SONAR HEALTH TECHNICAL CHALLENGE

Heart Rate Metrics Ingestion System

Overview: Build a FastAPI service that ingests heart rate data from multiple devices and provides efficient querying capabilities. The system should handle real-world scenarios like out-of-order data, duplicate readings, and concurrent requests.
Tech stack: FastAPI, Polars, Python 3.11+, Parquet (Data file format).

Part 1: Data Ingestion Endpoint (POST /metrics/heart-rate)

- Store data in local Parquet file(s) in a data/ directory
- Handle high throughput (100+ requests per second)
- Support data from multiple devices for the same user
- Validate heart rate values (must be between 30-220 bpm)

Request Body: {
"device_id": "device_a",
"user_id": "user_123",
"timestamp": "2024-01-15T10:00:00Z",
"heart_rate": 75
}
Response: { "status": "accepted" }

Part 2: Data Query Endpoint (GET /metrics/heart-rate)

- Return data sorted by timestamp (ascending)
- Aggregate the data in 1 minute buckets by the average
- Efficiently query time ranges, typically 1 day of data (should handle thousands of records)
- Use Polars for data processing

Query Parameters: user_id (required), start (required - ISO 8601 timestamp), end (required - ISO 8601 timestamp), device_id (optional).

Response: {
"user_id": "user_123",
"data": [
{
"timestamp": "2024-01-15T10:00:00Z",
"heart_rate": 75,
"device_id": "device_a"
},
{
"timestamp": "2024-01-15T10:01:00Z",
"heart_rate": 78,
"device_id": "device_a"
}
],
"count": 2
}

Part 3: Device Priority Handling

Some devices are more accurate than others. When multiple devices report heart rate for the same timestamp, use the device with higher priority. For now we can keep it simple storing this device priority as an internal configuration:

- device_a: priority 1 (highest - medical grade)
- device_b: priority 2 (consumer wearable)

Example:
10:00:00 - device_b reports HR=80
10:00:00 - device_a reports HR=75
→ Query should return HR=75 (device_a has higher priority)

Testing Your Solution

We will provide a data generator script (generate_data.py) that:
- Sends 10,000 heart rate readings over ~2 minutes
- Simulates realistic scenarios:
  - Multiple devices sending data concurrently
  - Out-of-order timestamps
  - Duplicate readings
  - Burst traffic patterns

To test your solution:
Terminal 1: Start your API -> uvicorn main:app --reload
Terminal 2: Run the data generator -> python generate_data.py
Terminal 3: Query the data -> curl "http://localhost:8000/metrics/heart-rate?user_id=user_123&start=2024-01-15T10:00:00Z&end=2024-01-15T10:30:00Z"

What We're Looking For

Core Functionality (Required):
Both endpoints are working correctly - Data persisted to Parquet format - Device priority logic implemented - Data validation (invalid heart rates rejected) - Proper error handling (400 for bad requests, 404 for no data found) - Type hints throughout the code.

Code Quality

Clean, readable code structure - Proper use of Polars (avoid row-by-row iteration) - Async/await patterns used correctly - Meaningful variable and function names

Evaluation Process

After submission, we'll:
1. Review your code structure and implementation
2. Run the data generator against your API
3. Test query performance and correctness
4. Discuss your design decisions in a 45-minute code review session

During the code review, we'll ask about: Design choices and trade-offs - How you'd handle production scenarios - Performance considerations - Potential improvements and scalability.

Production Considerations (Bonus)
Handles concurrent writes safely - Efficient file organization for fast queries - Graceful handling of edge cases (duplicates, out-of-order data) - Performance optimizations for high throughput - Basic observability (logging, metrics, health check endpoint) - Tests for critical functionality

Submission Guidelines
1. Code: Push to a GitHub repository.
2. README.md should include: Setup/installation instructions - How to run the API - How to run tests (if any) - Design decisions and trade-offs you made - Known limitations or future improvements.
3. Project Structure: Open to you - Organize your code logically.
4. Dependencies: Keep it minimal. Required: fastapi, uvicorn, polars, pydantic.

Hints & Tips
- Parquet is optimized for batch writes, not single-row appends
- Consider how you'll handle concurrent POST requests
- Think about how to organize files for efficient querying
- Polars has both eager and lazy evaluation modes
- The mode="append" parameter in write_parquet() might not work as you expect with concurrent writes

Questions?
If you have clarifying questions about requirements, please ask! We want to see how you handle ambiguity and make technical decisions.

Good luck!