Algorithm Complexity Analyzer

Sprint Number 4
Date 12/1/24

Name	Email Address
John Gahagan	john.gahagan398@topper.wku.edu
Jay Mistry	jay.mistry627@topper.wku.edu

CS 396
Fall 2024
Project Technical Documentation

Contents

List of Figures

1 Introduction

1.1 Project Overview

The Algorithm Complexity Analyzer is a tool designed to automate the process of analyzing an algorithm's time complexity. It is built using a microservices architecture, which divides the system into separate services to improve modularity and efficiency. The main purpose of this tool is to make analyzing time complexity faster and more accurate which would benefit various people like computer science students, educators, and software developers. The project is made up of three main microservices. The Time Complexity Service, the Recurrence Relation Service, and the Result Presentation Service are these microservices. The Time Complexity Service extracts recurrence relations from the input Java code. These recurrence relations are then analyzed by the Recurrence Relation Service, which calculates the time complexity in Big-O notation. The Result Presentation Service provides a user friendly web interface where users can input their code and view the results in a clear format. This tool solves the common problem of manually analyzing time complexity. This can be difficult, time consuming, and prone to mistakes. By automating this process, the tool ensures accurate results and saves time, allowing users to focus on understanding or improving their code. A key feature of the project is its use of Docker containers, which allow each microservice to run independently. This setup makes debugging, deploying, and scaling easier. The microservices communicate seamlessly, making sure that the system works as a single unit. The tool is designed for students learning algorithms, educators needing reliable teaching tools, and developers who want to analyze and improve their code. The Algorithm Complexity Analyzer bridges theoretical concepts with practical use, helping users understand how efficient their algorithms are. By automating a critical step in algorithm analysis, this tool promotes better learning and more effective software development.

1.2 Technical Requirements

1.2.1 Functional Requirements

Mandatory Functional Requirements
All micro service Docker containers must communicate using network sockets and TCP.
Each micro service must run in its own Docker container and communicate with other containers via defined API endp
The system must accept algorithm descriptions in the form of code snippets to analyze their time complexity.
The time complexity analysis micro service must correctly identify and model recurrence relations from the provided so
The recurrence relation analysis service must support standard recurrence forms, including divide-and- conquer recurre
The result presentation micro service must output the computed time complexity (big-O notation) along with an expla
Extended Functional Requirements
The system must validate the user's code input to ensure it is a valid Java method or algorithm before proceeding with

The system uses microservices to handle specific tasks like extracting recurrence relations from code, solving these relations, and presenting the results in a simple and clear format. Each service plays an important role. Each helping to keep the system modular and scalable. For example, the Big-O determination service simplifies the complex task of analyzing nested loops or recursive calls, while the result presentation service makes the output easy to understand by providing detailed visual feedback. By meeting these functional requirements, the project offers a dependable and practical tool for analyzing algorithm complexity, useful for both educational and professional purposes.

1.2.2 Non-Functional Requirements

Performance and scalability are important in order to handle different input sizes and multiple users at the same time. Using microservices and Docker ensures efficient resource management and allows each service to be updated or deployed independently. Security is maintained through input validation and container isolation,

Mandatory Non-Functional Requirements
The entire project should be deployable on any operating system with Docker installed, using at least three Docker con
The results must be presented in a clear and structured format, with big-O notation displayed prominently, followed by
Error messages should be clear and concise, helping users identify and correct any issues.
Input code files should range from 50 to 500 lines and should complete the commenting process within five seconds.
The deployment of the Algorithm Complexity Analyzer microservice containers must be automated using Infrastructure
Extended Non-Functional Requirements
The system should provide clear error messages at every stage we use the SLF4J Java Logging API for help with error

which prevent malicious inputs or unauthorized access which would help keep the system secure. Reliability is enhanced by automated testing and containerized environments, which reduce errors during deployment and minimize downtime. Usability is also a important, with a simple, responsive web interface and clear feedback to improve the user experience. These non functional requirements ensure the tool is not only functional but also reliable, secure, and efficient for real world use cases.

2 Project Modeling and Design

The overall architecture of the Algorithm Complexity Analyzer is built around a microservices model, leveraging modularity and scalability to ensure efficient analysis of algorithm complexity. The system comprises three primary microservices: a service for determining the Big O notation of an algorithm, a service for analyzing recurrence relations, and a service for presenting results. Each microservice operates independently within its own Docker container, enabling seamless integration and deployment. The System Architecture utilizes RESTful APIs to facilitate communication between microservices. Each service is responsible for a specific task, and their interactions are coordinated through clearly defined API contracts. Docker Compose is used to manage containers, ensuring consistent environments for all services. The result presentation service acts as the entry point for user interaction, forwarding requests to the appropriate backend services, and consolidating the responses. The design of the system was heavily influenced by the need for scalability, maintainability, and clarity in development. SparkJava was chosen as the framework for building RESTful APIs due to its simplicity and lightweight nature. FreeMarker templates power the result presentation service, offering a clean interface for rendering views. JavaParser is employed in the Big O and recurrence relation services to accurately parse user-submitted Java code, extracting the necessary components for analysis. Challenges during design included managing the parsing of complex user-provided code and ensuring accurate communication between services. These challenges were addressed through rigorous testing and the use of well documented libraries, such as JavaParser and Gson, for JSON parsing and data exchange. This architecture not only isolates concerns across the system but also allows for future extensibility. For instance, new microservices could be added to support additional types of algorithm analysis without impacting the existing system. The design reflects a balance between simplicity and functionality, tailored to meet the requirements of analyzing algorithm complexity in a user-friendly and efficient manner.

