System Design Exercise

Tasks:

1) Define some structure for this new service, what classes/methods/functions would you define? Just write stubs, no implementation details needed.

The system is designed to return a sorted list of content IDs based on a priority score that combines user preferences and the sales boost factor.
All codes are provided in the system design folder with specific Python files.

Solution Overview

My implementation includes three main components:

- ContentSorter: The core class that sorts the content based on scores.
- UserData: Fetches user affection scores for different content categories.
- ContentData: Fetches content details like category and sales boost.

Code Structure

1. ContentSorter Class

This is the main service class that performs the sorting logic.

- Methods:
  - sort_content: This method takes user_id and content_ids as input parameters and returns the content_ids sorted based on user preference and sales boost factor.
  - calculate_priority_score: Calculate priority score based on user affection and sales boost and return a score.

2. UserData Class

- Method:
  - get_user_affection_scores: Fetch user preference scores by user_id and returns a dictionary Example: { 'furniture': 12, 'dairy_products': 81 }

3. ContentData Class

- Method:
  - get_content_details: Returns a tuple containing the content category and sales boost.

Design Principles Followed

1. Dependency Injection:
   - I inject UserData and ContentData into ContentSorter, making the system flexible and testable.
2. Single Responsibility Principle (SRP):
   - ContentSorter: Focuses on sorting logic.
   - UserData: Fetches user scores.
   - ContentData: Fetches content details.
3. Modular Design:
   - Each class has a clear role and functionality.
4. Scalability:
   - We can replace UserData or ContentData with real implementations without changing the ContentSorter.

Summary

The code is clean, modular, and well-structured for solving the problem. The use of Dependency Injection to design principles like SRP makes it flexible and easy to extend.

Task2:

What would you need to change in order to be able to run an A/B-test with different sort-order-calculations?

What is A/B Testing?

A/B testing is a method to compare two or more versions of a system (in your case, different **sort-order calculations**) to determine which performs better.

Steps to Implement A/B Testing

1. Define Sorting Strategies:
    - Split the sort-order calculations into functions or classes (default and experimental).
2. User Group Assignment:
    - Implement logic to assign users to the control group or test group using user IDs
3. Integrate Strategies:
    - Modify the system to accept a sorting strategy dynamically.
    - Use the appropriate strategy based on the user's group assignment.
4. Track Metrics:
    - Log key performance indicators (CTR, engagement, conversion) for both groups.
5. Analyze Results:
    - Compare metrics from the control group and the test group.
    - Use statistical analysis to determine if the experimental strategy performs better.

Benefits of This Approach

1. Flexibility:
    - You can easily add more sorting strategies for future A/B tests.
2. Clean Design:
    - The sorting logic is separated into strategies, making the system easier to maintain and extend.
3. Scalability:
    - Works for more than two groups (A/B/C testing) if needed.
4. Minimal Code Changes:
    - You only need to modify how the system selects and applies the sorting logic.
5. Measurable Results:
    - By collecting metrics, you can objectively evaluate the impact of different sort-order calculations.

What Needs to Change?

There are two ways to run A/B test with different sort-order-calculations :
In order to do it there are some **changes** needed to apply. Here are 2 approaches that I suggest

1- Use a simple **if condition** to decide which formula to use to switch between sorting formulas, But there is not a good solution because whenever we want to change or add or even remove a condition we should modify the main code and this is not an appropriate way.

2- The next way is using **the Sorting Strategy Pattern** for the sorting logic, and each sort-order calculation becomes a separate strategy. I used this approach, which is a more efficient and scalable way. We can implement this approach within two types:

The first is using a **class-based solution** in which we can define an interface (or abstract class) for the sorting logic, and each sort-order calculation becomes a separate strategy. In the class-based approach, we Implement Different Sort Strategies by creating multiple classes and implementing the SortStrategy interface. Each class represents a different calculation logic. Modify the ContentSorter class to accept a strategy dynamically. The sorting logic will depend on the strategy injected at runtime. Instead of hardcoding the sorting logic, create an interface that allows plugging in different sorting strategies (e.g., "default" and "experimental"). This approach uses the Strategy Design Pattern to separate the logic. Log and monitor results for both groups (e.g., clicks, conversions) to determine which sorting strategy is better.

The Second is the **Function-Based Strategy Pattern** which You don't need to create classes for different strategies. The ContentSorter class now accepts a function for the strategy instead of a class. The ContentSorter class accepts a strategy function dynamically, making it clean and simple, also we can add new strategies easily by defining more functions.
This keeps the design flexible by Implementing a dynamic strategy selection method:
get_sort_strategy: determines which function to use based on the user ID.

