Backend Assessment For Usman Asif

Overview (goal)

Build a **multi-tenant ecommerce analytics backend** that can **ingest, store, query and stream millions of records** efficiently and safely. The system must support heavy bulk writes, real-time aggregation, streaming exports, idempotent webhooks, rate-limit & backpressure handling, and be observable and profiled for performance. Everything must be delivered as a runnable project (Docker optional) with documentation and performance test scripts.

Requirements: Implementation in Python using **Django + Django REST Framework** for core APIs.

DB: sqlite3 or MSSQL allowed, but must **implement and demonstrate SQL optimizations** (raw SQL, indexes, partitioning simulation) and **document (.md)** how they'd migrate/modify for Postgres/MySQL and production scale.

Setup & dataset

- Provide a script `gen_dataset.py` that **generates synthetic data** (CSV/SQL) for:
  - `tenants` (N=10)
  - `products` (per tenant: 500k)
  - `orders` (per tenant: 2M)
  - `order_items` (avg 3 items per order)
  - `price_history` (per product, 100 samples)
  - `stock_events` (millions of stock updates)

- The generator must:
  - produce **data files** and optionally **bulk insert** into DB in chunks,
  - support adjustable sizes (e.g., `--orders 2000000 --tenants 10`),
  - measure and print ingestion throughput (rows/sec).

- Because generating 10s of millions locally may be slow, include smaller presets (e.g., 1M orders) and instructions to scale.

Required APIs / Endpoints (must implement)

*Each endpoint is judged both for correctness and for how it handles **scale** (memory, CPU, I/O), concurrency, and edge cases.*

1) Bulk ingest — chunked, idempotent

Endpoint: `POST /api/v1/ingest/orders/`

- Accepts: `multipart/form-data` or `application/x-ndjson` with millions of order records in chunks, or `Content-Type: application/octet-stream` for zipped ndjson.

- Requirements:
    - Accepts an `Idempotency-Key` header. Requests with the same key must be safe to retry.
    - Support **chunked upload** (client sends many chunks); server must assemble or stream-insert without loading the entire payload into RAM.
    - Validate schema and reject bad rows but continue inserting other valid rows (return a partial success with per-row error metadata).
    - Provide **resumable upload** tokens so a failed chunk can resume rather than restart.
    - Insert must be batched and use the most-efficient method available (bulk insert / executemany / COPY-like technique). When using ORM, show tradeoffs and implement raw SQL paths for speed.
    - Response: summary (rows_received, rows_inserted, rows_failed, processing_time, idempotency_key).

2) High-throughput search + filtering (cursor + column projection)

Endpoint: `GET /api/v1/tenants/{tenant_id}/orders/search`

- Supports:
    - Cursor-based pagination with opaque continuation token (no offset pagination).
    - Complex filters: date ranges, product ids, price ranges, order_status, full-text search on customer name/email (simulate FTS).
    - Column projection parameter (client chooses which columns to return).
    - `fields` parameter and `limit` up to 100k per request (server must stream results).

- Requirements:

- Implement **streaming JSON** (JSON Lines or chunked JSON array) so large result sets do not overload memory.
- **Low memory footprint:** server must not build the entire result in memory.
- Provide an explanation of the chosen cursor format and how it handles deletes/updates.

3) Aggregation at scale — approximate + exact modes

Endpoint: `GET /api/v1/tenants/{tenant_id}/metrics/sales`

- Accepts `group_by` (day/hour/product/category), `start_date`, `end_date`, `precision` (`approx` or `exact`).
- `approx` mode must use a probabilistic algorithm (e.g., HyperLogLog for unique customers, t-digest for percentiles) or streaming sketch; `exact` mode must produce precise numbers.
- Requirements:
  - Must compute aggregations across millions of orders without fully materializing rows using streaming aggregation, windowed processing, and/or incremental pre-aggregations.
  - Implement a **materialized view** simulation (cached pre-agg table) and an invalidation policy.
  - Response must include metadata: method used, estimated error bounds if `approx`.

4) Real-time price-sensing API (delta detection)

Endpoint: `POST /api/v1/tenants/{tenant_id}/products/{product_id}/price-event` (webhook style)

- Accepts price update events; must:
  - Apply rate-limiting per product and per tenant.
  - Detect significant price anomalies (e.g., sudden >= X% change), mark events for review.
  - Support idempotency (webhook retries).
  - Publish to an internal change stream (for this assessment, store events in a `price_events` table and implement an endpoint to stream recent anomalies).

- Provide `GET /api/v1/tenants/{tenant_id}/products/{product_id}/price-anomalies` as a streaming endpoint.

