Development Decisions

1. How will you read messages from the queue?

Messages are read from the AWS SQS queue using the Boto3 library. The `receive_message` method of the SQS client is used to fetch messages from the queue.

2. What type of data structures should be used?

The messages are read into Python dictionaries, which are then processed and inserted into the PostgreSQL database. The dictionaries allow for easy manipulation and access to the data fields.

3. How will you mask the PII data so that duplicate values can be identified?

Personally Identifiable Information (PII) data is masked using SHA-256 hashing with a salt ("fetch_rewards"). This ensures that duplicate values produce the same hash, allowing for identification of duplicates without revealing the actual PII data.

4. What will be your strategy for connecting and writing to Postgres?

A connection to the PostgreSQL database is established using the `psycopg2` library. The processed data is then inserted into the `user_logins` table using SQL INSERT statements.

5. Where and how will your application run?

The application runs inside a Docker container. Docker Compose is used to manage the services, which include LocalStack (for the SQS queue) and PostgreSQL (for the database).

Production Deployment and Considerations

1. **How would you deploy this application in production?**

In production, the application would be deployed using container orchestration tools like Kubernetes or Docker Swarm. This would involve setting up the following:

- Kubernetes Cluster: For managing and scaling the application.

- AWS SQS: For message queuing.

- AWS RDS: For a managed PostgreSQL database.

- CI/CD Pipeline: For automated testing and deployment (using tools like Jenkins or GitHub Actions).

2. **What other components would you want to add to make this production ready?**

To make the application production-ready, the following components would be added:

- Logging and Monitoring: Using tools like Prometheus, Grafana, and ELK Stack to monitor application performance and log errors.

- Secrets Management: Using AWS Secrets Manager or HashiCorp Vault to manage database credentials and other sensitive information.

- Auto-scaling: Setting up auto-scaling policies to handle increased load.

- Load Balancer: To distribute incoming traffic and ensure high availability.

- Health Checks: To automatically restart unhealthy containers.

3. **How can this application scale with a growing dataset?**

The application can scale horizontally by adding more instances of the application container. The use of a managed database service (like AWS RDS) allows for easier scaling of the database. Additionally, the SQS queue can handle a large number of messages, ensuring that the application can scale to handle a growing dataset.

4. How can PII be recovered later on?

To recover PII, a secure key management system (KMS) would be required. The original PII data would be encrypted and stored securely, with the decryption keys managed by the KMS. This ensures that only authorized personnel can access the PII data, adhering to data protection regulations.

5. What are the assumptions you made?

- PII Masking: It is assumed that hashing with a salt is sufficient for masking PII and identifying duplicates.

- LocalStack and PostgreSQL: It is assumed that LocalStack and PostgreSQL running in Docker containers accurately simulate the production environment.

- Message Format: It is assumed that the messages in the SQS queue follow a specific JSON format.

- Security: Basic security measures are assumed to be in place, but additional measures would be needed for a production environment.