Nimbus2001 - Coding Platform

Colmán Larkin 22209077 Maximilian Girt 22205093 Paul Ermler 23204105

Dorian Bernard 23204094 Nataël Baffou 23204154

Accessing the repository:

https://github.com/maxgirt/COMP41720

* ykalyani17@gmail.com and biddu have been invited to our repo

Video: https://clipchamp.com/watch/Vqbq6HSbcWx

Video Google Drive Link:
https://drive.google.com/file/d/1cgsAqq5HqoibQqLkn0vca4qBfyzbOpVG/view

Synopsis:

System Description

We intended to develop a competitive programming platform similar to Codeforces, Leetcode and Codechef. The idea is that users can choose between different coding problems to solve and then submit their solutions which are automatically tested on test cases. When solving a problem the user can choose between different programming languages and is shown some test cases while others are hidden.

What is the application domain?

The application can be set under the domain of educational technology, specifically focusing on programming and computer science education.It's tailored for learners who want to practice and improve their coding skills. This domain encompasses users such as students, educators, self-learners, and potentially software development professionals. Key technologies in this domain include automated code evaluation, code completion, and a cloud-based infrastructure to support the distributed system.

What will the application do?

The application will provide the user with different coding problems and the option to solve them in Java or Python. Both the problems and test cases are stored in a database. The submissions will be graded automatically on the stored test cases and feedback is returned to the user. The application also stores a submission history. Additionally, the user can request help from an AI assistant that will give the user a hint about what might be wrong with the code.

Technology Stack

Spring Boot Framework

With Java Spring Boot, we create a RESTful API that serves as our interface for client interactions with the system. It handles HTTP requests for submitting solutions, retrieving problems, and communicating with other services in the architecture. We chose it since it is intuitive to use and provides plenty libraries and tools that simply our development and also has great dependency management. Its ability to create RESTful APIs makes it an ideal choice for handling HTTP requests for various operations such as submitting solutions and retrieving problems.

REST

We used REST for the Client-Broker communication and communication of any service with the database. REST makes communication between the React frontend and the backend services straightforward. Its stateless protocol and standard HTTP methods provide a scalable and efficient way to handle client-server interactions.

JMS (Java Message Service)

JMS is utilized for internal communication between the broker and the grading services and ai assistant. It enables asynchronous message passing, which decouples the components, enhancing scalability and reliability. This allows us to easily scale grading services simply by having more grading services consume messages from the message queue.

No-sql database - Mongo DB

MongoDB stores and retrieves coding problems and their solutions. It can handle large amounts of data and complex queries efficiently. It is a NoSQL database, which is also why we wanted to use it, known for its scalability and performance and due to being document-oriented it provides flexible data models which help with complex datasets such as coding problem descriptions and user submissions.

Mongo DB is ideal for handling semi-structured/unstructured data which is common for coding problems and submissions that you would find on a platform such as LeetCode. Because we want to store these submissions so users can keep track of their progress, it’s important that we use a database that allows for unstructured data.

The unstructured data includes problem descriptions, user submissions and test cases. MongoDB's flexible document model is ideal for storing this semi-structured data. Each problem or submission can be a document with fields varying as needed, allowing us to easily accommodate different types of information without the constraints of a rigid schema.

The documented oriented model of MongoDB means that we can create documents/collections of related data and store them together. So for instance, all of the submissions will be stored in a submission collection and so on.

Through its replica set architecture, MongoDB provides out-of-the-box support for data redundancy and high availability. This means that our platform can continue to function

smoothly even if some of the database nodes encounter issues, ensuring that our service remains reliable for the users.

As our platform grows in popularity, we will encounter an increase in data volume and user traffic. MongoDB's horizontal scalability is a significant asset here. We can distribute data across multiple servers as the load increases. This is particularly important for maintaining performance during coding contests when many users might be submitting solutions simultaneously.

Querying is easy with MongoDB as we can access specific data based on the ID we assign to it. Wcan quickly retrieve specific problems, user submissions, or test cases based on various criteria. Indexing capabilities also ensure that these queries remain fast, even as your data volume grows

We can represent all the data in JSON formats making it easy to see the data, specifically coding problems, submissions, and so on.

React Frontend

We used React to create a super simple frontend that allows the user to choose between problems and submit code. We chose it because there are a lot of resources and libraries available that made it easy to include a great code editor.

Docker

We use Docker to containerised the components of our system. It ensures that each system runs in a consistent environment and can be deployed easily. Furthermore, it helps us to easily scale our system. When we need more grading services to grade submissions, we can simply add new docker containers.

System Overview

Main Components

Client/Frontend

- This is where users interact with the system, writing and submitting code in Java or Python for grading and getting feedback.