2.1 System Architecture

The team uses a microservices based architecture designed for modularity, scalability, and clear separation of concerns. At its core, the system consists of three distinct microservices: the Time Complexity Service, the Recurrence Relation Service, and the Result Presentation Service. Each service operates independently within a Docker container, ensuring isolation, reproducibility, and ease of deployment. The architecture is based on RESTful APIs that facilitate communication between the services. This approach ensures that each microservice can be developed, tested, and deployed independently. The Docker containers ensure consistency across different environments, making the system highly portable and reliable. The system also includes health-check endpoints (e.g., /health) for monitoring the status of each service. These endpoints allow developers to quickly identify and resolve issues, ensuring smooth operation of the application. In terms of data flow, the user interacts with the system via the Result Presentation Service, which serves as the entry point. User-submitted code is sent to the Time Complexity Service to extract a recurrence relation. This relation is then passed to the Recurrence Relation Service to compute the Big-O complexity. The results are sent back to the Result Presentation Service for display.

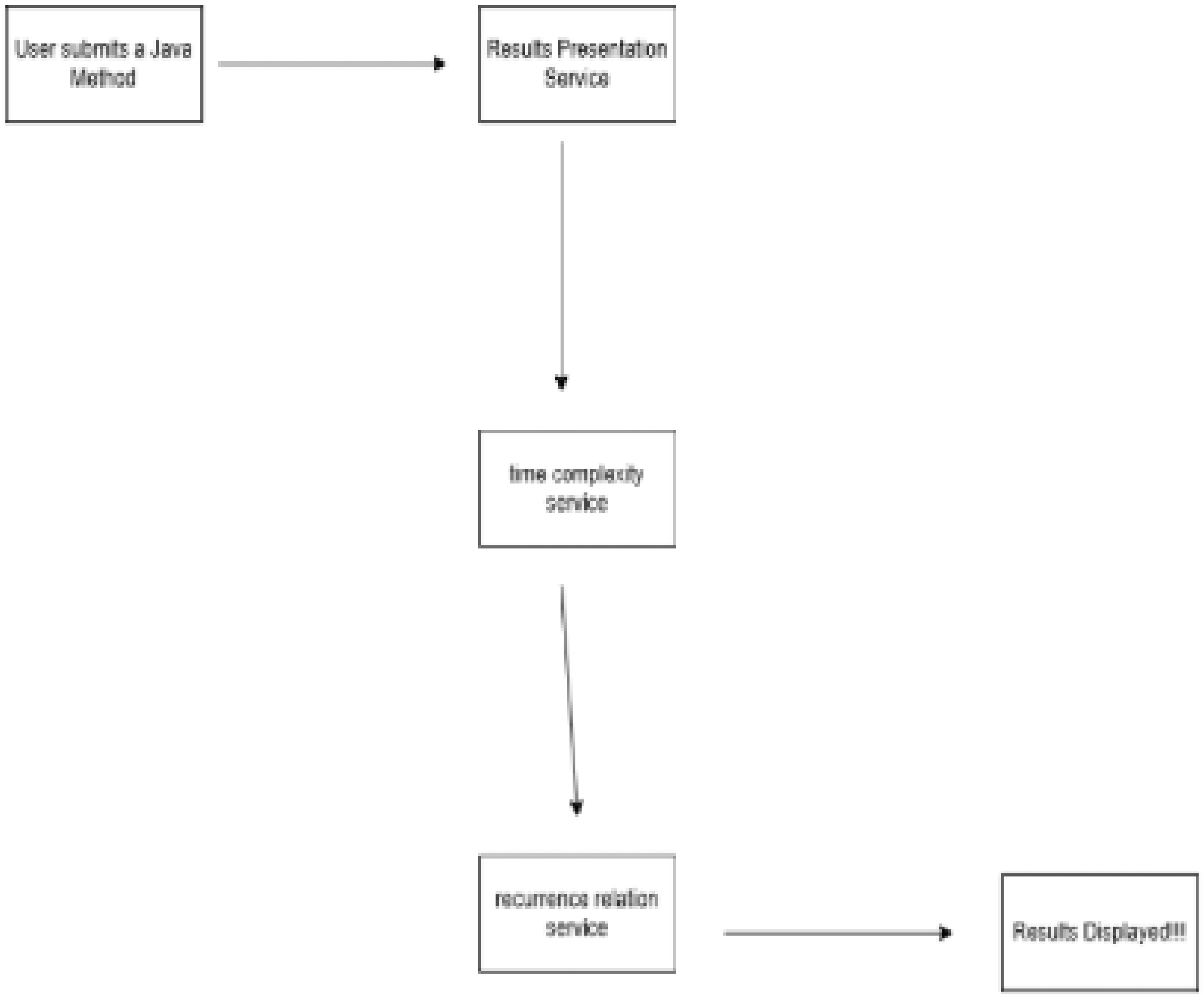

User submits a Java Method
Results Presentation Service
time complexity service
recurrence relation service
Results Displayed!!!

Figure 1: Overall Project Pipeline Output: A visual representation of the interaction between microservices.

This architecture is designed to support modularity, enabling future scalability and ease of maintenance.

