Instructions :

Please spend upto 1 hour on the interview, not more. The coding is evaluated in python.

Make any assumptions needed to solve the problem but note them down.

Question 1:

Background:
Your task is to develop a RESTful API for a product inventory system. This system interfaces with a relational database that stores product information including name, price, and stock levels. The API will be used frequently, so performance and efficiency are key considerations.

Task:
Design and describe a RESTful API that provides endpoints for the following functionalities:

1. Retrieving Product Information:
    a. An endpoint to retrieve details of a single product by its ID.
    b. An endpoint to retrieve a list of all products, with pagination support.
2. Updating Product Information:
    a. An endpoint to update the price and stock level of a product.
3. Searching for Products:
    a. An endpoint to search for products based on name or price range.

Requirements:
1. Database Interactions:
    a. Detail how your API will interact with the database to perform these operations. Include considerations for optimizing database queries.
2. Caching Strategy:
    a. Propose a caching strategy to enhance the performance of the API, especially for read-heavy operations. Explain how you would implement caching and how you would invalidate the cache when data changes.
3. Error Handling:
    a. Your API should handle and return appropriate responses for common error scenarios such as product not found, invalid input data, and server errors.
4. Data Consistency:
    a. Discuss how you would ensure data consistency between the cache and the database, particularly after update operations.
5. Scalability:
    a. Briefly describe how your API design would scale to handle a large number of requests.

Assumptions:

- You can choose any relational database of your preference (e.g., PostgreSQL, MySQL).
- Assume the database schema for the product table is already defined with fields like **id**, **name**, **price**, and **stock_level**.
- You can choose any backend of your choice (for example Flask, FastAPI, etc)

Bonus Question:

- **Rate Limiting:**
  - Propose a method to implement rate limiting on your API to prevent abuse. Discuss how you would decide the limits and how they would be enforced.

Question 2 :

Background:
You are given access to a large dataset from the Google Ads API. Due to the vast amount of data, querying this API for a range of dates sequentially can be time-consuming. To optimize this process, you need to implement a parallel data retrieval system in Python.

Task:
Write a Python script that uses parallel processing to query the Google Ads API for a specified range of dates. Each query should retrieve data for a single date. Assume that you have a function fetch_data(date) which takes a date as an argument and returns the data from the Google Ads API for that date.

Requirements:

Input:

The script should take two dates as input: start_date and end_date.
These dates define the range for which the data needs to be fetched.

Parallel Processing:

Implement parallel processing to query the API for each date in the range.
You may use Python's concurrent.futures, threading or asynci module or any other suitable library for parallel processing.

Saving:
Aggregate the data fetched for each date and save it to a csv. The overall folder for the data is one of the inputs

Error Handling:
Ensure that your script can handle potential errors or exceptions that might occur during API calls.

- Timeout Errors
- RateLimiting Errors
- OAuth Error

Performance Metrics:
Optionally, include code to measure and print the time taken to fetch and aggregate the data.

Assumptions:

- The fetch_data(date) function is already implemented and handles the actual API calls.
- Date inputs can be in any standard format, but they should be consistent throughout the script.

Bonus question :

Backfilling Historical Data:

The script should automatically calculate the date range for the last two years from the current date. Fetch and aggregate this historical data using the parallel processing approach.

Daily Updates:

When run daily, the script should determine which days' data is missing.
Implement a mechanism to identify missing dates and fetch data for those dates.

Data Storage Consideration:

Provide a strategy or pseudo-code for how the fetched data would be
stored, ensuring there are no duplicates and that missing data is correctly identified and fetched.