Message Broker

- Routes messages between the client and the other services e.g. forwards submissions to grading services, stores submissions in DB, retrieves problem descriptions for client

Apache ActiveMQ Server

- Provides message queue handling

Grading Services

- Separate services for Java and Python code evaluation.
- Receives code submissions from the message broker, gets test cases from the database, evaluates them, and sends back the results to the broker

AI Assistance Service

- Queries an external machine learning API to provide code improvement suggestions

MongoDB Server and Database

- Stores programming problems and their solutions as well as permissions
- Accessed by the Database Service to execute the proper CRUD calls.

Database Service

- Wraps database with simple REST API
- Accessed by the Broker Service to retrieve problems and to store submissions
- Accessed by the Grading Service to retrieve the correct solutions (test cases) for comparison.

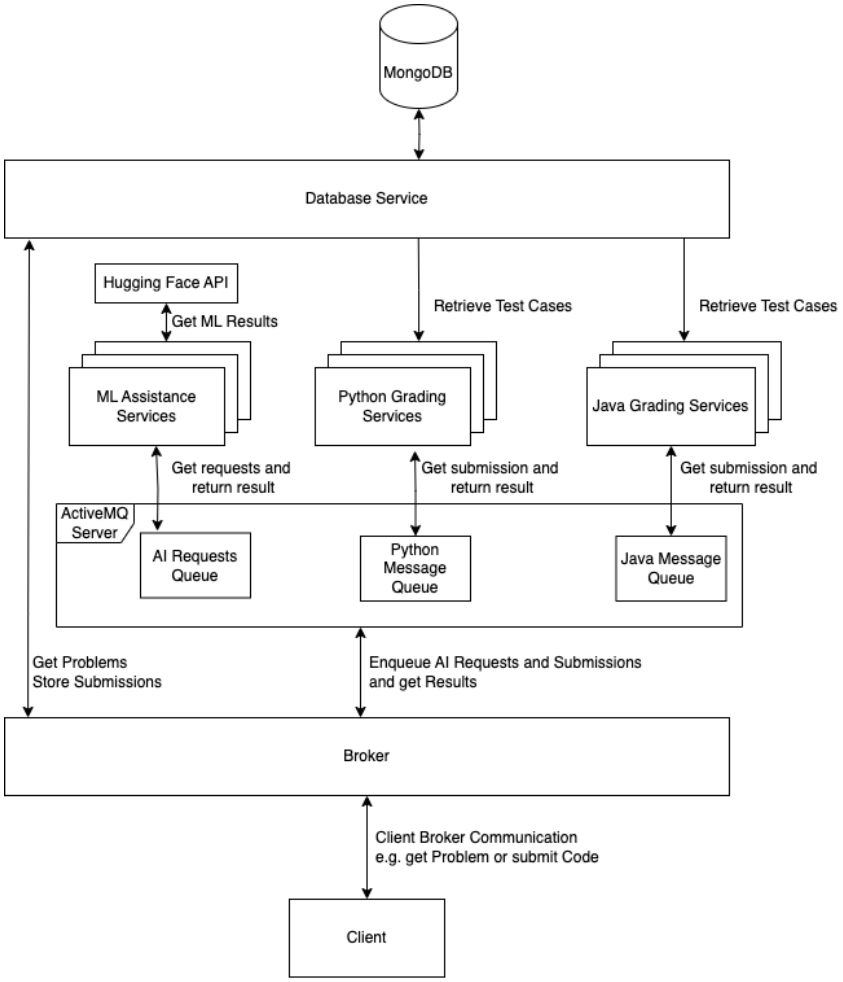

Figure 1. *System Architecture Diagram*

Functionality

Submitting Code

- When a client submits a solution, the request is sent to the broker via REST Api
- The broker then enqueue this submission to the message queue of the specific programming language
- The grader will retrieve the test cases from the Database Service based on the problem id that was sent in the submission.
- Once the problem has been graded, it is sent back to the Broker Service who then sends it to the Database Service to be saved to the submission collection in the database.

Getting a Problem

- the client can request a problem from the broker via REST API
- the broker forwards the request to the database service which queries the database and then return it to the broker
- the broker returns the result to the client

Asking for Assistance

- the client can request help on a a problem from the broker via REST API
- the broker forwards the request to the ai assistance service which queries an external machine learning API and return the result to the broker
- the broker returns the result to the client

Scalability and Fault Tolerance

Scalability

The design of the system's architecture significantly enhances its scalability, a very import aspect for a competitive programming platform. We have decoupled the services using the activeMQ message queues, and segregated different functionalities into individual components (e.g., grading services, database). This allows for seamless scale-out of these components in response to increased demand. This modular approach ensures that enhancements or expansions can be made to specific parts of the system without disrupting the entire infrastructure.

The use of ActiveMQ as a message broker is central to this scalability. It efficiently manages the increasing flow of messages, particularly submissions and requests for the machine learning API, ensuring that they are processed promptly, even under high load conditions. This capability is vital in maintaining system responsiveness and reliability.

