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

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

4.2 Unit Testing Plan

Test Plan Identifier: UT-001 Provides a unique identifier for this test plan. This plan focuses on unit testing the `naive_data_recurrence` function in the Complexity Analyzer software.

Introduction: This unit test plan verifies that the recurrence detection module correctly identifies simple loops, divide-and-conquer recurrences, and unknown code patterns. The objective is to detect functional errors early, ensure reliable classification, and support future regression testing.

Test item: The Software Under Test (SUT) is the function `naive_data_recurrence` in `analyzer.app`. It takes code snippets or recurrence equations as input and outputs a dictionary indicating the recurrence type and parameters.

Features to test/not to test: In scope: Correct identification of loops, divide-and-conquer recurrences, and unknown patterns. Validation of returned dictionary keys and values. **Out of scope:** Advanced recurrence analysis, multi-language support, and performance testing. Excluded due to low priority for unit-level testing.

Approach: Automated unit tests using `pytest` will supply input strings to `naive_data_recurrence` and assert expected outputs. Functional correctness is the focus. Manual intervention is not required, and performance or security testing is excluded.

Test deliverables: Deliverables include unit test scripts, expected results, actual test results, and pytest-generated reports. Documentation of test cases and execution logs will be maintained for traceability.

Item pass/fail criteria: Tests pass if the function correctly identifies the recurrence type and parameters. Any mismatch between expected and actual output constitutes a failure. All test cases must pass for the module to be considered correct.

Environmental needs: Python 3.13.1 with pytest 8.4.2 installed. Access to the `analyzer.app` module. No network or external services are required.

Responsibilities: Developers create and maintain test scripts. QA engineers execute tests, record results, and report issues. Both teams review failures and coordinate fixes.

Staffing and training needs: Testers must know Python, pytest, and the SUT module. Minimal training is required to understand recurrence types and testing procedures.

Schedule: Test cases are run alongside development.

Risks and Mitigation: Risks include incomplete coverage if new recurrence types are added without updating tests, or misclassification errors. Mitigation involves regular test updates, automated regression tests, and code reviews. Changes to the SUT interface require corresponding test updates.

Approvals: Team approval required.

4.3 Integration Testing Plan

Test Plan Identifier: IT-001 Provides a unique identifier for this test plan. This plan focuses on integration testing the Flask API endpoints of the Complexity Analyzer software.

Introduction: This integration test plan verifies that the '/analyze' API endpoint correctly handles valid and invalid requests. The objective is to ensure proper request processing, response formatting, and error handling when interacting with the recurrence detection module.

Test item: The Software Under Test (SUT) is the Flask app in `analyzer.app`. The '/analyze' endpoint receives JSON payloads containing code snippets, invokes `naive_detect_recurrence`, and returns a structured JSON response with recurrence type and Big-O hints or error messages.

Features to test/not to test: In scope: Correct response for valid code submissions, appropriate error responses for invalid/empty payloads, and proper JSON formatting. **Out of scope:** Authentication, rate limiting, performance under high load, and non-JSON input handling. Excluded due to focus on core functionality and unit integration.

Approach: Automated integration tests using Flask's test client will simulate POST requests to '/analyze' and validate responses. Tests focus on functional correctness, status codes, and response content. Manual testing is not required. No performance or security tests are included in this plan.

Test deliverables: Deliverables include integration test scripts, input payloads, expected outputs, actual responses, and test reports generated by pytest or CI tools. Execution logs and documentation will be maintained for traceability.

Item pass/fail criteria: Tests pass if the API returns the correct status code and response JSON for each input. Failures occur when response structure or content deviates from expected output. All test cases must pass for the endpoint to be considered functionally correct.

Environmental needs: Python 3.13.1 with Flask and pytest installed. Access to the `analyzer.app` module. Local development environment or CI server capable of running Flask test clients. No external services required.

Responsibilities: Developers write and maintain integration test scripts. QA engineers execute tests, record results, and report discrepancies. Both teams collaborate to address failures, review logs, and ensure reliability of API endpoints.

Staffing and training needs: Team members should be familiar with Python, Flask, pytest, and JSON handling. Minimal training required for understanding API endpoints and response validation for integration testing.

Schedule: Integration tests are run alongside development to ensure everything is running smoothly.

Risks and Mitigation: Risks include incomplete coverage if new endpoint features are added without updating tests or changes to JSON response structure. Mitigation includes maintaining test scripts alongside API updates, automated regression testing, and code review. API interface changes require corresponding test adjustments.

