Python Engineer Technical Assessment

Time limit:

- Implementation: **2 hours**
- Submission (push to GitHub): **15 minutes**

Overview

The goal of this exercise is to evaluate your ability to:

- **Design scalable applications** that process large volumes of data.
- Use **asynchronous programming** and **parallel processing** effectively.
- Work with **message brokers**, **queues**, and **websockets**.
- Implement robust **serialization/deserialization** and **error handling**.
- Demonstrate **clean code practices**, **architectural design choices**, and **performance awareness**.

⚠️ Completing both parts is highly recommended, but we value **well-thought-out architecture** and **clean, maintainable code** over simply solving everything in a trivial way.

Your final solution should be presented as a **GitHub repository** with a clear **README.md** that includes:

- Project overview
- Set up and run instructions
- Any assumptions or limitations
- requirements.txt file

We assume the end user has **Python 3.12** installed.

Part 1 – Historical & Live Data Replay Engine

You need to implement a **data replay engine** that can seamlessly replay both **historical** and **live** data streams using a **unified abstraction layer**.

Requirements

1. **Replay Abstraction**
    - Both historical and live data should share the same replay interface.
    - Assume **live data** is delivered asynchronously via **websocket** in **JSON format**.
    - Historical replay should use the same interface,.
2. **Message Queue**
    - All data points must be pushed into a **queue**.
    - The queue must:
        - Support **thousands of messages per second**.
        - Allow **multiple external consumers/processes** to read from it (used in Part 2).
        - Handle **backpressure** gracefully (avoid uncontrolled memory growth).
    - The choice of queue is up to you (e.g., Redis, Kafka, RabbitMQ, multiprocessing.Queue, etc.), but justify your decision.
3. **Historical Replay Logic**
    - You are provided with `historical_sample.csv` and `live_sample.csv`.
    - The **historical data** contains a `latency` column. Your replay must respect both the **timestamp** and the **latency**.
    - Example (replay order must respect effective arrival time = `timestamp + latency`):

      `Timestamp: 2025-01-01 00:00:00, latency: 120ms  [1]`
    - `Timestamp: 2025-01-01 00:00:00.200, latency: 1ms [2]`
    - `Timestamp: 2025-01-01 00:00:00.700, latency: 10ms [3]`

      `Queue order should be: [2, 3, 1]`

4. **Mode Switching**
    - The engine must support switching between **historical** and **live** replay.
    - Switching should be **dynamic**, e.g., via an API call, CLI input, or message command (your choice).
    - On receiving `mode = historical`, replay historical data.
    - On receiving `mode = live`, switch to live replay.
5. **Bonus (state persistence)**
    - If the app switches from `historical → live → historical`, it should **resume historical replay from the last processed point** rather than restarting.
6. **General Requirements**
    - Add structured **logging** (with timestamps, severity levels, and clear messages).
    - Handle errors gracefully (e.g., invalid input, queue overload, network interruptions).
    - Code should be modular, readable, and testable.

Part 2 – Mid-Price Processor

Create a **separate application** that consumes data from the queue (produced in Part 1) and computes **mid prices**.

Requirements

1. **Mid-Price Calculation**
    - Formula:

      ```
      mid_price = 0.5 * (bid_price + ask_price)
      ```

    - Results must be written to a file `mid_prices.log` in the format:

      ```
      Timestamp, mid_price
      ```

    - 2025-01-01 00:00:00.120, 200.5

2. **Latency Filtering**
    - If (and only if replay mode = historical) the message's latency is greater than a configurable `LATENCY_THRESHOLD` (e.g., 20ms):
        - Skip mid-price calculation.
        - Instead, log an error line to `errors.log` in the format:

```
No mid price at <timestamp> as latency
<latency>ms is bigger than
<LATENCY_THRESHOLD>ms
```

3. **Performance**
    - The system should be designed to process a **high throughput** of messages per second.
    - Minimize blocking operations, consider batching, buffering, or async writes.
    - Properly handle concurrent consumers.
4. **General Requirements**
    - Include robust logging and error handling.
    - Code should be clean, modular, and extensible.

Evaluation Criteria

We will evaluate your submission based on:

1. **Architecture & Design**
    - Abstractions, modularity, scalability
    - Correct use of async/parallelism
2. **Correctness**
    - Replay order correctness (timestamp + latency)
    - Accurate mid-price computation
    - Proper handling of latency threshold

3. **Performance**
    - Ability to handle thousands of messages per second
    - Efficient queue choice and consumer handling
4. **Robustness**
    - Logging, error handling, resilience to backpressure
5. **Code Quality**
    - Readability, structure, comments, typing hints (if used)
    - Clear documentation in README

Submission

- Submit your solution as a **GitHub repository**.
- The repository must contain:
    - `README.md` with:
        - Overview of design
        - Setup & run instructions
        - Any additional notes
    - Codebase for Part 1 and Part 2
    - Sample output logs (`mid_prices.log`, `errors.log`)