2.2 Defined Microservices

The system is built with three microservices, each having a specific role, input/output requirements, and interactions with other services. The Time Complexity Service analyzes the input code from the user and extracts the recurrence relation that describes its time complexity. This service accepts Java code as input through a RESTful API and uses JavaParser and custom logic to process the code. It outputs the recurrence relation in a standard mathematical format for the next service to use. The Recurrence Relation Service receives the recurrence relation from the Time Complexity Service and calculates its time complexity in Big-O notation. This service employs mathematical solvers and algorithms to process recurrence relations and returns the Big-O complexity as output. It communicates with the Time Complexity Service via RESTful calls and interacts with the Result Presentation Service to deliver its output. The Result Presentation Service serves as the system's user interface.

It allows users to input their code and view analysis results. This service interacts with both the Time Complexity Service and the Recurrence Relation Service to gather results. It uses SparkJava and FreeMarker to render the web pages, providing a smooth user experience. These microservices function independently but work together to ensure clear separation of tasks, scalability, and easy debugging. Their RESTful communication supports flexibility for expanding or modifying components without affecting the entire system.

2.3 Development Considerations

The architecture and design of the project involved several key decisions and trade-offs to ensure functionality, scalability and maintainability. One primary consideration was adopting a microservices architecture. This decision isolated responsibilities into distinct services, allowing each service to be developed, tested and deployed independently. It also enables scalability as individual services can be scaled based on demand. The choice of Docker containers addressed the need for consistent deployment across various environments. Docker ensures that each microservice runs in its own isolated environment with necessary dependencies, minimizing compatibility issues. This also simplifies testing and deploying updates. For programming languages and frameworks, Java was chosen for its robustness and extensive ecosystem. SparkJava was selected as the lightweight framework for building RESTful APIs due to its simplicity and effectiveness for smaller projects. FreeMarker was used for rendering the front-end views as it integrates well with SparkJava and offers a straightforward way to manage templates. Key trade-offs included balancing simplicity with functionality. While more advanced tools like Kubernetes could have been used for container orchestration, they were deemed unnecessary for this project's scope. Instead, Docker Compose was used to manage multi-container deployments, providing sufficient functionality without added complexity. Challenges included parsing user-provided Java code, which required precise handling to extract valid recurrence relations. JavaParser was chosen to leverage an established library, though it required careful integration to meet the system's needs. The system's reliance on RESTful APIs ensures modularity but introduced the challenge of managing inter-service communication. To address this, clear API contracts were defined and extensive testing verified seamless interactions between services. These decisions collectively shaped an efficient, maintainable system well-suited for analyzing algorithm complexity in a modular way.

3 Implementation Approach

3.1 Software Process Model

The development of the Algorithm Complexity Analyzer followed an Agile methodology. This approach allowed the team to divide the project into manageable sprints, with frequent reviews and incremental improvements. Agile's flexibility ensured that the system's requirements could adapt to new challenges or feedback as development progressed. Regular team meetings and check ins facilitated collaboration and enabled the team to prioritize tasks effectively. User stories and backlog tracking were used to focus on delivering functional components in each sprint, ensuring progress toward the final goal.

3.2 Key Algorithms and Techniques Used

The project employs several key algorithms and techniques to achieve its goals. Master's Theorem was used to solve recurrence relations, providing an efficient method to determine time complexity. The JavaParser library was utilized to parse user-submitted Java code, enabling the system to extract recurrence relations and identify the structure of algorithms. Additionally, RESTful APIs were designed using SparkJava, ensuring smooth communication between microservices. These techniques were crucial for creating a modular and efficient system capable of handling complex inputs.

3.3 Microservices and Docker Usage

The implementation relies on a microservices architecture, where each component operates as an independent service. The system includes three services, the time complexity service, recurrence relation service, and result presentation service.For the time complexity service, it extracts recurrence relations from user-submitted code. For the recurrence relation service, it solves the recurrence relations to determine Big-O complexity. For the result presentation service, it provides the user interface and consolidates results. Each microservice is deployed within its own Docker container, ensuring isolation, consistent environments, and easy deployment. Docker Compose was used to manage and orchestrate these containers, allowing for seamless integration and testing across different services. This modular approach simplifies development and debugging while enabling scalability and maintenance.

3.4 Container Network Communication Setup and Testing

The containers communicate over a shared Docker network, established using Docker Compose. RESTful APIs are used for communication between services. Each service is exposed on a specific port (e.g., 5000, 5001, 5002), and requests are routed through these ports. Testing involved verifying the end-to-end communication flow. Requests from the user interface were sent to the appropriate back end service. Responses were validated for accuracy and consistency. Health-check endpoints (e.g., /health) were monitored to ensure all services were operational. Integration testing was performed by simulating various workflows, ensuring that all microservices communicated seamlessly.

3.5 Infrastructure as Code approach

The project uses Docker Compose as an Infrastructure as Code (IaC) tool to define and manage the system's deployment architecture. The docker-compose.yml file specifies the services, networks and volumes required for the application. This approach ensures the entire system can be set up consistently across different environments with minimal manual configuration. Using Docker Compose allowed rapid deployment and teardown of the system for testing or updates. It provided portability across machines and platforms and a version controlled infrastructure setup that supports reproducibility and collaboration. By defining the infrastructure as code, the team ensured deployment was automated and error free, reducing downtime and simplifying scalability.

