Question 1: Data Migration from MongoDB to a SQL Database

Problem Statement:

You are tasked with migrating data from a MongoDB database to a SQL database (e.g., PostgreSQL). The MongoDB database contains a collection called products with the following sample document structure:

```
{
    "_id": "12345",
    "name": "Laptop",
    "price": 999.99,
    "stock": 50,
    "category": "Electronics",
    "attributes": {
        "brand": "TechBrand",
        "color": "Silver"
    },
    "created_at": "2023-10-01T10:00:00Z"
}
```

The target SQL database should have a normalized schema to store this data efficiently. Write a script to:

1. Design a normalized SQL schema for the products data.
2. Write a program (in Python or another language of your choice) to migrate the data from MongoDB to the SQL database.
3. Handle potential issues like duplicate records, missing fields, and data type mismatches.

Requirements:

- Ensure the migration is efficient for large datasets (e.g., millions of records).
- Provide error handling for connection issues and data inconsistencies.
- Write clean, maintainable, and well-documented code.

Bonus:

- Add a mechanism to log migration errors for debugging.
- Support resuming the migration in case of failure.

Evaluation Criteria:

- Correctness of the SQL schema (normalization, indexing).
- Efficiency of the migration process (batch processing, connection pooling).
- Robustness of error handling and logging.
- Code readability and documentation.

Question 2: Real-Time Inventory Deduction API

Problem Statement:

You are tasked with building a real-time communication API to handle inventory deductions for an e-commerce platform. The API must support concurrent requests for inventory deduction, ensure fault tolerance, and prevent over-deductions (e.g., negative stock). The system should handle real-time updates and notify clients of success or failure.

Requirements:

1. Design a REST or WebSocket API to deduct inventory for a given product.
2. Implement a mechanism to handle concurrent requests safely (e.g., using locks or optimistic concurrency).
3. Ensure fault tolerance (e.g., retry mechanisms, circuit breakers, or failover).
4. Provide real-time feedback to clients (e.g., via WebSocket or server-sent events).
5. Handle edge cases like insufficient stock or network failures. 6. Write clean, maintainable code with proper documentation.

Bonus:

- Add a mechanism to roll back deductions in case of partial failures.
- Implement rate limiting to prevent abuse.

Evaluation Criteria:

- Correctness of concurrency handling (no race conditions, no over-deductions).
- Robustness of fault tolerance and error handling.
- Scalability of the real-time communication mechanism.
- Code readability, modularity, and documentation.

Notes for Interviewers

- **Difficulty Adjustment**: For junior candidates, simplify the requirements (e.g., skip WebSocket or fault tolerance). For senior candidates, add complexity (e.g., distributed transactions, sharding).

- **Follow-Up Questions**:
    - How would you scale the migration to handle 100M records?
    - How would you handle distributed inventory across multiple warehouses?
    - What trade-offs exist between REST and WebSocket for real-time updates?
- **Common Pitfalls to Watch For**:
    - Ignoring concurrency issues (e.g., no locks or transactions).
    - Poor schema design (e.g., unnormalized tables).
    - Lack of error handling or logging.
    - Overcomplicating the real-time mechanism (e.g., using a message queue when WebSocket suffices).