Complexity Analyzer

Sprint 3
10/26/2025

Name	**Email Address**
Ryerson Brower	ryerson.brower178@topper.wku .edu
Aaron Downing	aaron.downing652@topper.wku.edu

CS 396
Fall 2024
Project Technical Documentation

Contents

1 Introduction **1**
1.1 Project Overview . . . 1
1.2 Project Scope . . . 1
1.3 Technical Requirements . . . 1
1.3.1 Functional Requirements . . . 1
1.3.2 Non-Functional Requirements . . . 2

2 Project Modeling and Design **3**
2.1 System Architecture . . . 3
2.2 Defined Microservices . . . 3
2.3 Development Considerations . . . 3

3 Implementation Approach **3**
3.1 Software Process Model . . . 3
3.2 Key Algorithms and Techniques Used . . . 3
3.3 Microservices and Docker Usage . . . 4
3.4 Container Network Communication Setup and Testing . . . 4
3.5 Infrastructure as Code approach . . . 4

4 Software Testing and Results **4**
4.1 Software Testing with Visualized Results . . . 4

5 Conclusion **4**

6 Appendix **5**
6.1 Software Product Build Instructions . . . 5
6.2 Software Product User Guide . . . 5
6.3 Source Code with Comments . . . 5

List of Figures

1 Introduction

1.1 Project Overview

The Algorithm Complexity Analyzer project aims to develop a microservice-based system that automates the evaluation of algorithmic efficiency through Big-O and recurrence relation analysis. The project provides a scalable platform that decomposes the complexity analysis process into independent services, each responsible for a specific function such as time complexity detection, recurrence modeling, and result presentation. By using Docker containers, the system ensures cross-platform compatibility, simple deployment, and reliable communication between services.

The primary purpose of the project is the simplify the process of analyzing algorithmic time complexity, which is often performed manually and can often times be done wrong. Developers and students frequently struggle to derive Big-O notation from algorithmic structures. This tool addresses that challenge by automating the process and identifying recurrence relations from code snippets and giving an explanation behind each result.

The project is designed to solve several practical and educational problems. It eliminates the need for manual complex mathematical analysis by providing automated, accurate, and explainable evaluations. It also bridges the gap between theoretical computer science problems and practical software engineering. Furthermore, by containerizing each microservice, the system supports execution, easy maintenance, and scalable updates without the risk of affecting other services.

The target audience for this system includes, software developers, computer science students, and educators who would either like to automate their own code, or learn how to evaluate the computational complexity of algorithms efficiently. Educational institutions could integrate the tool into programming or algorithm design courses. Overall, this project delivers a platform-independent, educational, and beginner-friendly solution for algorithmic complexity evaluation. By combining automation, explanation, and scalability within a microservice architecture, it deepens our understanding of computer science theory and software engineering practice.

1.2 Project Scope

The Algorithmic Complexity Analyzer project focuses on developing a microservice-based tool that automates the evaluation of algorithmic efficiency through Big-) and recurrence relation analysis. The project will deliver a containerized system highlighting the microservice architecture that allows us to work on separate services independently. This ensures scalability and modularity.

Deliverables: The key deliverables include: source code for each microservice, Dockerfiles and configuration files for deployment, a working prototype of the analyzer capable of accepting code files, automated orchestration scripts using Docker Compose, and documentation detailing installation, usage, and system architecture. The final deliverable will be a deployable system that can evaluate and explain algorithmic complexity within five seconds of receiving input.

Work and Resources: The project required about two weeks of development, testing, and documentation. Tasks include system design, microservice implementation, Docker setup, and testing. Required resources include Docker Desktop, a code editor such as Visual Studio Code, a GitHub repository for version control, and access to a command line interface for deployment testing.

Benefits and Outcomes: The primary benefit of this system is automating analyzing algorithmic efficiency, which is a complex and error-prone process. The outcome is user-friendly to those who have experience using a command line interface, educational, and easily scalable. It helps students understand computational complexity by giving explanations and can assist developers in optimizing algorithms during the design phase.

1.3 Technical Requirements

1.3.1 Functional Requirements

The functional requirements ensure that the test automation framework effectively supports quality assurance throughout all stages of development. Automated testing validates the functionality of individual modules, allowing early detection of errors before the final integration. Integration testing ensures that interconnected components communicate and function correctly before deployment. A centralized test case management system enables developers to easily create, modify, and organize tests. This improves accessibility and collaboration efforts.