4 Software Testing and Results

—

4.1 Software Testing with Visualized Results

Test Plan Identifier: MicroserviceHealthCheck

Introduction: This test ensures that all microservices, including the Time Complexity Service, Recurrence Relation Service, and Result Presentation Service, are operational by verifying the `/health` endpoints for each service.

Test item: The system under test includes the three microservices. Each service must respond with a confirmation of operational status when accessed via its respective `/health` endpoint.

Features to test/not to test: This test focuses on verifying that the `/health` endpoints confirm that the services are running. It does not test performance or security aspects of these endpoints, as these are outside the current scope.

Approach: The `/health` endpoints of the microservices are accessed via their respective localhost ports (e.g., 5000, 5001, and 5002). The response from each endpoint is validated to confirm that it indicates the service is operational.

Test deliverables: The deliverables for this test include results showing the operational status of each microservice and screenshots demonstrating the output of the `/health` endpoints.

Item pass/fail criteria: The test passes if the `/health` endpoint response states "Service is running." If the response does not indicate the expected operational status, the test fails.

Environmental needs: The test requires Docker containers running locally, with microservices exposed on ports 5000, 5001, and 5002.

Responsibilities: Developers and testers are responsible for ensuring the services are running as expected and resolving any issues identified during the test.

Staffing and training needs: This test does not require additional training.

Schedule: The test is conducted during the integration testing phase as part of the deployment timeline.

Risks and Mitigation: Docker containers may fail to start due to configuration issues. Using `docker-compose logs` can help identify and resolve these errors before the test is conducted.

Approvals: The project lead reviews and approves the test results to ensure they comply with system requirements.

localhost:5000/health

Figure 2: Health Check Results for Each Microservice at `/health` Endpoints.

—

Test Plan Identifier: TimeComplexityAnalyzeTest

Introduction: This test ensures that the `/analyze` endpoint in the Time Complexity Service can process Java code and correctly calculate its time complexity in Big-O notation.

Test item: The Time Complexity Service's `/analyze` endpoint processes Java code to determine the time complexity for a given algorithm.

Features to test/not to test: The focus of this test is to verify the accuracy of the Big-O notation calculated by the `/analyze` endpoint. Other performance metrics, such as latency, are not part of this test.

Approach: A Java implementation of Merge Sort is submitted to the `/analyze` endpoint. The system processes the code and calculates the time complexity $O(n \log n)$, which is validated against the expected result.

Test deliverables: Deliverables include test results showing input Java code, calculated time complexity, and screenshots of the endpoint's response.

Item pass/fail criteria: The test passes if the output matches the expected time complexity $O(n \log n)$. It fails if the output is incorrect or incomplete.

Environmental needs: Docker containers with the Time Complexity Service must be running and accessible on port 5000.

Responsibilities: Developers and testers are responsible for executing the test and addressing any discrepancies.

Risks and Mitigation: Submission of invalid Java code may result in errors. This can be mitigated by implementing robust input validation.

Approvals: The test results are reviewed and approved by the project lead.

Time Complexity: O(n log n)

Figure 3: Time Complexity: $O(n \log n)$.

—

Test Plan Identifier: RecurrenceRelationSolveTest

Introduction: This test validates the `/solve` endpoint in the Recurrence Relation Service, ensuring it accurately extracts and represents recurrence relations from the input Java code.

Test item: The Recurrence Relation Service's `/solve` endpoint processes recurrence relations extracted from Java code and verifies their accuracy.

Features to test/not to test: The focus of this test is to verify the correctness of the recurrence relation extracted by the `/solve` endpoint. Performance metrics, such as execution time, are not part of this test.

Approach: A recurrence relation extracted from the Time Complexity Service is submitted to the `/solve` endpoint. The system processes it and outputs the recurrence relation $T(n) = 2T(n/2) + O(n)$, which is validated against expected results.

Test deliverables: Deliverables include input recurrence relations, extracted recurrence relations, and screenshots of the endpoint's response.

Item pass/fail criteria: The test passes if the output matches $T(n) = 2T(n/2) + O(n)$. It fails if the recurrence relation is incomplete or incorrect.

Environmental needs: Docker containers with the Recurrence Relation Service must be running on port 5001.

Responsibilities: Developers and testers are responsible for conducting the test and resolving any issues.

Risks and Mitigation: Errors due to invalid input recurrence relations can be mitigated by implementing robust input validation.

Approvals: The test results are reviewed and approved by the testing team.

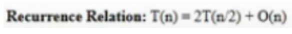

Recurrence Relation: T(n) = 2T(n/2) + O(n)

Figure 4: Recurrence Relation: $T(n) = 2T(n/2) + O(n)$.

—

Test Plan Identifier: ResultPresentationDisplayTest

Introduction: This test verifies the functionality of the `/display` endpoint in the Result Presentation Service, ensuring it consolidates outputs into a user-friendly format.

Test item: The Result Presentation Service's `/display` endpoint displays consolidated results from other microservices.

Features to test/not to test: The focus is on the accuracy and clarity of displayed results. Aesthetic details are out of scope.

Approach: Test inputs which are recurrence relation and time complexity, are provided to the endpoint, and the consolidated output is compared to expected results.

Test deliverables: Deliverables include test inputs, output format, and screenshots of the displayed results.

