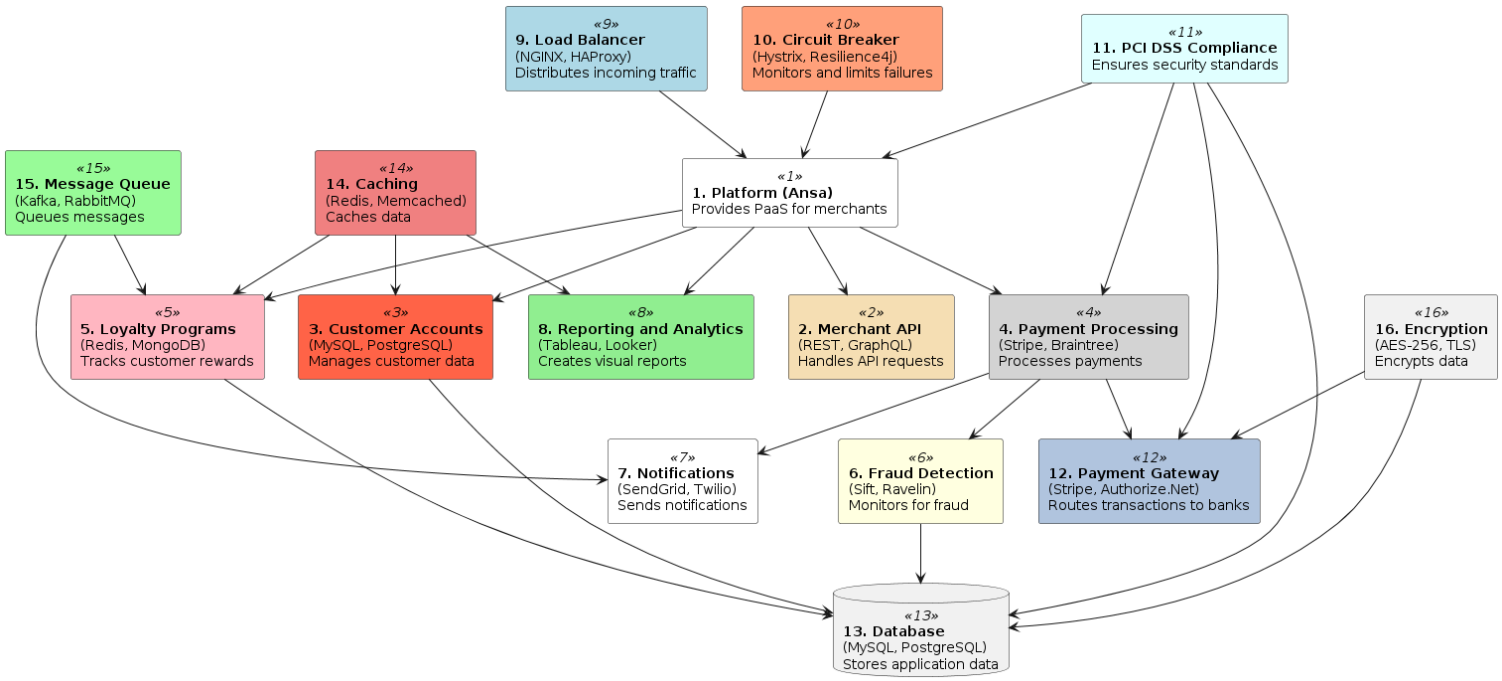
Scalability and Resilience:

The system is designed to handle high transaction volumes and scale horizontally by adding more instances of each service as needed. **Load balancers** distribute traffic evenly across service instances.

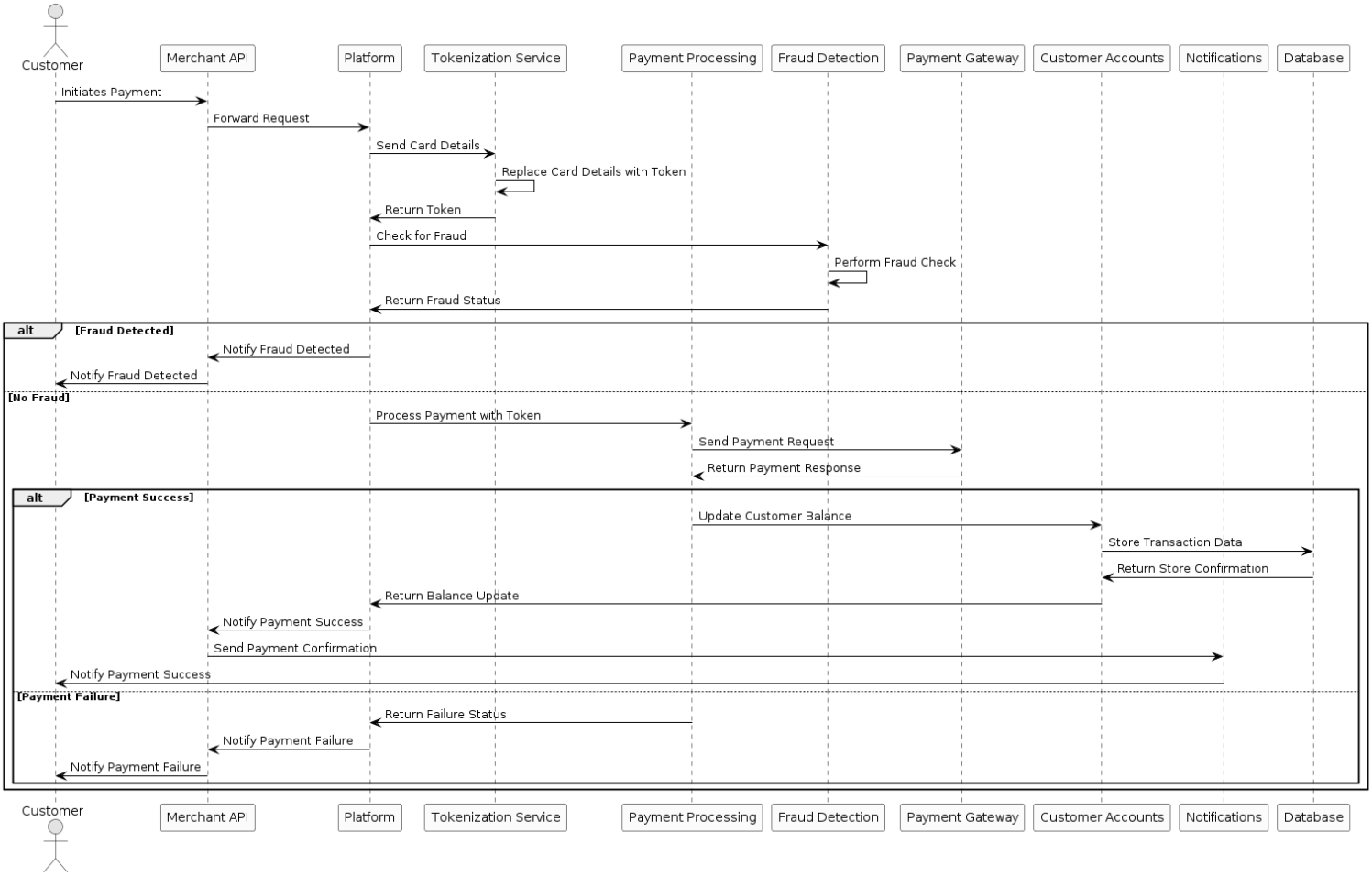
Message queues (e.g., **Kafka**, **RabbitMQ**) are used to decouple services and ensure reliable asynchronous processing of tasks, such as sending notifications or updating loyalty rewards.

