Decisions

1. **Masking the PII data:**

   To achieve this, I opted for deterministic encryption among the three familiar techniques, alongside hashing and Base64 encoding.

   I did not go with hashing because it's not reversible and though there's very little chance of collisions, collisions are still a possibility. The reason why I did not go with Base64 encoding is that it's easily revertable, anybody who has access to the data could easily get their hands on sensitive information.

   However deterministic encryption hits the sweet spot, it's only reversible if there's a key and a one-to-one relationship between fields and encrypted values without collisions and identifying duplicate values.

2. **Choices made during table creation:**

   a. *I chose user_id and create_date as compound primary key*
   This is because a user may have multiple logins, and choosing the user_id as the only primary key or not having a primary key might lead to problems

   b. *Used string type for app_version*
   As the app_version is the form '0.3.4', I could not figure out a way to insert it as integers without loss of information or splitting it into three different columns (major, minor, and patch version)

   c. *Used the timestamp for type create_date*
   Because of Choice A, if a user has multiple logins during the same day, it'll still count as a duplicate key, so this timestamp type to the seconds level will avoid it

   d. *Used the timestamp of when the data was extracted from SQS*
   I used the timestamp of when the message was extracted from the SQS service

3. **Choices I made during writing to Postgres:**

   a. *Retrieving all messages till the queue is empty*
   Here, I first emptied the queue and stored the information someplace and then I inserted it, in some production cases this might not be a very ideal

   b. *Not going for a staging area*
   Because of the first choice I made, I opted not to implement a staging area, such as a JSON file, this would not be my choice in production-level code, as I might be interested in some clean-up scripts.

Questions

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

   Various factors would influence my decision in this case, like the frequency of the messages, the infrastructure available, and the cost ceiling.
   For production-level code, I would not wait for the queue to get empty and insert as if the logins are frequent the queue might never end. A time-based or number of messages-based queueing-insertion cycle can be implemented. The choice here again is very much dependent on why the data is being inserted into the Postgres DB.

   a. *Serverless services*
      This is the first approach that came to my head, we can make use of the Lambda service on AWS, which keeps reading these messages at a set interval and another Lambda reads and inserts these messages and inserts in the Postgres DB. Here the staging area might not be effective, as concurrency might become an issue.

   b. *Container Orchestraion*
      This is also something that we could do, write the code containerize it using Docker, and orchestrate using Kubernetes. In this case, we can use a staging area, as it is containerized and the staging area also scales accordingly

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

   a. *Secret management*
      As you might have noticed the encryption keys and other sensitive information lying around in the code. This should be avoided at the prod level by using Secret Management services in AWS or GCP

   b. *Backup and Disaster recovery*
      This is pretty much self-explanatory having a replica of your DB that gets updated at a certain interval is always not a bad idea

   c. *Monitoring and Logging*
      Monitoring tools like AWS CloudWatch to monitor the health, performance metrics, and resource utilization of your Kubernetes clusters or Lambda functions. Implementing centralized logging using tools like Elasticsearch, to aggregate and analyze logs from your application.

   d. *More Testing*
      More testing is always a good practice for a production-ready code ensuring the robustness of it

e. *Better Error handling*
   While receiving messages, all the errors are handled by one except a statement, which should be avoided in prod code, better error handling should be written

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

   a. Lambda being a serverless service can scale by itself based on the demand we have
   b. We can also provision to horizontally scale the container service if we choose that method.

4. **How can PII be recovered later on?**

   We'll be storing the key for encryption in Secrets Manager, which can be used to decrypt the information when needed. The secret key for the demo project is right in the script, which is a bad practice

5. **What are the assumptions you made?**

   a. Storing the app_version as varchar as I thought we could split the columns and have it as int type upon need, rather than having three different columns (major, minor, and patch version)

   b. Using the timestamp of when the message was extracted from the SQS service as the time of login, as there's no other time available