Mandatory Functional Requirements
The system must support the creation and execution of automated unit tests for individual components of the disaster response coordination application to ensure that each unit functions correctly in isolation.
The framework must facilitate automated integration testing to verify that different components of the application work together as intended, ensuring data flows correctly between modules.
The system must provide a mechanism for managing test cases, allowing developers to create, modify, and organize test cases within the automation framework for easy access and execution.
The automation framework must seamlessly integrate with existing CI/CD pipelines to enable automatic triggering of test execution upon code commits or pull requests, ensuring immediate feedback on code quality.
The system must generate detailed reports on test results, including pass/fail rates, code coverage metrics, and historical trends, allowing teams to assess code quality and identify areas for improvement.
Extended Functional Requirements
Ext. Req 1
Ext. Req 2
Ext. Req 3

Integration with CI/CD pipelines provides continuous code validation, preventing bugs from progressing into larger problems. Finally, detailed reporting on test performance along with tracing the code through historical reports helps teams track quality trends in order to make data-driven improvement decisions. Together, these functions streamline the development process, enhancing reliability, and ensures all systems components perform as intended under real-world conditions.

1.3.2 Non-Functional Requirements

Mandatory Non-Functional Requirements
The framework must include robust error handling and logging mechanisms to capture and report any failures or exceptions during test execution, providing developers with actionable insights for debugging and resolution.
The test automation framework must execute all automated tests within an average time of under 5 minutes to ensure timely feedback during the development process, facilitating rapid iteration.
The system must be designed to accommodate an increasing number of test cases and applications without degradation in performance, allowing the addition of new tests as the project evolves.
The system must implement security best practices to protect test data and configurations, ensuring that sensitive information is not exposed during test execution or reporting processes.
The test automation framework must provide an intuitive and streamlined interface that allows developers to easily create, manage, and run test cases, ensuring a smooth onboarding process with helpful documentation and interactive tutorials to enhance user engagement and satisfaction.
Extended Non-Functional Requirements
Ext. Req 1
Ext. Req 2
Ext. Req 3

The non-functional requirements focus on making the test automation framework reliable, fast, and easy to use. Strong error handling and logging are needed so developers can quickly see what went wrong during testing and fix problems without wasting time or causing delays. The tests should run in under five minutes to give quick feedback, helping teams improve efficiency and make improvements right away. The framework also needs to handle test cases and applications without slowing down, so it can grow as the project expands. Security is important to make sure that any test data, results, or settings are kept safe and private at all times. Finally, the system should be simple to use with clear instructions, examples, and tutorials that help new developers easily

use the application. Altogether, these requirements ensure the network is dependable, secure, and efficient for all development teams.

2 Project Modeling and Design

2.1 System Architecture

The way we designed the architecture was as a Client-Server model where the "client" interacts with the initial microservice, which then orchestrates the remaining steps. All internal microservice communication is mandated to use TCP network sockets for API calls. The Client, Submits source code. The Complexity Analysis Service, Ingests the code, identifies the core algorithm structure, and models the recurrence relation, this uses one of the docker containers. The Recurrence Relation Service, Solves the recurrence relation and computes the Big-O notation, this also uses a docker container. The Result Presentation Service, Generates the final output (Big-O and explanation) and serves it to the client, this is the last one that uses a docker container. The last one is Docker Compose, which automates the building, starting, and connecting of the three microservice containers. The flow of the data and usage goes like this. Client submits code to the Complexity Analysis Service (Input). Complexity Analysis Service sends the generated recurrence relation to the Recurrence Relation Service. Recurrence Relation Service sends the calculated Big-O notation and solution steps to the Result Presentation Service. Result Presentation Service formats the final result and sends it back to the Client (Output).

2.2 Defined Microservices

Our first microservice is the Complexity Analysis Service (CAS). Its role is to be the entry point of the pipeline. It parses the input code snippet (e.g., using Abstract Syntax Trees or code analysis techniques) to mathematically model its runtime behavior, primarily focusing on loops, recursion, and conditional branches, to derive a formal recurrence relation. The next one is the Recurrence Relation Service (RRS). Its role is to solve the provided recurrence relation to determine the asymptotic time complexity. It must implement algorithms (like the Master Theorem or expansion/substitution methods) to handle standard forms, especially divide-and-conquer recurrences. The last one is the Result Presentation Service (RPS). Its role is to finalize the process by compiling the Big-O notation and the detailed analysis explanation into a single, highly readable and structured format for the user. It ensures the Big-O is prominent and the explanation is detailed.