Circuit breakers (e.g., **Hystrix**, **Resilience4j**) and retry mechanisms are implemented to handle temporary failures and prevent cascading failures across services.

High Level Architecture Diagram:



Sequence Diagram: Processing a Payment with Tokenization



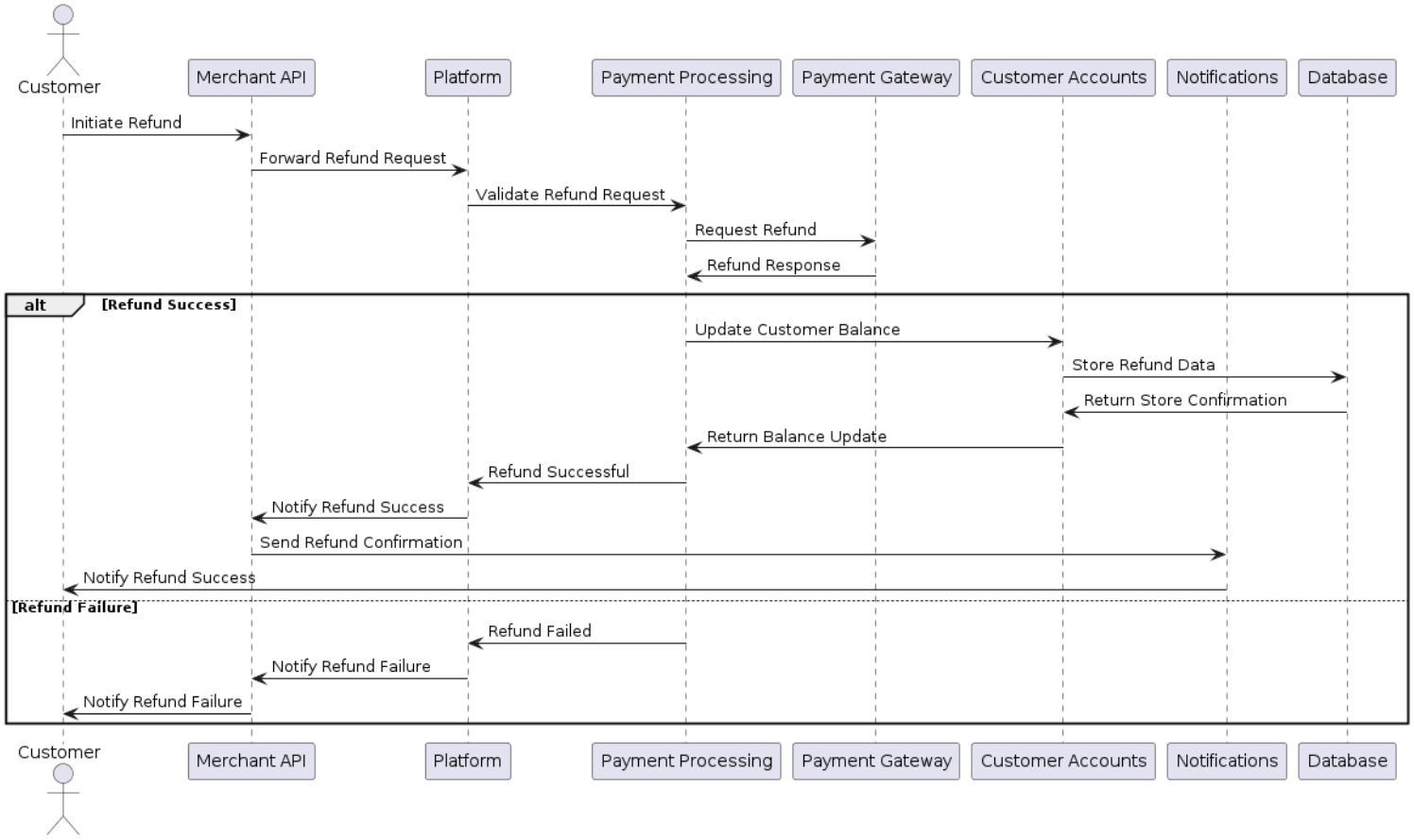
Explanation of Sequence Diagram:

1. **Customer Initiates Payment:**
   - The customer initiates a payment, sending the request to the `Merchant API`.
2. **Request Forwarding:**
   - The `Merchant API` forwards the request to the Ansa `Platform`.
3. **Tokenization:**
   - The `Platform` sends the card details to the `Tokenization Service`.
   - The `Tokenization Service` replaces the card details with a token and returns the tokenized details to the `Platform`.
4. **Fraud Check:**
   - The `Platform` requests the `Fraud Detection` service to perform a fraud check.
   - The `Fraud Detection` service performs the fraud check and returns the fraud status to the `Platform`.
5. **Fraud Detected:**
   - If fraud is detected, the `Platform` notifies the `Merchant API`, which then notifies the customer.
6. **No Fraud Detected:**
   - If no fraud is detected, the `Platform` forwards the payment request to the `Payment Processing` component.

7. **Payment Processing:**
    - The `Payment Processing` component sends the payment request to the `Payment Gateway`.
    - The `Payment Gateway` processes the payment and returns the response to the `Payment Processing` component.
8. **Payment Success:**
    - If the payment is successful, the `Payment Processing` component updates the customer's balance in `Customer Accounts`.
    - The `Customer Accounts` component stores the transaction data in the `Database`.
    - The `Database` returns the store confirmation to the `Customer Accounts`, which then updates the `Platform`.
    - The `Platform` notifies the `Merchant API` of the payment success.
    - The `Merchant API` sends a payment confirmation to the `Notifications` service, which then notifies the customer of the payment success.
9. **Payment Failure:**
    - If the payment fails, the `Payment Processing` component returns the failure status to the `Platform`.
    - The `Platform` notifies the `Merchant API`, which then notifies the customer of the payment failure.

Sequence Diagram for Refund Process:

This sequence diagram details the interactions involved in processing a refund. It shows the steps from initiating a refund request by the customer to updating the customer balance and notifying the customer about the refund status.