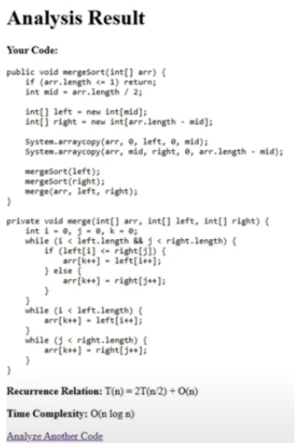
Item pass/fail criteria: The test passes if results are accurate and well-presented.

Environmental needs: Docker containers with the Result Presentation Service must be running on port 5002.

Responsibilities: Developers and testers ensure the display outputs meet expectations.

Risks and Mitigation: Errors from upstream services can be mitigated through comprehensive prior testing.

Approvals: The test results are reviewed and signed off by the testing team.


Analysis Result

Your Code:

```
public void mergeSort(int[] arr) {
    if (arr.length <= 1) return;
    int mid = arr.length / 2;

    int[] left = new int[mid];
    int[] right = new int[arr.length - mid];

    System.arraycopy(arr, 0, left, 0, mid);
    System.arraycopy(arr, mid, right, 0, arr.length - mid);

    mergeSort(left);
    mergeSort(right);
    merge(arr, left, right);
}

private void merge(int[] arr, int[] left, int[] right) {
    int i = 0, j = 0, k = 0;
    while (i < left.length && j < right.length) {
        if (left[i] <= right[j]) {
            arr[k++] = left[i++];
        } else {
            arr[k++] = right[j++];
        }
    }
    while (i < left.length) {
        arr[k++] = left[i++];
    }
    while (j < right.length) {
        arr[k++] = right[j++];
    }
}
```

Recurrence Relation: T(n) = 2T(n/2) + O(n)

Time Complexity: O(n log n)

Figure 5: Merge Sort Example Output Showing Correct Results.

—

Microservice	**Health Check Path**	**Functional Test Path**
Time Complexity Service	`/health`	`/analyze`
Recurrence Relation Service	`/health`	`/solve`
Result Presentation Service	`/health`	`/display`

Table 1: Paths for Testing Microservices

5 Conclusion

The Algorithm Complexity Analyzer was created to provide a comprehensive tool for analyzing the time complexity of algorithms while focusing on modularity, scalability, and efficiency. The system uses a microservices architecture divided into the Time Complexity Service, the Recurrence Relation Service, and the Result Presentation Service. Each service runs in its own Docker container and communicates through RESTful APIs. This design ensures smooth integration and independent functionality. The purpose of the project was to simplify understanding algorithmic performance and make it accessible to students, educators, and software developers. Despite the success of the system, there are limitations. The JavaParser library, while effective, struggles with very complex or unconventional code, leading to possible errors. Additionally, the system currently only supports Java code, limiting its use for other programming languages. Docker Compose is sufficient for this project but may not be suitable for larger systems that need advanced deployment management. Future improvements will address these shortcomings by expanding support to more programming languages and enhancing parsing capabilities to handle a broader range of inputs. Using Kubernetes for container management can improve scalability. Adding advanced visualizations and customizable analysis features will make the tool more user friendly and versatile. These updates will ensure the tool remains practical and valuable for algorithm complexity analysis.

6 Appendix

6.1 Software Product Build Instructions

To set up the **Algorithm Complexity Analyzer** on a new computer for continued development, follow these steps:

1. Prerequisites:

- Ensure Java Development Kit (JDK) version 11 or higher is installed.
- Install Docker and Docker Compose.
- Install Gradle (if not already installed).
- Clone the GitHub repository for the project.

2. Cloning the Repository:

- Open a terminal and navigate to the desired directory for storing the project.
- Run the command:

```
git clone <repository_url>
```

Replace `<repository_url>` with the URL of the project repository.

3. Setting Up Microservices:

- Navigate to each microservice folder:

```
cd path/to/microservice
```

- Build the microservice using Gradle:

```
gradle clean build
```

4. Docker Setup:

- Navigate to the root of the project directory where the `docker-compose.yml` file is located.
- Build and start the Docker containers:

```
docker-compose up --build
```

This will set up and start all microservices in their respective containers.

5. Environment Configuration:

- Ensure that each service has access to the required ports:
  - Time Complexity Service: `5000`
  - Recurrence Relation Service: `5001`
  - Result Presentation Service: `5002`
- Modify any configuration files if required (e.g., `application.properties` or environment variables) to match your environment.

6. Testing and Verification:

- Verify that all services are running:
  - Access the `/health` endpoint of each service to confirm they are operational:

    ```
    curl http://localhost:<port>/health
    ```

    Replace `<port>` with the appropriate service port (e.g., 5000, 5001, or 5002).

7. Accessing the Application:

- Open a browser and navigate to the result presentation service:

```
http://localhost:5002/
```

- Use the interface to input code and analyze algorithm complexity.

8. Development Workflow:

- To make changes, modify the relevant microservice source code.
- Rebuild the specific microservice:

```
gradle clean build
docker-compose up --build
```

6.2 Software Product User Guide

This section provides a comprehensive guide for both general users and administrative users to effectively utilize the **Algorithm Complexity Analyzer**.

General User Guide: The **Algorithm Complexity Analyzer** is designed to analyze the time complexity of algorithms through a simple and intuitive web interface.

Accessing the System:

- Open a web browser and navigate to the following URL:

```
http://localhost:5002/
```

- You will be greeted with a form labeled `Enter Your Code`.

