Software Engineer Team Lead

Technical Vetting Assignment

Instructions & Requirements

Take Home Assignment: Loyalty Points Service (C# + SQL + AWS)

Context

You're working on Zapper's Value Added Services team. Product wants a small "Loyalty Points" capability to reward customers for purchases across multiple merchants. Merchants will send purchase events; the system should award points based on simple rules and maintain a reliable balance per customer and merchant. Merchants may occasionally resend the same purchase event, and traffic can arrive in bursts. The feature must be production-minded: resilient, observable, and easy to deploy in containers.

Core technologies to use:

- C# (.NET, any recent LTS)
- A relational database (SQL Server, PostgreSQL, or MySQL)
- AWS (choose at least one managed service in your implementation)
- Docker for containerization

You decide the API design, data model, error behavior, and overall architecture.

Requirements

The tasks to be completed.

Task 1: Purchase ingestion and points ledger

Build a small system that:

- Accepts an inbound "purchase event" from a merchant and stores it.
- Calculates and applies loyalty points for the customer and merchant.
- Ensures the same purchase, if sent multiple times, only impacts points once.
- Exposes a way to retrieve a customer's current points balance per merchant and view recent points entries.

Business notes:

- Merchants send a purchase JSON similar to:

```
{
  "transaction_id": "m-123-abc",
  "merchant_id": "zcoffee-001",
  "customer_id": "cust-789",
  "amount": 245.70,
  "currency": "ZAR",
  "timestamp": "2025-03-05T10:15:30Z",
  "payment_method": "card"
}
```

- A simple rule is sufficient (for example, 1 point per 10 ZAR; you decide rounding).
- Amounts may vary; the same transaction might be sent again by a merchant.
- Design your own data model and API shapes for:
    - Ingesting purchases
    - Retrieving a customer's balance and transaction/points history

Constraints:

- Use C# and a relational database.
- Handle concurrent requests that affect the same customer or purchase without awarding points multiple times.
- Include a basic seed or script to run the app locally with a database.

Task 2: AWS integration and asynchronous processing or archiving

Integrate with at least one AWS managed service and reflect it in the code:

- Choose one of the following paths (pick one):
    - Queue-driven processing: After accepting a purchase, place a message onto a managed queue and process it in a background worker to compute/apply points. Assume the queue can surface the same message more than once.
    - Object storage archiving: Archive the raw purchase JSON to managed object storage at ingestion time (e.g., one object per purchase), and compute/apply points in your service.

Expectations:

- Your code should run locally without requiring an actual AWS account. Provide a simple way to run locally (e.g., a switchable provider, stub, or local emulator) while keeping the production-facing AWS implementation in the codebase.
- Briefly document in SOLUTION.md how you would deploy this component in AWS (service choice, compute, networking, and how it would evolve with traffic growth).

Task 3: Operations, packaging, and confidence

Make the service production minded and easy to run:

- Containerize the service(s) with Docker.
- Provide health and status endpoints suitable for container orchestration.
- Add logging that helps trace a purchase from ingestion through points application.
- Include minimal automated tests that give confidence in:
    - Your points calculation
    - Preventing multiple awards for the same purchase
- Provide a short note in SOLUTION.md on:
    - How you'd monitor it (key signals/metrics/logs you'd watch)
    - How you'd handle bursts and ensure stable behavior
    - How you'd evolve the data model if new reward rules arrive (briefly outline the approach)

Deliverables

1. A single GitHub repository containing:

- All the source code for the assignment
- A README.md with instructions to run locally.
- A SOLUTION.md describing the design and trade-offs you made.

2. Video Demo (5-10 minutes using Loom or similar):

- Demonstrate the working project by doing a voice over
- Walk through your code architecture
- Explain key design decisions and any trade-offs made
- Highlight interesting technical implementations

Time Estimate

This should take ±4 hours to complete, but you can take as long as you need.