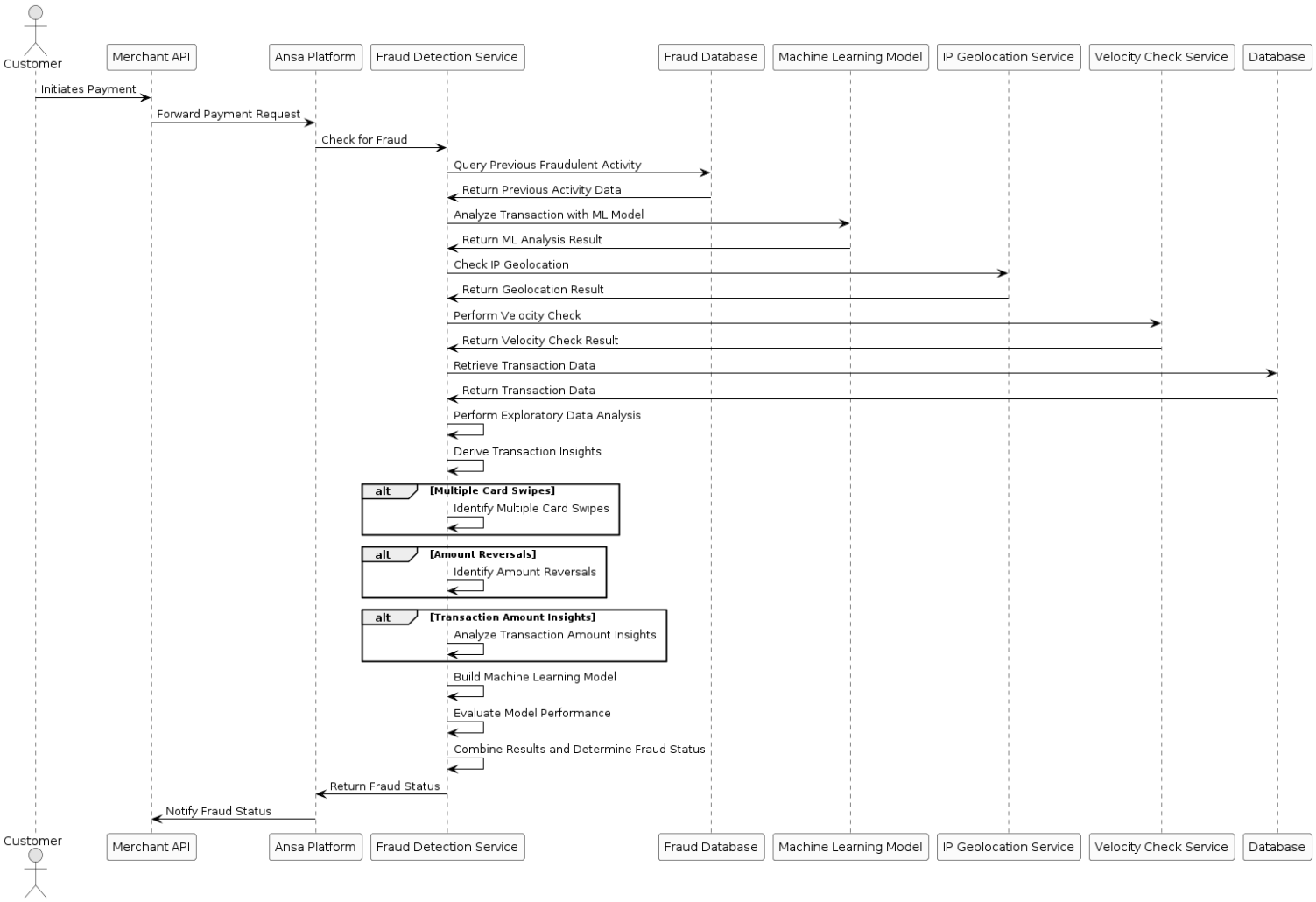
Explanation of Refund Sequence Diagram:

- **Customer Initiates Refund:**
  - The customer initiates a refund request, sending the request to the `Merchant API`.
- **Request Forwarding:**
  - The `Merchant API` forwards the refund request to the Ansa `Platform`.
- **Refund Validation:**
  - The `Platform` sends the refund request to the `Payment Processing` component for validation.
- **Refund Request:**
  - The `Payment Processing` component sends the refund request to the `Payment Gateway`.
- **Refund Response:**
  - The `Payment Gateway` processes the refund request and returns the response to the `Payment Processing` component.
- **Refund Success:**
  - If the refund is successful, `Payment Processing` updates the customer's balance in `Customer Accounts`.
  - The `Customer Accounts` component stores the refund data in the `Database`.
  - The `Database` returns the store confirmation to the `Customer Accounts`, which then updates the `Payment Processing` component.
  - The `Payment Processing` component informs the `Platform` of the successful refund.
  - The `Platform` notifies the `Merchant API`, which then sends a refund confirmation to the `Notifications` service.

- The `Notifications` service notifies the customer of the refund success.

- **Refund Failure:**
  - If the refund fails, the `Payment Processing` component informs the `Platform` of the failure.
  - The `Platform` notifies the `Merchant API`, which then notifies the customer of the refund failure.

Sequence Diagram for Fraud Detection



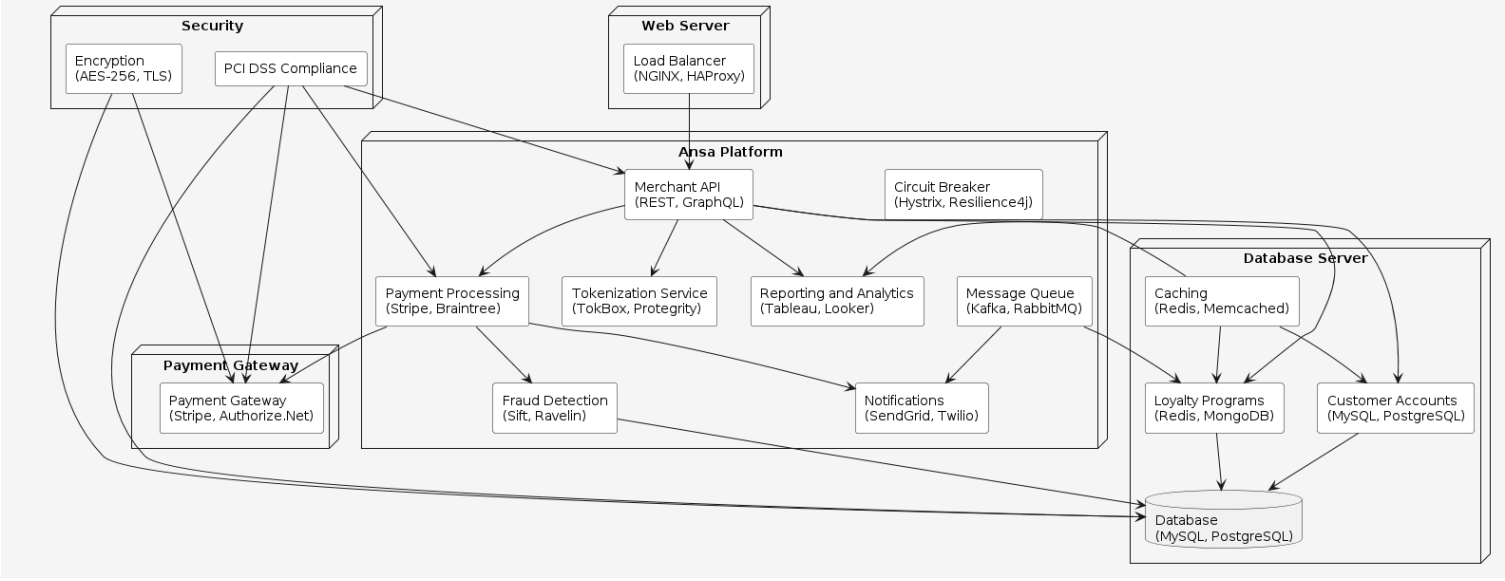
Explanation:

1. **Customer Initiates Payment:**
   - The customer initiates a payment, sending the request to the `Merchant API`.
2. **Request Forwarding:**
   - The `Merchant API` forwards the payment request to the `Ansa Platform`.
3. **Fraud Detection Initiation:**
   - The `Ansa Platform` requests the `Fraud Detection Service` (FDS) to perform a fraud check.
4. **Query Fraud Database:**

- The `FDS` queries the `Fraud Database` (FDB) for any previous fraudulent activity related to the transaction.