Using the Interface:

- Input your Java code into the provided text box. Ensure the code contains a valid method.
- Click the `Analyze` button to initiate the time complexity analysis.
- View the results displayed on the next page, which include the recurrence relation and the Big-O notation for your algorithm.

Understanding the Results:

- The results will indicate the extracted recurrence relation and the calculated time complexity in Big-O notation.
- If an error occurs (e.g., invalid Java code), the system will display an error message prompting you to correct your input.

Administrative User Guide: Administrative users are responsible for maintaining the application and monitoring its performance.

Monitoring System Health:

- Check the health of individual microservices by accessing their `/health` endpoints:

  ```
  http://localhost:<port>/health
  ```

  Replace `<port>` with the port of the desired service (e.g., 5000, 5001, or 5002).
- A response of `Service is running!` indicates that the service is operational.

Managing Docker Containers:

- Use Docker to manage the microservices:
  - Start all services:

    ```
    docker-compose up
    ```
  - Stop all services:

    ```
    docker-compose down
    ```

Updating the Application:

- Pull the latest updates from the GitHub repository:

  ```
  git pull origin main
  ```
- Rebuild the Docker containers to apply changes:

  ```
  docker-compose up --build
  ```

Error Handling:

- Monitor logs to identify issues:

  ```
  docker logs <container_name>
  ```

  Replace `<container_name>` with the name of the service container.

- Restart specific services if needed:

  ```
  docker-compose restart <service_name>
  ```

6.3 Source Code with Comments

GitHub Repository: `https://github.com/SaviorFs/Algorithm-Complexity-Microservices`

C.1 Time Complexity Service

C.1.1 App.java

```
package com.example;

import static spark.Spark.*;
import com.google.gson.Gson;
import com.github.javaparser.StaticJavaParser;
import com.github.javaparser.ast.CompilationUnit;
import com.github.javaparser.ast.body.MethodDeclaration;
import java.util.HashMap;
import java.util.Map;

public class App {
    public static void main(String[] args) {
        port(5000);

        // Health-check route
        get("/health", (req, res) -> "Service is running!");

        // Analyze code for time complexity
        post("/analyze", (req, res) -> {
            res.type("application/json");
            Map<String, String> result = new HashMap<>();
            try {
                Map<String, String> data = new Gson().fromJson(req.body(), Map.class);
                if (data == null || !data.containsKey("code")) {
                    throw new IllegalArgumentException("Request must contain 'code' field.");
                }

                String code = data.get("code");
                if (code == null || code.isEmpty()) {
                    throw new IllegalArgumentException("Code cannot be null or empty.");
                }

                // Extract recurrence relation from the provided code
```

```
                String recurrenceRelation = extractRecurrenceRelation(code);
                result.put("recurrence_relation", recurrenceRelation);
                return new Gson().toJson(result);
            } catch (Exception e) {
                res.status(400);
                result.put("error", e.getMessage());
                return new Gson().toJson(result);
            }
        });
    }

    public static String extractRecurrenceRelation(String code) throws Exception {
        try {
            // Wrap the code in a temporary class if it's not a full class
            String wrappedCode = "class TempClass { " + code + " }";
            CompilationUnit cu = StaticJavaParser.parse(wrappedCode);

            MethodDeclaration method = cu.getClassByName("TempClass")
                .orElseThrow(() -> new Exception("No class found"))
                .getMethods().get(0);

            // Example: Replace with actual recurrence extraction logic
            return "T(n) = 2T(n/2) + O(n)";
        } catch (Exception e) {
            throw new Exception("Parse error: " + e.getMessage(), e);
        }
    }
}
```

C.1.2 Dockerfile

```
# Build stage
FROM maven:3.8.4-openjdk-11 AS build
WORKDIR /app

# Copy pom.xml and download dependencies
COPY pom.xml .
RUN mvn dependency:go-offline -B

# Copy the source code
COPY src ./src

# Build the project
RUN mvn clean package -DskipTests

# Run stage
FROM openjdk:11-jre-slim
WORKDIR /app

# Copy the packaged JAR from the build stage
COPY --from=build /app/target/time_complexity_service-1.0-SNAPSHOT.jar app.jar

# Expose the port your application runs on
```

```
EXPOSE 5000

# Healthcheck for the service
HEALTHCHECK --interval=30s --timeout=10s CMD curl --fail http://localhost:5000/health || exit 1

# Run the application
ENTRYPOINT ["java", "-jar", "app.jar"]
```

C.1.3 build.gradle

```
plugins {
    id 'java'
    id 'application'
}

group 'com.example.timecomplexity'
version '1.0-SNAPSHOT'

repositories {
    mavenCentral()
}

dependencies {
    // SparkJava for web framework
    implementation 'com.sparkjava:spark-core:2.9.4'

    // Gson for JSON parsing
    implementation 'com.google.code.gson:gson:2.8.9'

    // SLF4J for logging
    implementation 'org.slf4j:slf4j-simple:1.7.36'

    // JavaParser for parsing Java code
    implementation 'com.github.javaparser:javaparser-core:3.25.4'
}

application {
    mainClass = 'com.example.App'
}

jar {
    manifest {
        attributes(
            'Main-Class': application.mainClass
        )
    }
    from {
        configurations.runtimeClasspath.collect { it.isDirectory() ? it : zipTree(it) }
    }
    duplicatesStrategy = DuplicatesStrategy.EXCLUDE
}
```