In this approach, I implement 2 methods for separate strategies:

- default_sort_strategy: for returning default sort strategy
- and experimental_sort_strategy: for returning experimental sort strategy

Benefits of Function-Based Strategy Pattern

1. Simpler and Cleaner:
   No need for classes or interfaces. Each strategy is just a function.
2. Flexible:
   You can dynamically switch between strategies by passing the appropriate function.
3. Testable:
   Each strategy function can be tested independently.
4. Scalability:
   Add new strategies easily by defining more functions.

Summary
This function-based approach combines the benefits of the Strategy Pattern with simplicity. You achieve flexibility, clean design, and easy A/B testing without needing classes for strategies.

Collect Metrics for Evaluation

To measure the success of each sorting strategy, you need to log metrics for both groups. Key metrics to track include:

- Click-through rate (CTR): Percentage of content clicked by users.
- Engagement: Time spent viewing the sorted content such as view_duration in brochures views dataset.

This data will help you analyze which sorting strategy performs better.

In this part, I answered both Task 3 and 4 together

Task 3:

What problems could occur if the list of content-ids passed as a parameter becomes very large (e.g., 10,000)?

Task 4:

Any suggestions what kind of data store to use for storing content_data and user_data so it can be queried in a fast and scalable way?

When we want to pass large data as a parameter, some problems may occur:

High Memory Usage: It will use high memory and increase the risk of slowing down the application or even crushing it.

Increase processing time: when we send a request to our API or database to pass large data the processing time will increase because sorting 10,000 content IDs requires calculating scores for each content-ID and then sorting the list based on these scores is a time-consuming process.

In order to solve these problems, we have some **solutions**, for example:

Batch processing: Instead of processing all data at once, we divide data into smaller chunks or batches like machine learning the fetch and process them.

Caching : Use a caching layer like Redis to store frequently accessed data, it has some benefits like reducing database load and speeding up response time for frequently accessed data.
Examples: User preference scores for content categories.Content details like category and sales-boost factors.
Check the cache first. If the data is present, return it directly.
If not, fetch data from the database, store it in the cache, and return it.

Asynchronous Processing: Use background workers or async calls to fetch and process data concurrently.

Optimize Data Fetching: Fetch only required data instead of retrieving unnecessary fields. Set limit also helps us to optimize our data fetching.

Precomputing: Precomputing helps us save time, reduce query time, and improve response time. For example, in our case, we can precompute user scores and update them every hour. When a request comes in, we fetch the precomputed scores instead of calculating them on the spot. This makes the process much faster.

NoSQL Databases: For large datasets, we can use NoSQL databases like MongoDB and DynamoDB in AWS because they scale horizontally, and we can handle increasing loads of large data.
MongoDB stores data in a JSON-like document format, perfect for content_data and user_data, where each item has a simple structure.
Amazon DynamoDB (NoSQL Database)
Managed, serverless, and scales automatically with no need for manual maintenance.
Extremely fast key-value lookups for user_id or content_id.

Indexing the Database: Most repeated columns being queried must be indexed. Indexes make lookups much faster, significantly improving query performance. For example, in our case, **content_id** should be indexed.

Task 5:

What technology stack would you suggest to implement this solution?

To answer this question, I suggest two approaches for implementation of my solutions:

Approach 1: Simple Approach with MongoDB and Microservice
Approach 2: AWS-Based Infrastructure

Lets start with Approach 1**: Simple Approach with MongoDB and Microservice**

Introduction

This approach implements a simple content sorting service using a Python-based microservice and MongoDB for data storage. It focuses on simplicity, flexibility, and ease of deployment without cloud-native services.

System Overview

The system is designed to process sorting requests dynamically, with the following workflow:

1. Input: Requests containing user_id and a list of content_ids.
2. API Gateway: Routes incoming requests to the Scoring Service.
3. Scoring Service: Fetches user preferences and content.
4. A/B Testing Logic: Dynamically selects between two sorting approaches:
    - Sort Logic A: Default scoring and sorting logic.
    - Sort Logic B: Experimental or improved sorting logic.
5. Sorting Service: Applies the chosen sorting logic to order content IDs.
6. Output: The final sorted list of content IDs is stored and returned.
7. Monitoring: Logs system performance and A/B test metrics for analysis