5. **Machine Learning Analysis:**
   - The `FDS` sends transaction details to the `Machine Learning Model` (ML) for analysis.
   - The `ML` returns the analysis result to the `FDS`.
6. **IP Geolocation Check:**
   - The `FDS` requests the `IP Geolocation Service` (IPGeo) to check the IP location of the transaction.
   - The `IPGeo` returns the geolocation result to the `FDS`.
7. **Velocity Check:**
   - The `FDS` performs a velocity check by interacting with the `Velocity Check Service` (VCS).
   - The `VCS` returns the velocity check result to the `FDS`.
8. **Retrieve Transaction Data:**
   - The `FDS` retrieves transaction data from the `Database` (DB).
   - The `DB` returns the transaction data to the `FDS`.
9. **Exploratory Data Analysis:**
   - The `FDS` performs exploratory data analysis on the transaction data.
   - It derives transaction insights from the data.
10. **Detailed Checks:**
    - The `FDS` identifies multiple card swipes, amount reversals, and analyzes transaction amount insights if applicable.
11. **Machine Learning Model Building:**
    - The `FDS` builds and evaluates a machine learning model based on the transaction data.
12. **Determine Fraud Status:**
    - The `FDS` combines the results from all checks to determine the fraud status.
13. **Return Fraud Status:**
    - The `FDS` returns the fraud status to the `Ansa Platform`.
    - The `Ansa Platform` notifies the `Merchant API` of the fraud status.

Deployment Diagram:



Explanation of Deployment Diagram:

1. **Web Server:**
   - `Load Balancer (NGINX, HAProxy)`: Distributes incoming requests across multiple servers.
2. **Ansa Platform:**
   - `Merchant API (REST, GraphQL)`: Handles requests from merchants.
   - `Payment Processing (Stripe, Braintree)`: Manages payment transactions.
   - `Fraud Detection (Sift, Ravelin)`: Performs fraud checks.
   - `Notifications (SendGrid, Twilio)`: Manages notifications.
   - `Reporting and Analytics (Tableau, Looker)`: Generates reports and analytics.
   - `Circuit Breaker (Hystrix, Resilience4j)`: Manages service failures.
   - `Tokenization Service (TokBox, Protegrity)`: Handles card tokenization.
   - `Message Queue (Kafka, RabbitMQ)`: Manages asynchronous communication.
3. **Database Server:**
   - `Database (MySQL, PostgreSQL)`: Main database.
   - `Customer Accounts (MySQL, PostgreSQL)`: Manages customer account data.
   - `Loyalty Programs (Redis, MongoDB)`: Manages loyalty program data.
   - `Caching (Redis, Memcached)`: Caches frequently accessed data.
4. **Payment Gateway:**
   - `Payment Gateway (Stripe, Authorize.Net)`: Processes payments.
5. **Security:**
   - `PCI DSS Compliance`: Ensures PCI DSS compliance.
   - `Encryption (AES-256, TLS)`: Encrypts sensitive data.

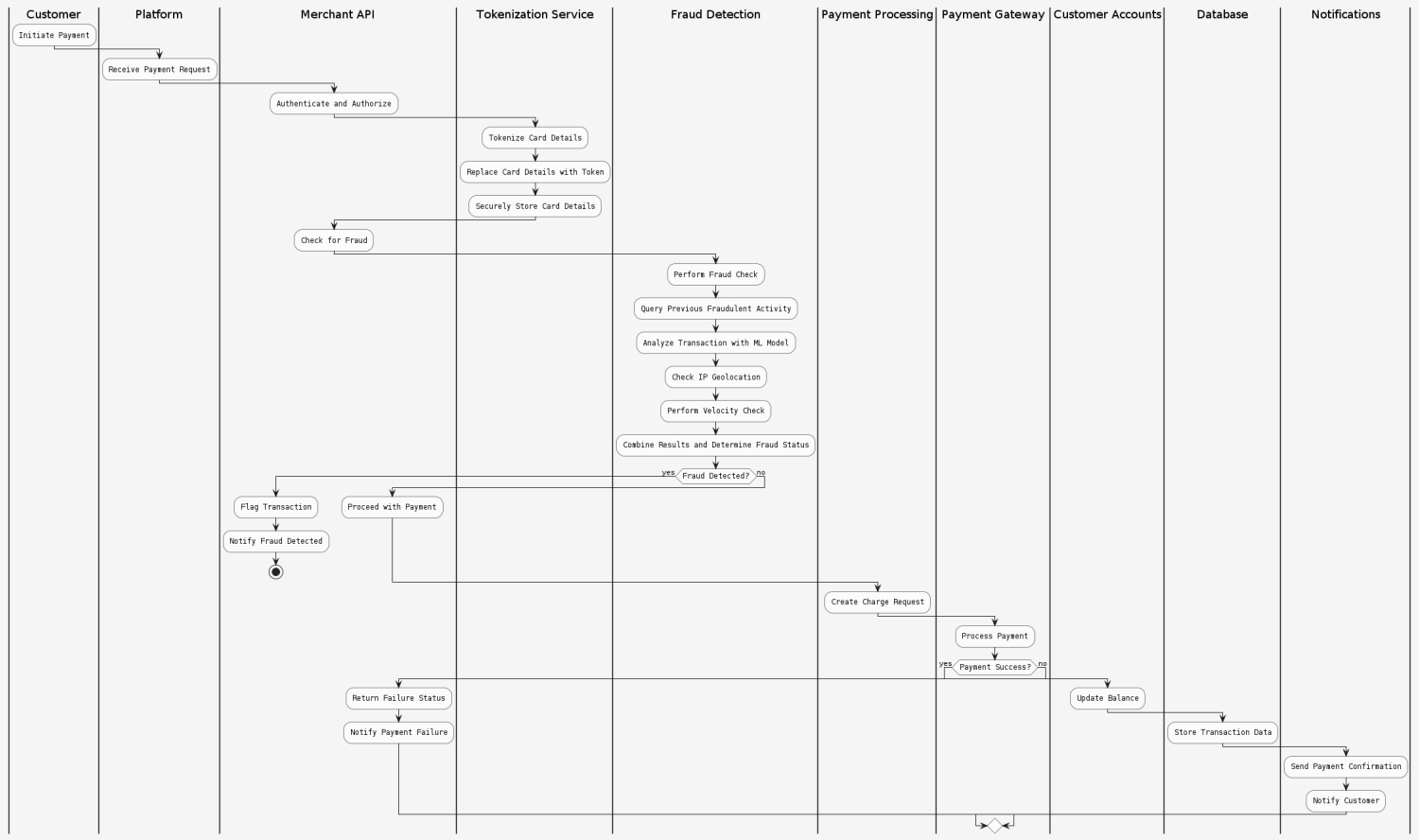
Interaction:

- The `Load Balancer` forwards incoming requests to the `Merchant API`.
- The `Merchant API` interacts with various services such as `Payment Processing`, `Fraud Detection`, and `Tokenization Service`.

- The `Payment Processing` component sends payment requests to the `Payment Gateway`.
- The `Customer Accounts` and `Loyalty Programs` components store and manage customer-related data in the `Database`.
- The `Fraud Detection` component queries the `Database` and performs checks.
- The `Notifications` component sends notifications.
- The `Reporting and Analytics` component logs transactions for reporting.
- The `Message Queue` manages asynchronous communication between services.
- The `Security` node ensures data encryption and compliance with PCI DSS standards.

