**Solution Overview**

The ETL DAG follows a simple approach to get, clean, and store the data. The fetch\_data task gets transaction data from the given API using a POST request. It uses environment variables to keep API keys safe, but the .env file is not encrypted when the project is in GitHub. If this project were to be used in production, I would store in a secrets manager and save in secure place. This would keep API keys and database passwords safe from leaks. The transform\_data task cleans the data by removing duplicates, filling in missing values, and making sure all transactions are valid. It also adds categories based on transaction amounts and calculates total spending per customer. The last step, load\_data, saves the cleaned data into a MySQL database and creates the transactions table if needed.

To make the process more efficient, Used xcom\_push and xcom\_pull to pass data between tasks without saving extra files, also checked and fixed timestamps so the data stays consistent. One limitation is that we set a fixed date range when fetching data, which might not work well if more flexibility is needed. Another downside is that the table is dropped and recreated each time, which might not be ideal for handling new data over time. This DAG works well for batch processing, but it could be improved for real-time or incremental updates.

**Theoretical questions**

1. How do you scale this solution to 10x or 100x, taking into account different potential latencies needed?

If we need to handle 10x or 100x more data, we can make API calls in parallel instead of one by one to speed things up. We can also break the data into smaller chunks so we don’t overload the system. If the data gets too big, we could use a tool like Apache Spark to process it faster. When storing the data, we should use bulk inserts instead of inserting rows one at a time. If the database slows down, we can split it into smaller parts (sharding) or move to a cloud-based system like BigQuery or Redshift, which are built for big data.

1. How to handle changes within the source data, assuming some transactions would be potentially modified or backdated.

If transactions can change after they are first recorded (for example, a payment gets cancelled or added later), we need to update our records properly. One way is to add an updated at column, so we know which records have changed and only update those. Another way is to use an upsert (update or insert), so if the record exists, we update it, and if not, we add it. If some transactions get reversed, we can add a column that shows their status, like "active," "reversed," or "pending," so we don’t delete anything but still keep track of changes.

1. Assuming the company is getting this data for the first time and has never had it before in a usable format for analytics or automations - what would be the single most important thing to build from it or take from it to deliver value?

If the company is getting this data for the first time, we need to load all the historical data while making sure it's accurate. We should create a database setup that makes it easy to use the data later, like adding indexes to speed up searches. It’s also important to check for mistakes, like missing values or duplicate records. If the company expects to collect more data over time, we should store both raw and cleaned data separately, maybe in a data warehouse or data lake, so it's easier to use in the future.