---

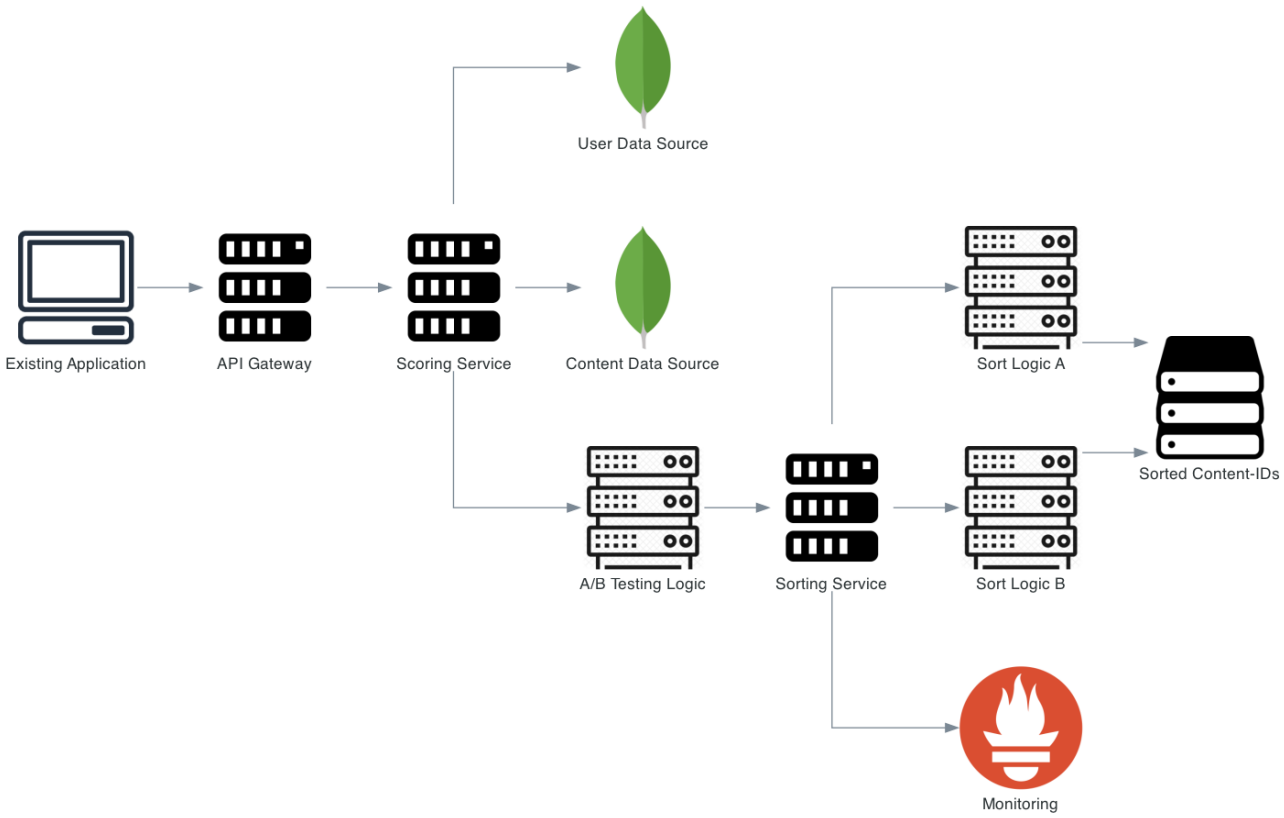
Scalability Considerations

1. MongoDB indexing for faster queries.
2. Pagination or batch processing for large input lists.
3. Dockerization for easy deployment.

Technology Stack

- Backend: Python (Flask/FastAPI).
- Database: MongoDB.
- Deployment: Dockerized microservice using Docker Compose.

Approach 1 Diagram



Content Sorting System with MongoDB and A/B Testing

Approach 2 :

AWS-Based Infrastructure

Introduction

his AWS-based design provides a scalable, reliable, and cost-effective solution for **real-time content sorting** with **A/B testing** for sort-order calculations. Using AWS Lambda for serverless computing, DynamoDB for low-latency data access, and SQS for decoupled processing ensures the system can handle large-scale workloads efficiently. Monitoring via CloudWatch ensures performance visibility and A/B test evaluation for continuous optimization.