2.3 Development Considerations

There wasn't really many considerations we took into account. We just wanted to finish it as fast as possible and also we already had a pretty good idea of what we wanted to do so we just stuck with it.

3 Implementation Approach

3.1 Software Process Model

Our approach to this project was to just get it done as fast as possible. So we did it in one big sprint. We went and did all the coding first and got it done with and working. Then we went on to the documentation.

3.2 Key Algorithms and Techniques Used

The system employs a layered analytical strategy spanning regular expression matching and symbolic math:

Service: Complexity Analysis Service (CAS), Algorithm: Naive Recurrence Detection (Regex). Uses Python's re module to perform simple, pattern-based recognition. It specifically targets the divide-and-conquer structure T(n)=aT(n/b)+f(n) and provides basic hints for for/while loops. This is a fast, but limited, form of static analysis.

Service: Recurrence Relation Service (RRS), Algorithm: Master Theorem Implementation. Implements the logic of the Master Theorem to solve recurrences of the form $T(n) = aT(n/b) + f(n)$. The service calculates the threshold $\log_b a$ and compares it to $f(n)$ (simplified as n^c) to determine the final Θ bound (Cases 1, 2, and 3).

Service: Result Presentation Service (RPS), Algorithm:Result Aggregation and Formatting. This service acts as an API Gateway/Aggregator, consuming the raw analysis from the CAS (which, in turn, includes the RRS result) and compiling the final, structured output, ensuring the Big-O notation is prominent.

3.3 Microservices and Docker Usage

The project is structured around three independent microservices, each running within its own Docker container:

Service Name: analyzer (CAS), Code name: analyzer.py. Its role is to Receive code, detects recurrence, and orchestrates the call to the RRS. On port 5000.

Service Name: recurrence (RRS), Code name: recurrence.py. It solves the Master Theorem recurrence. On port 5001

Service Name: presenter (RPS), Code name: presenter.py. It is the Final aggregation and output formatting.On port 5002

Docker Usage: Each service uses a minimal python:3.11-slim base image defined in its dedicated Dockerfile. This ensures a lightweight, reproducible environment, satisfying the non-functional requirement for platform independence.

3.4 Container Network Communication Setup and Testing

Communication Setup:

For the first step of the setup. The protocols are all inter-service communication uses the Hypertext Transfer Protocol (HTTP), which operates over TCP network sockets, fulfilling Functional Requirement. The second part of the setup is the Service Discovery. Here the key is the use of service names defined in docker-compose.yml: The analyzer service uses http://recurrence:5001/solve to reach the RRS. The presenter service uses http://analyzer:5000/analyze to reach the CAS. Docker Compose's internal networking and DNS provide this name resolution. For our network testing we made a simple curl command to http://localhost:5002/present (which should trigger the entire internal pipeline) validates the end-to-end communication path. Then we did time constraint test. The requests library calls include a timeout (5 or 6 seconds), aligning with Non-Functional Requirement 4 to complete the analysis quickly.

3.5 Infrastructure as Code approach

4 Software Testing and Results

4.1 Software Testing with Visualized Results

Test Plan Identifier:
Introduction:
Test item:
Features to test/not to test:
Approach:
Test deliverables:
Item pass/fail criteria:
Environmental needs:
Responsibilities:
Staffing and training needs:
Schedule:
Risks and Mitigation:
Approvals:

5 Conclusion

Overall, this was a very useful project to work on. We had very limited knowledge using Docker and microservices before this project. Although we are nowhere near experts in the space yet, we gained footing in containerization practices. The project itself is very useful and fortified our understanding on algorithm analysis as we developed the product. To make it easier on ourselves, we stuck to a command line interface. In past projects, we have tried to stay away from this, but for this one the most important part was making sure it was calculating things effectively and efficiently. Going forward, if we had more time to work on this project or were to continue to work on it after this sprint is done. I believe we could create a simple web-based or graphical user interface. Then we could make the system simpler to use and be able to insert more complex code. As of now, it is pretty basic but we really enjoyed the direction it was going as we went through the tough learning curve.

6 Appendix

6.1 Software Product Build Instructions

Text goes here.

6.2 Software Product User Guide

Text goes here.

6.3 Source Code with Comments

Text goes here.