This deployment diagram provides a high-level view of how the components are distributed across different servers and their interactions, adding a physical perspective to the system architecture.

Activity Diagram: Processing a Payment



Explanation of Activity Diagram:

1. **Customer**:
   - Initiates the payment.
2. **Platform (Ansa)**:
   - Receives the payment request.
3. **Merchant API**:
   - Authenticates and authorizes the request.

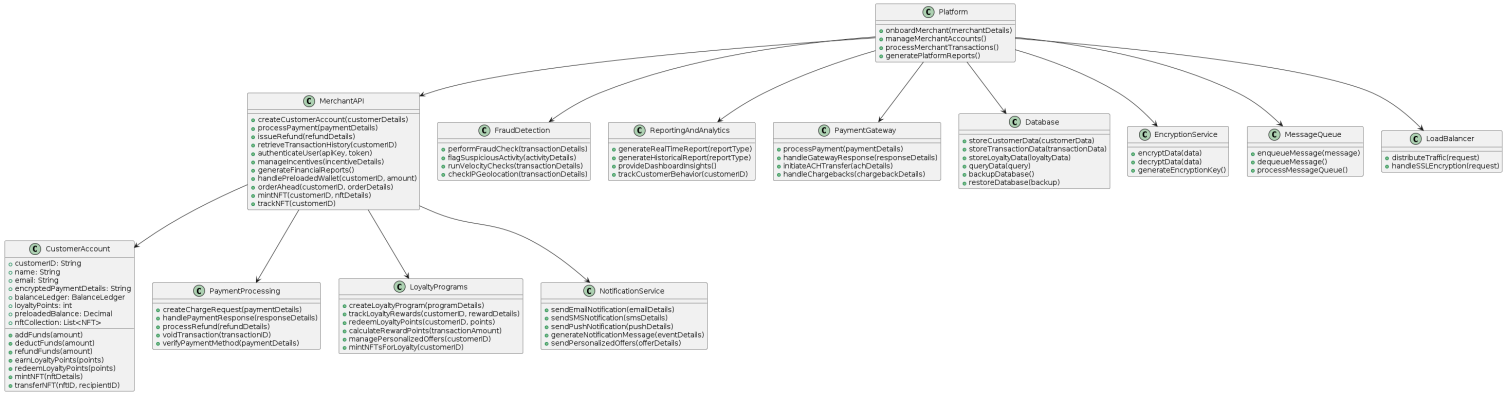
- Interacts with the Tokenization Service and Fraud Detection Service.

4. **Tokenization Service**:
   - Tokenizes card details and securely stores them.
   - Returns the tokenized details to the Merchant API.
5. **Fraud Detection**:
   - Performs a fraud check.
   - Queries previous fraudulent activity.
   - Analyzes the transaction with a machine learning model.
   - Checks IP geolocation and performs a velocity check.
   - Combines results to determine the fraud status.
6. **Merchant API**:
   - Checks the fraud status and decides whether to proceed with the payment or flag the transaction for fraud.
7. **Payment Processing**:
   - Creates a charge request.
8. **Payment Gateway**:
   - Processes the payment.
   - Returns the payment response to the Payment Processing component.
9. **Customer Accounts**:
   - Updates the customer balance if the payment is successful.
   - Stores transaction data in the database.
10. **Notifications**:
    - Sends a payment confirmation to the customer if the payment is successful.
    - Notifies the customer if the payment fails.

This activity diagram provides a detailed view of the workflow for processing a payment, highlighting the interactions between various components and decision points.

Class Diagram:

This class diagram includes detailed attributes and methods for each class, reflecting the complexity of an enterprise-level application. For example, the `CustomerAccount` class now includes methods for adding, deducting, and refunding funds, as well as maintaining loyalty points. The `PaymentProcessing` class includes methods for verifying payment methods and handling chargebacks.



Explanation

1. **Platform**: Central class for Ansa as a PaaS, managing merchant accounts, processing transactions, and generating platform-level reports.
2. **MerchantAPI**: Interface for merchants to interact with the platform, including customer account management, payment processing, refunds, and other merchant-specific operations.
3. **CustomerAccount**: Represents customer details, including encrypted payment information, balance ledger, loyalty points, and NFT collection.
4. **PaymentProcessing**: Handles the creation of charge requests, payment responses, refunds, transaction voids, and payment method verification.
5. **LoyaltyPrograms**: Manages loyalty programs, tracking rewards, redeeming points, calculating reward points, managing personalized offers, and minting NFTs for loyalty.
6. **FraudDetection**: Performs fraud checks, flags suspicious activity, runs velocity checks, and checks IP geolocation.
7. **NotificationService**: Sends various types of notifications (email, SMS, push) and personalized offers.
8. **ReportingAndAnalytics**: Generates real-time and historical reports, provides dashboard insights, and tracks customer behavior.
9. **PaymentGateway**: Processes payments, handles gateway responses, initiates ACH transfers, and manages chargebacks.
10. **Database**: Stores customer data, transaction data, loyalty data, and handles database queries, backups, and restores.
11. **EncryptionService**: Encrypts and decrypts data, generates encryption keys.
12. **MessageQueue**: Manages message queues for processing messages.
13. **LoadBalancer**: Distributes traffic and handles SSL encryption.

By incorporating the Platform class, this diagram reflects the broader role of Ansa as a PaaS, coordinating interactions between the customer, merchants, and various services.