System Overview

The workflow processes sorting requests with the following components:

1. Input:
    - Requests from the Existing Application are sent to API Gateway.
2. Queue Management:
    - Amazon SQS queues incoming requests to input handling and processing.
3. Scoring Service:
    - AWS Lambda retrieves data and computes content scores based on user preferences and sales-boost factors.
4. A/B Testing Logic:
    - Dynamically assigns requests to one of two sorting paths: Sort Logic A or Sort Logic B.
5. Sorting Service:
    - Lambda Functions apply the selected sorting logic to rank content IDs.
6. Output:
    - Sorted content is stored in Amazon S3 for retrieval.
7. Monitoring:
    - Amazon CloudWatch logs metrics, system performance, and A/B testing results.

Technology Stack

API Management: Amazon API Gateway.

Queue Service: Amazon SQS for reliable request handling.

Serverless Compute: AWS Lambda for scoring and sorting processes.

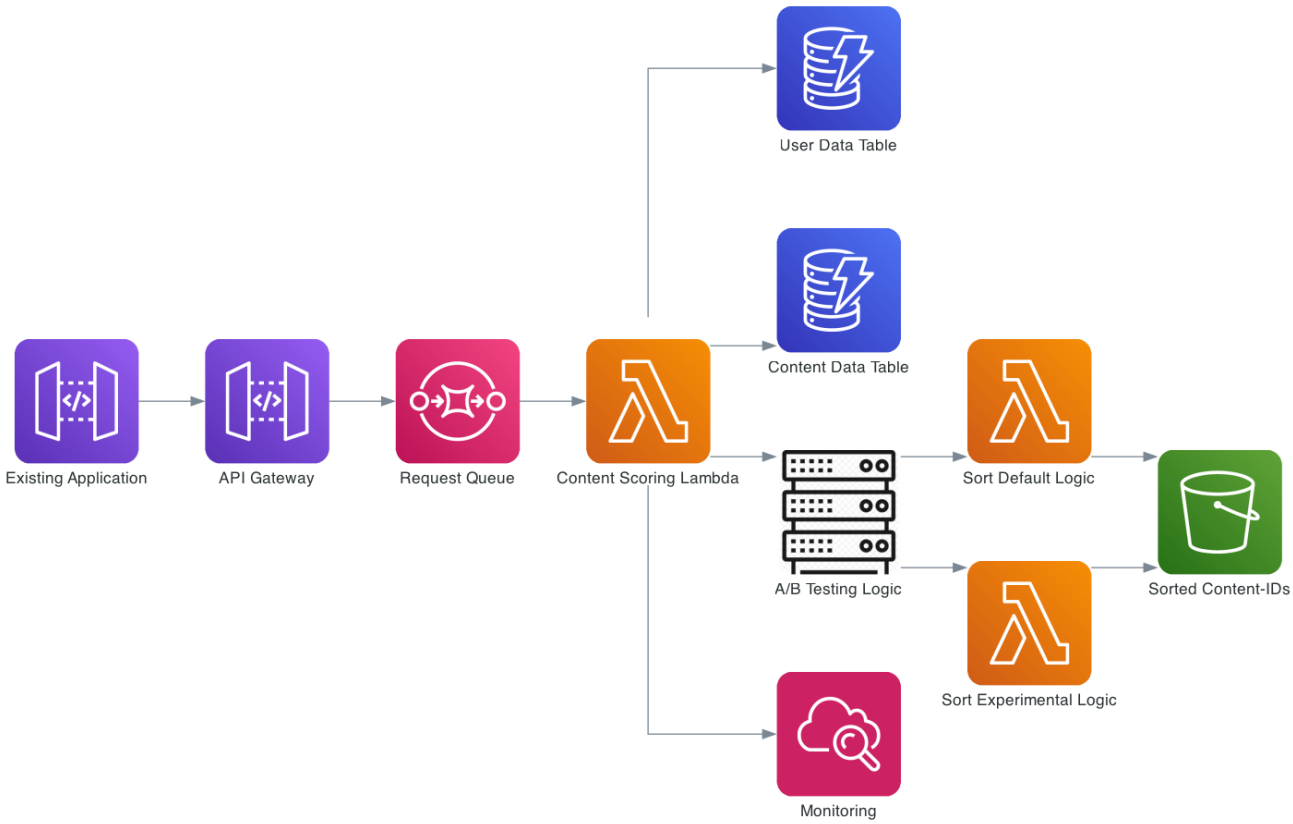
Database: Amazon DynamoDB for user and content data.

Storage: Amazon S3 for sorted output storage.

Monitoring: Amazon CloudWatch for metrics and logging.

Language: Python for Lambda functions.

AWS-Based Infrastructure



AWS Content Sorting System with A/B Testing