Decision record template for Mocroservices

Introduction

1. Prologue (Summary)
2. Discussion (Context)
3. Solution (Decision)
4. Consequences (Results)

Specifics

5. Prologue: In the context of the system architecture, we face the issue of large load on servers, so, we decided for MicroServices to achive maintanability and scalability accepting higher complexity.
6. Discussion: We face the issue of large load on a single monolithic server due to large AI models for the report generation. Moreover, the failure in one server should not bring down the entire system. We also need to cater with the user need to add more features to the report. Since the users come from diiferent companies and thus the requirements can be different which will need us to extend the services in the future. So, we need easier scalability. Also, failure of one server should not affect the entire system.
7. Solution: We decided to use Microservices Architecture to achieve:
    1. Scalability: Microservices allows individual components of an application to scale independently based on demand.
    2. Flexibility: Each microservice operates independently, enabling us to choose the best technology stack for each component of the AI.
    3. Resilience: Failure in one microservice typically won't bring down the entire system.
    4. Continuous Delivery and Deployment: Microservices promote continuous integration and delivery practices. We can add further festures in the report if the user needs the same.
    5. Easier Maintenance: With smaller, focused services, it's easier to understand, maintain, and debug the system.
8. Consequences: We accept the downsides which come along whith this decision which are:
    1. Complexity: Managing a distributed system with numerous interconnected microservices can be complex.
    2. Data Management Challenges: Microservices often have their own databases or data stores, leading to data duplication and consistency challenges.

Decision record template for MongoDB

Introduction

1. Prologue (Summary)
2. Discussion (Context)
3. Solution (Decision)
4. Consequences (Results)

Specifics

5. Prologue: In the context of the system architecture, we face the issue of large load on servers, so, we decided for MicroServices to achive maintanability and scalability accepting higher complexity.
6. Discussion: A proper SQL would have been very cumbersome to adjust to our data base schema. The usage of the database should also be easy. We also need flexibility in the database.
7. Solution: We decided to use MongoDB to achieve:
    1. Schema flexibility: MongoDB is a NoSQL database, which means it doesn't require a predefined schema. This flexibility allows us to store and manage unstructured or semi-structured data easily, making it ideal for our application.
    2. Scalability: MongoDB is designed to scale out horizontally, meaning we can easily distribute data across multiple servers or clusters to handle increasing loads.
    3. High performance: MongoDB's document-oriented data model and built-in sharding capabilities contribute to its high performance. It supports various types of queries, including complex aggregations, and can efficiently handle read and write operations even at scale.
8. Consequences: We accept the downsides which come along whith this decision which are:
    1. Memory usage: MongoDB can consume significant amounts of memory, especially when working with large datasets or performing complex queries. Proper indexing and schema design can mitigate this to some extent, but it's essential to monitor and manage memory usage, particularly in memory-constrained environments.

Decision record template for flask and Fastapi

Introduction

1. Prologue (Summary)
2. Discussion (Context)
3. Solution (Decision)
4. Consequences (Results)

Specifics

1. Prologue (Summary): For sending API calls to our backend which is rwritten in python, we needed to find a routing framework to handle the incoming requests. We have used FASTAPI for handling most of the calls but have used Flask for the AI models.
2. Discussion (Context): We used FastAPI for most of the endpoints as it is very performant for handling large amounts of data. Moreover, we had experience using FastAPI. Flask was used for calling the AI models as the data transfer

over network between them is not that much. Thus FLask was ideal as it has a very easy to use interface and is easier to maintain and extend.
3. Solution: Our choices for using the appropriate framework at the proper position ensure that the codebase is easier to maintain and extend in the future while still not compromising on performance.
4. Consequences(Results): We found that the api endpoints defined using FastAPI are very performant and reduce latency while api endpoints in Flask are not a bottleneck for our code as majority of the time is taken in generating the response. The code base is also easier to maintain for the AI module

Decision record template for restful

Introduction

1. Prologue (Summary)
2. Discussion (Context)
3. Solution (Decision)
4. Consequences (Results)

Specifics

5. Prologue (Summary): In the context of sending api requests, we needed to select a architecture style between REST and GraphQL. We decided for using RESTful architecture as opposed to GraphQl to achieve simpler and more maintainable codebase. While this allows lesser control over our api requests, we believe it is justified as our requests are not complex.
6. Discussion (Context): Most of the team had little experience with graphQL and were much more comfortable with using REST. Due to the easier ease of implementation as well, it was decided that REST architecture will be much more conducive for smoother and faster development
7. Solution: The decision will help us make API getways for our separate services in a simple and maintainable manner
8. Consequences: We found that the team found no issues while implementing API endpoints in the REST architecture. Our code is also performant and does not suffer from any latencies. The codebase is also much more cleaner than if we had implement GraphQL