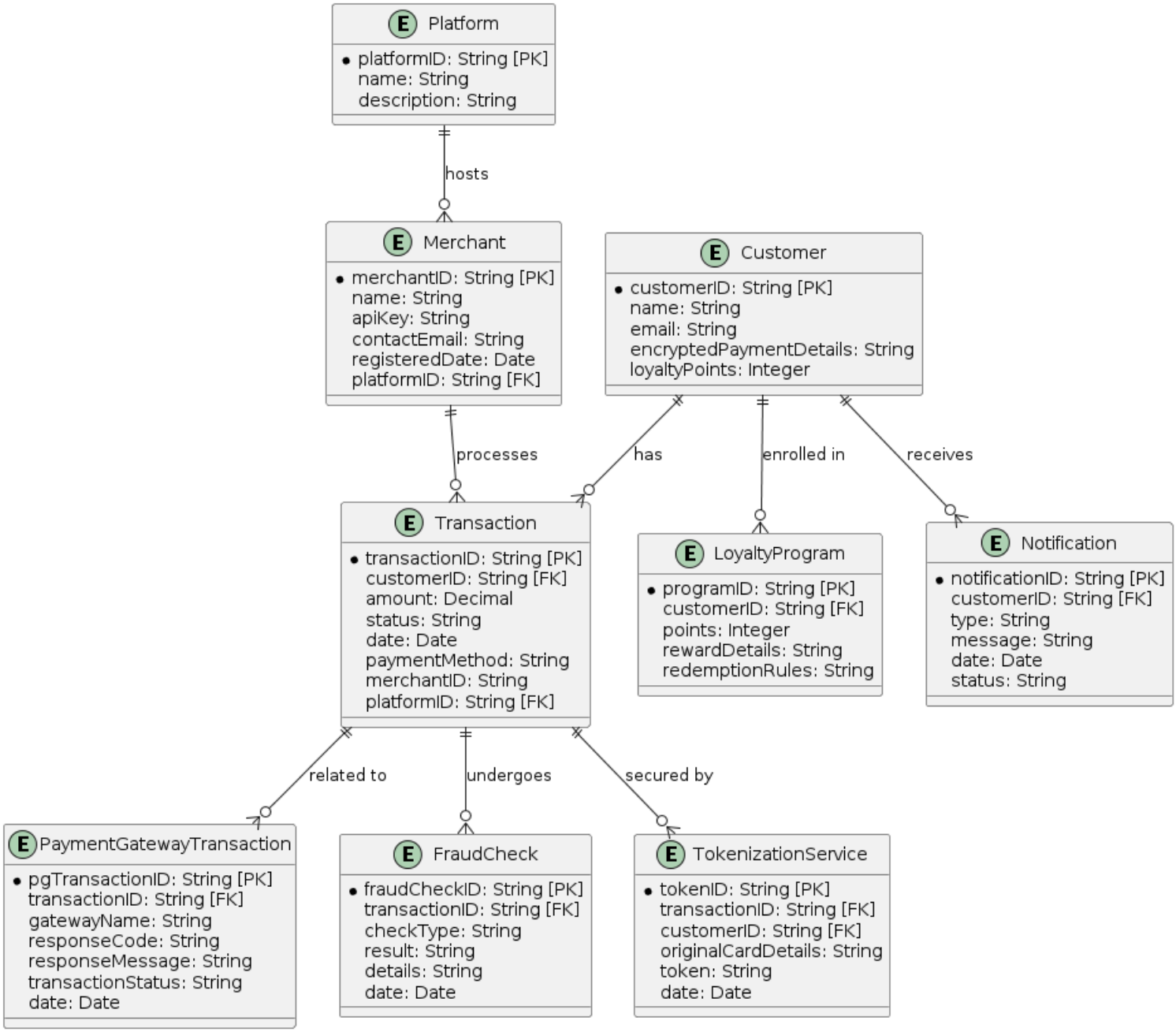
Summary

This class diagram represents a comprehensive and modular system where each class has specific responsibilities and interacts with other classes to ensure the seamless processing of customer transactions, managing loyalty programs, detecting fraud, sending notifications, generating reports, and

handling secure data storage and retrieval. The addition of `TokenizationService`, `ShardingService`, and `ConsensusMechanism` enhances security and scalability, addressing ledger scaling challenges and ensuring secure handling of sensitive information.

ERD (Entity-Relationship-Diagram):

The Entity-Relationship Diagram (ERD) represents the database schema, showing entities such as `Customer`, `Transaction`, `LoyaltyProgram`, and `Notification`. The relationships between these entities, such as a customer having multiple transactions, are also depicted.
The ERD has been expanded to include more entities such as `Merchant`, `PaymentGatewayTransaction`, and `FraudCheck`. Each entity has detailed attributes, and the relationships between entities are clearly defined. For instance, the `transaction` entity now relates to `pgTransaction` and `fraudCheck`, indicating the details of gateway transactions and fraud checks conducted on each transaction.



Explanation:

- **Customer**: Represents the customers using the platform, storing their ID, name, email, encrypted payment details, and loyalty points.
- **Transaction**: Captures transaction details including ID, customer ID, amount, status, date, payment method, and merchant ID.
- **LoyaltyProgram**: Manages loyalty programs, linking to customers and detailing points, rewards, and redemption rules.
- **Notification**: Stores notifications sent to customers, including type, message, date, and status.
- **Merchant**: Represents merchants on the platform, storing ID, name, API key, contact email, and registration date.
- **PaymentGatewayTransaction**: Records payment gateway transaction details, linking to transactions and including gateway name, response code, message, status, and date.
- **FraudCheck**: Details fraud checks performed on transactions, storing ID, transaction ID, check type, result, details, and date.
- **TokenizationService**: Manages the tokenization of card details, linking to transactions and customers, storing original card details, token, and date.
- **Platform**: Represents Ansa as a PaaS, storing ID, name, API key, and contact email.

Relationships:

- **Customer to Transaction**: A customer can have multiple transactions.
- **Customer to LoyaltyProgram**: A customer can be enrolled in multiple loyalty programs.
- **Customer to Notification**: A customer can receive multiple notifications.
- **Merchant to Transaction**: A merchant can process multiple transactions.
- **Transaction to PaymentGatewayTransaction**: Each transaction can be related to multiple payment gateway transactions.
- **Transaction to FraudCheck**: Each transaction can undergo multiple fraud checks.
- **Transaction to TokenizationService**: Each transaction can be secured by multiple tokenization services.
- **Platform to Merchant**: The platform hosts multiple merchants.

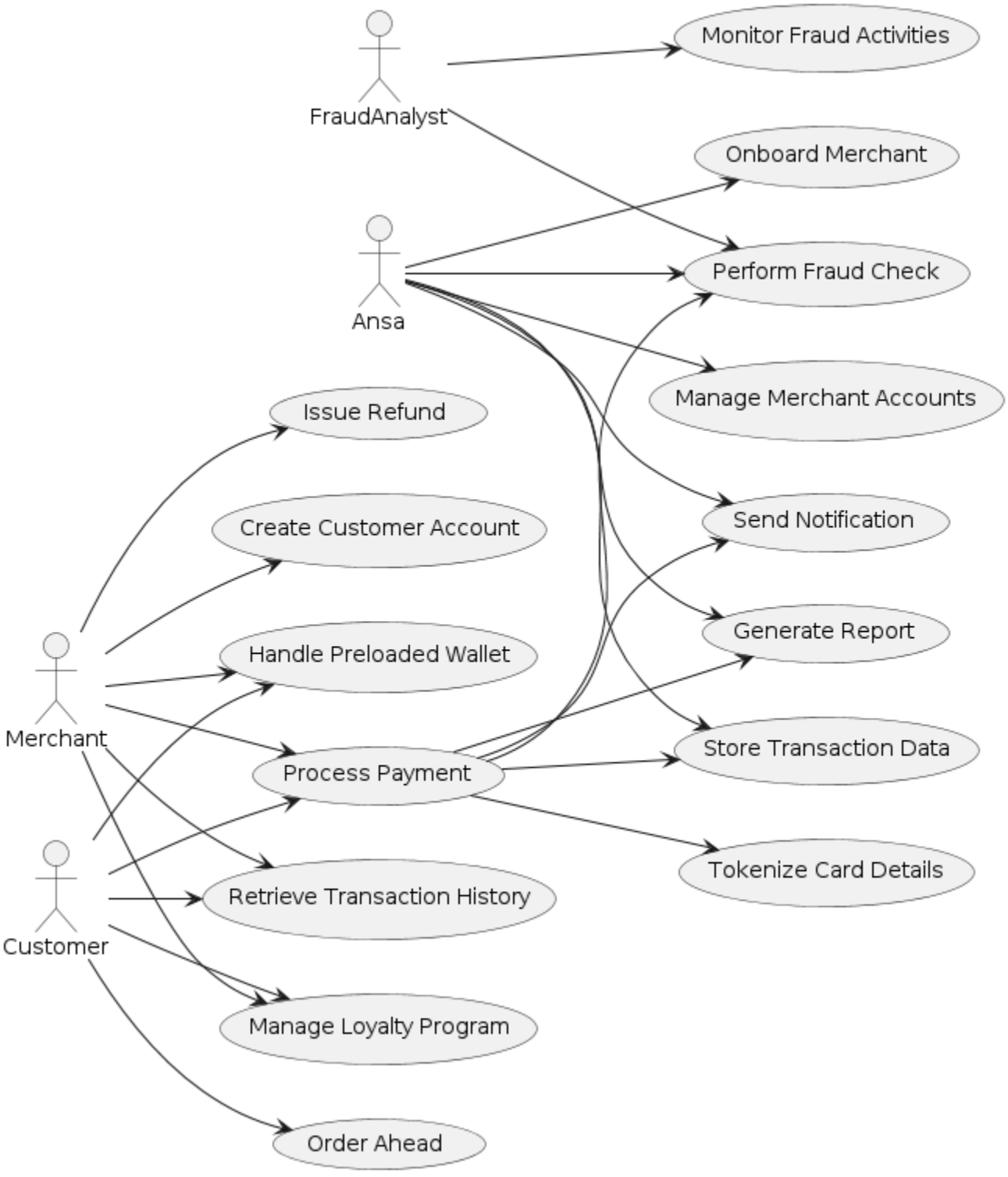
Use Case Diagram:

The use case diagram visualizes the interactions between users (actors) and the system. It shows the different use cases, such as creating a customer account, processing payments, and generating reports, and the actors involved, including merchants, customers, and fraud analysts.