C.2 Recurrence Relation Service

C.2.1 App.java

```
package com.example;

import static spark.Spark.*;
import com.google.gson.Gson;
import java.util.HashMap;
import java.util.Map;

public class App {
    public static void main(String[] args) {
        port(5001);

        // Health-check route
        get("/health", (req, res) -> "Service is running!");

        // Solve recurrence relation
        post("/solve", (req, res) -> {
            res.type("application/json");
            Map<String, String> result = new HashMap<>();
            try {
                Map<String, String> data = new Gson().fromJson(req.body(), Map.class);
                if (data == null || !data.containsKey("recurrence_relation")) {
                    throw new IllegalArgumentException("Request must contain 'recurrence_relation' fi
                }

                String recurrence = data.get("recurrence_relation");
                if (recurrence == null || recurrence.isEmpty()) {
                    throw new IllegalArgumentException("Recurrence relation cannot be null or empty.'
                }

                String bigO = solveRecurrence(recurrence);
                result.put("big_o", bigO);
                return new Gson().toJson(result);
            } catch (Exception e) {
                res.status(400);
                result.put("error", e.getMessage());
                return new Gson().toJson(result);
            }
        });
    }

    private static String solveRecurrence(String recurrence) throws Exception {
        if (recurrence.contains("T(n/2)")) {
            return "O(n log n)";
        }
        return "O(n)";
    }
}
```

C.2.2 Dockerfile

```
# Build stage
```

```
FROM maven:3.8.4-openjdk-11 AS build
WORKDIR /app

# Copy the Maven POM file
COPY pom.xml .

# Download dependencies to cache them
RUN mvn dependency:go-offline -B

# Copy the source code
COPY src ./src

# Build the project
RUN mvn clean package -DskipTests

# Run stage
FROM openjdk:11-jre-slim
WORKDIR /app

# Copy the packaged JAR from the build stage
COPY --from=build /app/target/recurrence_relation_service-1.0-SNAPSHOT.jar app.jar

# Expose the port the service runs on
EXPOSE 5001

# Healthcheck for service availability
HEALTHCHECK --interval=30s --timeout=10s CMD curl --fail http://localhost:5001/health || exit 1

# Run the application
ENTRYPOINT ["java", "-jar", "app.jar"]
```

C.2.3 build.gradle

```
plugins {
    id 'java'
    id 'application'
}

group 'com.example.recurrencerelation'
version '1.0-SNAPSHOT'

sourceCompatibility = '11'

repositories {
    mavenCentral()
}

dependencies {
    // SparkJava for web framework
    implementation 'com.sparkjava:spark-core:2.9.4'

    // Gson for JSON parsing
    implementation 'com.google.code.gson:gson:2.8.9'
    // SLF4J Simple binding for logging
    implementation 'org.slf4j:slf4j-simple:1.7.36'

    // Apache Commons Math for mathematical operations
    implementation 'org.apache.commons:commons-math3:3.6.1'
}

application {
    mainClass = 'com.example.App'
}

jar {
    manifest {
        attributes(
            'Main-Class': application.mainClass
        )
    }
    from {
        configurations.runtimeClasspath.collect { it.isDirectory() ? it : zipTree(it) }
    }
    duplicatesStrategy = DuplicatesStrategy.EXCLUDE
}
```

C.3 Result Presentation Service

C.3.1 App.java

```
package com.example;

import static spark.Spark.*;
import com.google.gson.Gson;
import freemarker.template.Configuration;
import freemarker.template.TemplateExceptionHandler;
import spark.ModelAndView;
import spark.template.freemarker.FreeMarkerEngine;

import java.util.HashMap;
import java.util.Map;

public class App {
    public static void main(String[] args) {
        port(5002);

        // Health-check route
        get("/health", (req, res) -> "Service is running!");

        // Root route to render the form
        get("/", (req, res) -> {
            Map<String, Object> model = new HashMap<>();
            return createFreeMarkerEngine().render(new ModelAndView(model, "index.ftl"));
        });

        // Handle form submission
        post("/analyze", (req, res) -> {
            Map<String, Object> model = new HashMap<>();
            try {
                String code = req.queryParams("code");
                model.put("code", code);

                // Call Time Complexity Service
                String recurrenceRelation = getRecurrenceRelation(code);

                // Call Recurrence Relation Service
                String bigO = solveRecurrence(recurrenceRelation);

                // Populate model with results
                model.put("recurrence_relation", recurrenceRelation);
                model.put("big_o", bigO);

                return createFreeMarkerEngine().render(new ModelAndView(model, "result.ftl"));
            } catch (Exception e) {
                model.put("error", e.getMessage());
                return createFreeMarkerEngine().render(new ModelAndView(model, "index.ftl"));
            }
        });
    }

    private static String getRecurrenceRelation(String code) throws Exception {
        Map<String, String> payload = new HashMap<>();
        payload.put("code", code);
        String response = HttpClient.post("http://localhost:5000/analyze", new Gson().toJson(payload)

        Map<String, String> result = new Gson().fromJson(response, Map.class);
        if (result.containsKey("recurrence_relation")) {
            return result.get("recurrence_relation");
        } else if (result.containsKey("error")) {
            throw new Exception("Time Complexity Service Error: " + result.get("error"));
        } else {
            throw new Exception("Unexpected response from Time Complexity Service");
        }
    }

    private static String solveRecurrence(String recurrence) throws Exception {
        Map<String, String> payload = new HashMap<>();
        payload.put("recurrence_relation", recurrence);
        String response = HttpClient.post("http://localhost:5001/solve", new Gson().toJson(payload));

        Map<String, String> result = new Gson().fromJson(response, Map.class);
        if (result.containsKey("big_o")) {
            return result.get("big_o");
        } else if (result.containsKey("error")) {
            throw new Exception("Recurrence Relation Service Error: " + result.get("error"));
        } else {
            throw new Exception("Unexpected response from Recurrence Relation Service");
        }
    }
    private static FreeMarkerEngine createFreeMarkerEngine() {
        Configuration configuration = new Configuration(Configuration.VERSION_2_3_31);
        configuration.setClassForTemplateLoading(App.class, "/templates");
        configuration.setDefaultEncoding("UTF-8");
        configuration.setTemplateExceptionHandler(TemplateExceptionHandler.RETHROW_HANDLER);
        return new FreeMarkerEngine(configuration);
    }
}
```