5) Streaming export — CSV / Parquet (resumable & compressed)

Endpoint: `POST /api/v1/tenants/{tenant_id}/reports/export`

- Accepts filters and a `format` param (`csv` or `parquet`).
- Builds the export **streaming** to disk (or to response) without loading all data into RAM.
- Must support:
    - **Resumable downloads** (range requests / continuation token).
    - On-the-fly compression (gzip) and chunked transfer encoding.
    - Checkpointing for long exports (if interrupted, resume).
- Provide a sample download client demonstrating a resume.

6) Conflict resolution & transactional batch updates

Endpoint: `PUT /api/v1/tenants/{tenant_id}/stock/bulk_update`

- Accepts a batch of stock events that must be applied **transactionally per product** (i.e., product-level atomicity). Some products can succeed, others fail.
- Requirements:
    - Use row-level locking or application-level advisory locks to avoid race conditions.
    - If two concurrent bulk updates touch the same product, ensure consistent final state.
    - Provide conflict resolution options: `last_write_wins`, `merge` (apply additive delta), or `reject`.

7) Observability and backpressure

- Implement:
    - Basic metrics endpoints (Prometheus-style `/metrics` or JSON): ingestion rate, average processing time, queue length, memory usage, DB query durations (P50/P95).
    - Graceful degradation on overload: if ingestion queue is too long, return `429` with Retry-After and expose current queue length and expected delay.
    - Logging structured JSON with request ids and idempotency keys.

Additional required artifacts (deliverables)

1. **Source code** implementing the above (Django project + DRF). Provide `requirements.txt` and startup instructions.
2. `gen_dataset.py` to generate synthetic dataset and optionally bulk insert.
3. **README** with:
    - How to run the project (locally with sqlite3 or MSSQL).
    - How to run performance tests.
    - Explanation of design decisions and where optimizations were applied.
    - Migration plan and changes required for production DB (Postgres), plus estimated hardware for target scale (justify numbers).

4. **Performance test scripts**: sample `wrk` or `locust` scripts (or Python scripts using `requests`) that:
    - Ingest 1M orders in X chunks and report throughput.
    - Query large resultset streaming and measure memory and latency.
    - Execute concurrent stock bulk updates to show correctness under race.

5. **EXPLAIN ANALYZE** or query plan evidence for heavy queries (even with sqlite(if you are using sqlite), show `EXPLAIN` output), and proposed indexes.
6. **Short report** (max 2 pages) with:
    - Bottlenecks discovered,
    - Changes to implement for production (partitioning, z-order, columnstores, WAL tuning, read replicas),
    - How to handle GDPR / PII removal requests in this system.

7. **Unit + integration tests** for correctness (including idempotency behavior, resumable uploads, conflict resolution).

Constraints & evaluation environment

- If you are using **sqlite3** locally, you must explain how features like partitioning, parallel bulk insert, COPY / COPY FROM (or COPY-like) would map to Postgres/MySQL.
- Assume a machine with **8 vCPU and 16 GB RAM** (for test guidelines). Must provide benchmark numbers from your environment (screenshot).
- Memory budget target: **API endpoints must avoid reading more than 200MB total RAM for a streaming query or export** (measured during tests).

Scoring rubric

Total: **100 points** — pass threshold set high.

Architecture & design — 25 pts

- Clear multi-tenant data model, tenancy isolation and secure access (10)
- Thoughtful scaling plan: indexing, partitioning, read replicas, caching (10)
- Observability, metrics, and graceful degradation (5)

Correctness & robustness — 25 pts

- Idempotency correctness across ingestion/webhooks (8)
- Transactional batch integrity and conflict resolution (8)
- Proper error handling and partial failures (9)

Performance & scalability — 25 pts

- Bulk ingest throughput, documented and optimized (10)
- Streaming search/export works with low memory and measured (10)
- Aggregation: approximate vs exact modes and materialized pre-agg strategy (5)

Code quality & tests — 15 pts

- Clean, documented code; OpenAPI; unit/integration tests (8)
- Security practices (input validation, rate limiting, auth placeholders) (7)

Deliverables & documentation — 10 pts

- Dataset generator, README, EXPLAIN outputs, performance reports. [Documentation created with AI will be rejected, but you can use ChatGPT for rephrasing and formating the documentation]. (10)

Bonus (extra hard):

- +10 pts: Provide an alternative implementation path using async (FastAPI/ASGI) for ingestion and justify tradeoffs.
- +10 pts: Implement a simple in-memory sketch (HyperLogLog or t-digest) from scratch and use it in `approx` aggregations.