The diagram includes detailed interactions such as:

- **Merchants** can create customer accounts, process payments, issue refunds, retrieve transaction history, manage loyalty programs, generate reports, manage merchant accounts, and handle preloaded wallets.
- **Customers** can process payments, retrieve transaction history, manage loyalty programs, handle preloaded wallets, and order ahead.
- **Fraud Analysts** can perform fraud checks and monitor fraud activities.

The interactions between these use cases show how various actors engage with the system, providing a comprehensive view of their roles and responsibilities.

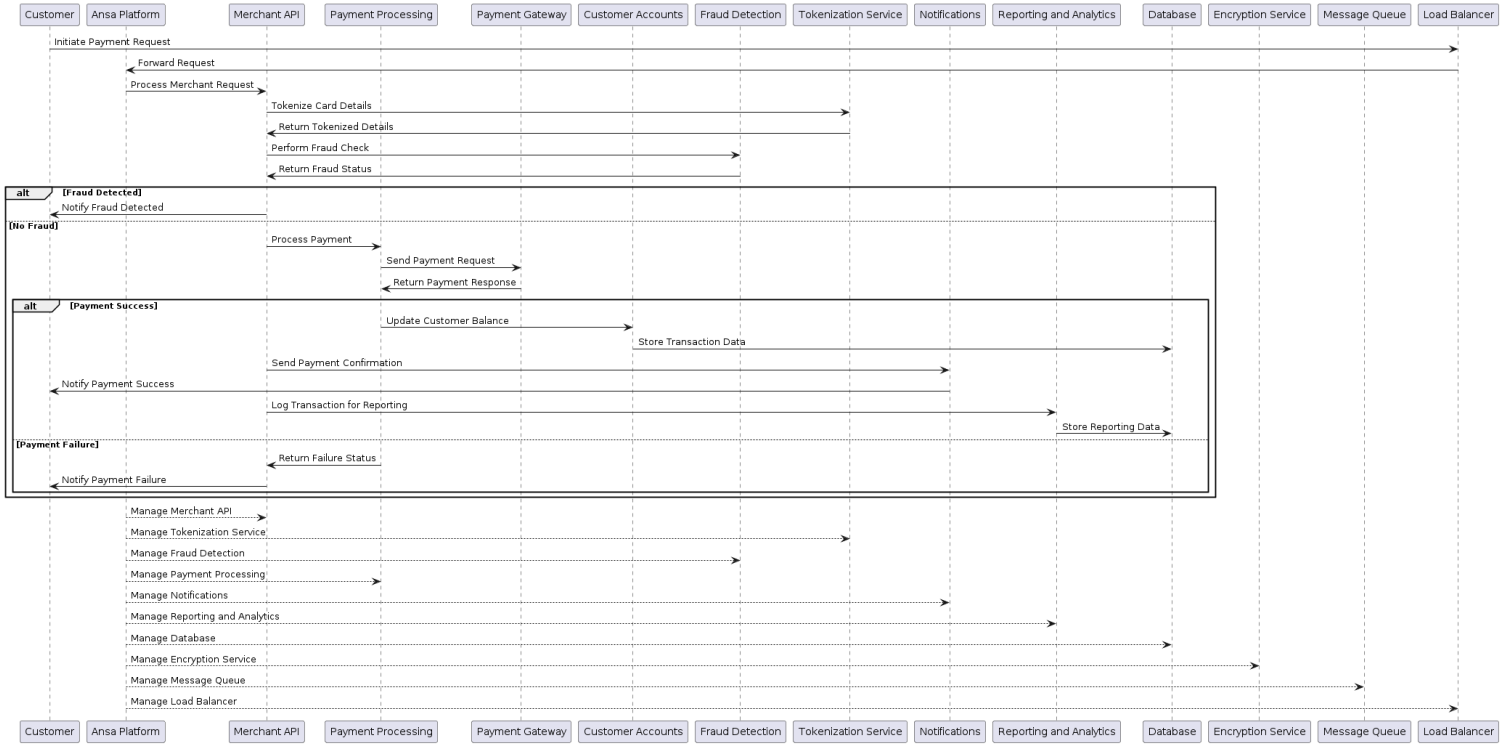


Explanation:

- **Actors:**
  - **Ansa**: Represents the platform providing services to merchants, customers, and fraud analysts.
  - **Merchant**: Merchants who use the platform to manage their customers and transactions.
  - **Customer**: Customers who interact with the merchants and use the services provided by the platform.
  - **FraudAnalyst**: Analysts who monitor and check for fraudulent activities on the platform.
- **Use Cases:**
  - **Onboard Merchant**: Ansa onboards new merchants to the platform.
  - **Manage Merchant Accounts**: Ansa manages accounts for merchants.
  - **Perform Fraud Check**: Ansa performs fraud checks on transactions.
  - **Store Transaction Data**: Ansa stores transaction data.
  - **Send Notification**: Ansa sends notifications to customers or merchants.
  - **Generate Report**: Ansa generates reports for merchants and platform administrators.
  - **Create Customer Account**: Merchants create customer accounts.
  - **Process Payment**: Merchants process payments for their customers.
  - **Issue Refund**: Merchants issue refunds for transactions.
  - **Retrieve Transaction History**: Merchants and customers retrieve transaction history.
  - **Manage Loyalty Program**: Merchants manage loyalty programs for their customers.
  - **Handle Preloaded Wallet**: Merchants handle preloaded wallet amounts for customers.
  - **Perform Fraud Check**: Fraud analysts perform fraud checks.
  - **Monitor Fraud Activities**: Fraud analysts monitor fraud activities on the platform.
  - **Tokenize Card Details**: Ansa tokenizes card details during the payment process.
  - **Order Ahead**: Customers order ahead using the platform.

This updated diagram shows the interactions between various actors and the platform, demonstrating how Ansa facilitates the operations of merchants, customers, and fraud analysts.