C.3.2 HttpClient.java

```
package com.example;

import java.io.OutputStream;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.Scanner;

public class HttpClient {
    public static String post(String urlString, String jsonPayload) throws Exception {
        URL url = new URL(urlString);
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setDoOutput(true);
        conn.setRequestMethod("POST");
        conn.setRequestProperty("Content-Type", "application/json");

        // Set timeouts
        conn.setConnectTimeout(5000); // Connection timeout: 5 seconds
        conn.setReadTimeout(5000);    // Read timeout: 5 seconds

        try (OutputStream os = conn.getOutputStream()) {
            os.write(jsonPayload.getBytes());
            os.flush();
        }

        int responseCode = conn.getResponseCode();

        try (Scanner scanner = new Scanner(
                responseCode == 200 ? conn.getInputStream() : conn.getErrorStream(),
                "UTF-8")) {
            String response = scanner.useDelimiter("\\A").next();
            if (responseCode != 200) {
                throw new RuntimeException("HTTP error code: " + responseCode + " Response: " + respo
            }
            return response;
        }
    }
}
```

C.3.3 index.ftl

```
<!DOCTYPE html>
<html>
```

```
<head>
    <title>Time Complexity Analyzer</title>
</head>
<body>
    <h1>Enter Your Code</h1>
    <form action="/analyze" method="post">
        <textarea name="code" rows="10" cols="50"></textarea><br>
        <input type="submit" value="Analyze">
    </form>
    <#if error??>
        <p style="color:red;">Error: ${error}</p>
    </#if>
</body>
</html>
```

C.3.4 result.ftl

```
<!DOCTYPE html>
<html>
<head>
    <title>Analysis Result</title>
</head>
<body>
    <h1>Analysis Result</h1>
    <p><strong>Your Code:</strong></p>
    <pre>${code}</pre>
    <p><strong>Recurrence Relation:</strong> ${recurrence_relation}</p>
    <p><strong>Time Complexity:</strong> ${big_o}</p>
    <a href="/">Analyze Another Code</a>
</body>
</html>
```

C.3.5 Dockerfile

```
# Build stage
FROM gradle:7.6.1-jdk11 AS build
WORKDIR /app
COPY . /app
RUN gradle clean build --no-daemon

# Run stage
FROM openjdk:11-jre-slim
WORKDIR /app
COPY --from=build /app/build/libs/result_presentation_service-1.0-SNAPSHOT.jar app.jar
EXPOSE 5002
ENTRYPOINT ["java", "-jar", "app.jar"]
```

C.3.6 build.gradle

```
plugins {
    id 'java'
    id 'application'
}
```

```
group 'com.example.resultpresentation'
version '1.0-SNAPSHOT'

sourceCompatibility = '11'

repositories {
    mavenCentral()
}

dependencies {
    // SparkJava for web framework
    implementation 'com.sparkjava:spark-core:2.9.4'

    // Gson for JSON parsing
    implementation 'com.google.code.gson:gson:2.8.9'

    // SLF4J Simple binding for logging
    implementation 'org.slf4j:slf4j-simple:1.7.36'

    // FreeMarker for rendering HTML templates
    implementation 'org.freemarker:freemarker:2.3.31'
    implementation 'com.sparkjava:spark-template-freemarker:2.7.1'
}

application {
    mainClass = 'com.example.App'
}

jar {
    manifest {
        attributes(
            'Main-Class': application.mainClass
        )
    }
    from {
        configurations.runtimeClasspath.collect { it.isDirectory() ? it : zipTree(it) }
    }
    duplicatesStrategy = DuplicatesStrategy.EXCLUDE
}
```

C.4 Docker Compose File

```
version: '3.8'
services:
  time_complexity_service:
    build:
      context: ./time_complexity_service
      dockerfile: Dockerfile
    ports:
      - "5000:5000"
    restart: always

  recurrence_relation_service:
    build:
      context: ./recurrence_relation_service
      dockerfile: Dockerfile
    ports:
      - "5001:5001"
    restart: always

  result_presentation_service:
    build:
      context: ./result_presentation_service
      dockerfile: Dockerfile
    ports:
      - "5002:5002"
    restart: always
    depends_on:
      - time_complexity_service
      - recurrence_relation_service
```