Another vital aspect of scalability is the potential implementation of load balancing. By evenly distributing incoming requests across multiple server instances, load balancing prevents overloading any single point in the system. This not only enhances the system's ability to handle large volumes of requests but also contributes to a more efficient and faster processing time, improving the overall user experience.

MongoDB is scalable by default due to its distributed architecture and built-in features. It employs sharding, a process of distributing data across multiple machines, allowing the database to handle more data and traffic by spreading the load. As your data grows, you can add more nodes, and MongoDB will automatically balance the data across the cluster. It also supports replica sets, ensuring data is replicated across servers, enhancing availability and fault tolerance. This means if one server fails then others can take over without data loss. Additionally, MongoDB's flexible schema design allows for easy accommodation of changes in data structure. These features make MongoDB inherently scalable, capable of growing with our needs without extensive reconfiguration or downtime.

Fault Tolerance

Fault tolerance is an integral part of the system's robustness, ensuring uninterrupted service even in the face of component failures or unexpected issues. The system employs several strategies to achieve high fault tolerance.

Message queuing, facilitated by the message broker, is a critical feature for maintaining system integrity. In scenarios where a grading service is temporarily unavailable, submissions are not discarded. Instead, they remain in the queue for processing, ensuring no data loss and that each submission is eventually processed. This feature is particularly important for a competitive programming platform where the integrity and preservation of user submissions is key.

Database redundancy and backups for MongoDB further enhance fault tolerance. By ensuring that there are multiple copies of the database and regular backups, the system can quickly recover from any data loss incidents, maintaining data integrity and availability. This aspect is crucial not only for preserving submission data but also for maintaining user trust and platform reliability.

Service isolation, particularly into docker containers, ensures that a problem in one service does not cascade into other areas of the system. This isolation limits the impact of any single point of failure and is a great strategy in maintaining continuous service across the application.

MongoDB achieves fault tolerance via its built in replica sets. Replica sets are a group of MongoDB servers that maintain the same dataset. One node is the primary node while the others are the secondary nodes. This is like a master-slave relationship. If the primary node fails, one of the secondary nodes is automatically elected to be the new primary. This ensures that the database remains available and can continue to accept read and write operations. This process minimizes down time. Even if one server goes down the data is not last since its replicated. This is pivotal for fault tolerance.

Reflections

What were the key challenges you have faced in completing the project? How did you overcome them?

- **Integration Complexity**: Integrating various technologies (Spring Boot, MongoDB, React, etc.) posed significant challenges. Overcoming this required diligent research and collaborative troubleshooting sessions to ensure compatibility and smooth interaction between components.
- **Debugging Distributed Systems**: Identifying and fixing bugs in a distributed system was complex due to the multiple moving parts. We adopted systematic logging and monitoring practices and ensured clear communication within the team to pinpoint and resolve issues.
    a. For example, when trying to import classes from the database service (another spring boot app) into the broker service, an error occurred in which the import could not be found. This did not make sense as the package we were importing existed. However, it turns out a Springboot application cannot be dependent on another Spring Boot application in this context.
    b. It is difficult to keep track of the flow of data without proper logs.
- **Learning Curve**: Each team member had varying familiarity with the technologies used. We overcame this through collaborative learning sessions, sharing resources, and dividing tasks according to individual strengths while still encouraging cross-learning.

What would you have done differently if you could start again?

- **Communication**: Doing the entire project over zoom/discord proved to be more difficult than in person. If we could do the project over again we would have met regularly in person as this would have made our discussions easier.
- **Not Use Java:** The majority of our group is more comfortable in Python and it would have been interesting to learn about building distributed systems in Python such as through using rabbitmq.
- **Leave more time for debugging:** A lot of tasks took a lot longer than expected as we ran into bugs along the way which considerably slowed down progress or put in place roadblocks that kept us from continuing. If we had planned for bugs in our initial time estimates or had given ourselves more generous time estimates it would have been a more relaxed experience.

What have you learnt about the technologies you have used? Limitations? Benefits?

- **Better Documentation**: Maintaining detailed documentation from the start would have streamlined the learning and development process, especially for team members less familiar with certain technologies.
- **React:** Great for building dynamic user interfaces with a vast ecosystem of libraries that make it super simple to create a nice frontend quickly
- **Docker:** Offers excellent consistency in deployment environments and eases scalability, but requires a good understanding of container orchestration and especially network configuration for effective use
- **MongoDB:** Its document-oriented nature is ideal for handling unstructured data and scales well horizontally.

Contributions

Colmán :

- set up Java grading service
- set up communication between grading service and broker

Paul:

- Implemented the Python Grading Service
- Set up the Frontend/client
- Set up the communication between broker and grading services
- Implemented the AI assistance service
- Docker Setup

Max:

- Set up MongoDB
- Set up the Database Service
- Setup data communications between database service and grader services + broker
- Bug Fixes

Dorian

- Broker

Natael

- Broker
- Bug fixes