Component Interaction Diagram:



Explanation:

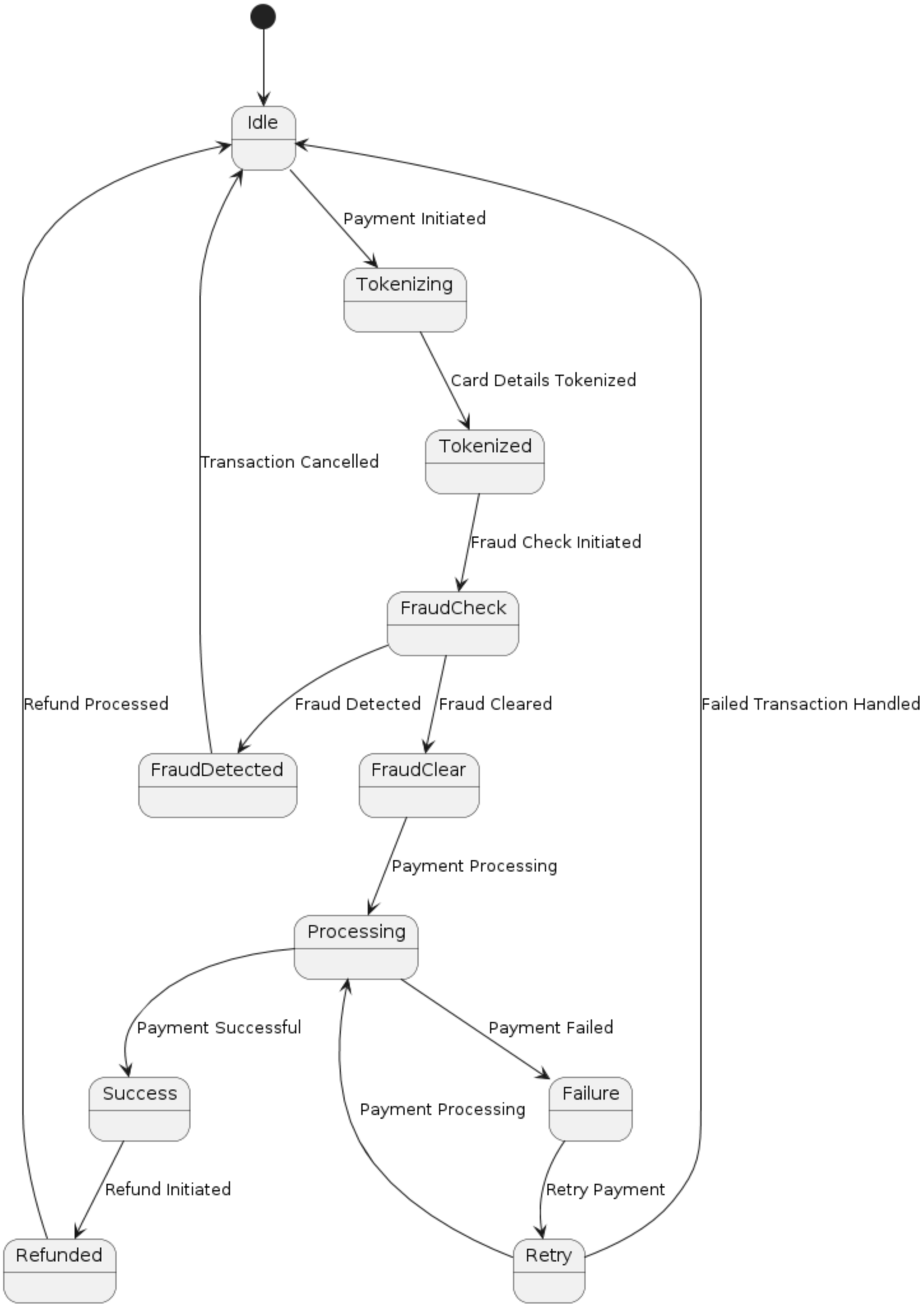
1. **Customer Initiates Payment:**
   - The customer initiates a payment request, which is sent to the `Load Balancer`.
2. **Request Forwarding:**
   - The `Load Balancer` forwards the request to the `Merchant API`.
3. **Tokenization:**
   - The `Merchant API` sends card details to the `Tokenization Service`, which replaces the card details with a token.
   - The `Tokenization Service` returns the tokenized details to the `Merchant API`.
4. **Fraud Check:**
   - The `Merchant API` requests the `Fraud Detection` service to perform a fraud check.
   - The `Fraud Detection` service returns the fraud status to the `Merchant API`.
5. **Fraud Detected:**
   - If fraud is detected, the `Merchant API` notifies the customer of the fraud detection.
6. **No Fraud Detected:**
   - If no fraud is detected, the `Merchant API` proceeds with the payment by sending the request to `Payment Processing`.
7. **Payment Processing:**
   - The `Payment Processing` component sends the payment request to the `Payment Gateway`.
   - The `Payment Gateway` processes the payment and returns the response to `Payment Processing`.
8. **Payment Success:**
   - If the payment is successful, `Payment Processing` updates the customer's balance in `Customer Accounts`.

- `Customer Accounts` stores the transaction data in the `Database`.
- The `Merchant API` sends a payment confirmation to the `Notifications` service.
- The `Notifications` service notifies the customer of the payment success.
- The `Merchant API` logs the transaction for reporting in the `Reporting and Analytics` component.
- `Reporting and Analytics` stores the reporting data in the `Database`.

9. **Payment Failure:**
   - If the payment fails, `Payment Processing` returns the failure status to the `Merchant API`.
   - The `Merchant API` notifies the customer of the payment failure.
10. **Ansa Platform:**
    - The `Ansa Platform` manages and oversees the interactions of various services, ensuring the smooth operation of the `Merchant API`, `Tokenization Service`, `Fraud Detection`, `Payment Processing`, `Notifications`, `Reporting and Analytics`, and `Database`.

State Diagram for Transaction:

This state diagram illustrates the various states a transaction can go through, from initiation to processing, success, failure, and potential refund. It shows the transitions between these states based on different events.


Idle
Payment Initiated
Tokenizing
Card Details Tokenized
Tokenized
Transaction Cancelled
Fraud Check Initiated
FraudCheck
Refund Processed
Fraud Detected
Fraud Cleared
Failed Transaction Handled
FraudDetected
FraudClear
Payment Processing
Processing
Payment Successful
Payment Failed
Success
Failure
Payment Processing
Refund Initiated
Retry Payment
Refunded
Retry

Explanation:

1. **Idle:**
   - The initial state where no transaction is being processed.
2. **Tokenizing:**
   - When a payment is initiated, the system transitions to the Tokenizing state.
   - The card details are sent to the Tokenization Service for tokenization.
3. **Tokenized:**
   - Once the card details are tokenized, the system transitions to the Tokenized state.
4. **FraudCheck:**
   - The tokenized details are sent for a fraud check.
   - The system transitions to the FraudCheck state.
5. **FraudDetected:**
   - If fraud is detected during the fraud check, the system transitions to the FraudDetected state.
   - The transaction is canceled, and the system returns to the Idle state.
6. **FraudClear:**
   - If no fraud is detected, the system transitions to the FraudClear state, indicating the transaction is clear of fraud.
7. **Processing:**
   - The payment processing begins, transitioning the system to the Processing state.
8. **Success:**
   - If the payment is successfully processed, the system transitions to the Success state.
   - The transaction can be refunded, transitioning to the Refunded state, and then back to Idle once processed.
9. **Failure:**
   - If the payment fails, the system transitions to the Failure state.
   - The transaction can be retried, transitioning to the Retry state, and then back to Processing.
10. **Refunded:**
    - When a refund is initiated, the system transitions to the Refunded state.
    - Once the refund is processed, the system returns to Idle.
11. **Retry:**
    - If a failed transaction is retried, the system transitions to the Retry state and then back to Processing.