Approvals: Team approval required.

4.4 System Testing Plan

Test Plan Identifier: ST-001 Provides a unique identifier for this test plan. This plan focuses on system-level testing of the local Flask app of the Complexity Analyzer software.

Introduction: This system test plan verifies that the Flask application responds correctly to '/analyze' requests. The objective is to ensure functional correctness of the complete system, including request processing, response structure, and handling of valid, empty, and unknown inputs.

Test item: The Software Under Test (SUT) is the Flask application in `analyzer.app`. The '/analyze' endpoint accepts JSON payloads, invokes the recurrence detection logic, and returns structured JSON responses including recurrence type and Big-O hints.

Features to test/not to test: In scope: Correct HTTP responses for valid code submissions, empty requests, and unknown code patterns. Verification of status codes and JSON structure. **Out of scope:** Docker deployment, external network communication, authentication, and high-load performance. These are excluded because this test focuses on local system functionality.

Approach: Automated system tests using Flask's built-in test client will simulate POST requests to '/analyze'. Tests validate status codes, JSON keys, and response content. Functional correctness is prioritized. No manual testing or performance testing is included.

Test deliverables: Deliverables include the system test scripts, input payloads, expected outputs, actual responses, and pytest-generated reports. Logs and documentation of test results will be maintained for traceability and regression testing.

Item pass/fail criteria: Tests pass if the Flask app returns correct status codes and response JSON for all input scenarios. Failures occur when response content, structure, or status code differs from expected results. All tests must pass for the system to be considered correct.

Environmental needs: Python 3.13.1 with Flask and pytest installed. Access to the local `analyzer.app` module. No external services or network connections are required.

Responsibilities: Developers create and maintain system test scripts. QA engineers execute tests, document results, and report failures. Both teams review and address issues to ensure correct system behavior.

Staffing and training needs: Team members should be familiar with Python, Flask, pytest, and HTTP request/response handling. Minimal training is needed to understand endpoint behavior and system-level testing procedures.

Schedule: As soon as possible. Automated execution enables repeated testing after code changes.

Risks and Mitigation: Risks include incomplete coverage if new API features are added without updating tests, or misclassification in the recurrence detection module. Mitigation includes maintaining test scripts alongside code updates, automated regression testing, and code review. Changes to the API interface require corresponding updates to system tests.

Approvals: Team approval required.

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

To continue development of the Algorithm Complexity Analyzer on a new computer, follow these steps:
Step 1: Copy the Project Folder
Copy the entire project directory to the new machine, ensuring all subfolders and files are preserved.
Step 2: Install Prerequisites
Install the following on the new machine:
- Docker Desktop - PowerShell 7 (optional) - VS Code or another code editor (optional)
Step 3: Verify Docker
Open PowerShell and run:

```
docker --version
docker compose version
```

Step 4: Start the Microservices
Navigate to the project folder:

```
cd "C:\path\to\algorithm-analyzer"
```

Then run one of the following:

```
docker compose up --build
# or run in background
docker compose up -d --build
# or use the script
.\run.ps1
```

Step 5: Test the Setup
Send a test request:

```
Invoke-RestMethod -Uri "http://localhost:5002/present" ‘
  -Method Post ‘
  -Headers @{ "Content-Type" = "application/json" } ‘
  -Body ’{"code":"T(n) = 2T(n/2) + n"}’ | ConvertTo-Json -Depth 10
```

Step 6: Stop the Microservices

```
docker compose down
```

—

6.2 Software Product User Guide

The Algorithm Complexity Analyzer allows users to submit algorithm code and receive time complexity analysis.

6.2.1 General User

General users can:
Capabilities: - Submit algorithm code snippets in standard programming syntax. - Receive analysis results including: - Big-O notation of the algorithm - Explanation of the analysis - Recurrence solution (if applicable)
Example: To submit code, send a POST request to the Presentation microservice API:

```
POST http://localhost:5002/present
Body: {"code":"T(n) = 2T(n/2) + n"}
```

The response will contain JSON with the ‘big_o‘ value, explanation, and recurrence solution.

6.2.2 Administrative User

Administrative users manage the microservices and development environment. Their responsibilities include:
Starting and Stopping Docker Containers:

```
docker compose up -d
docker compose down
```

Updating or Rebuilding Services After Code Changes:

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

Monitoring Container Status:

```
docker ps
```

Administrators can also modify or extend microservices for new algorithms or features.

6.3 Source Code with Comments

File: analyzer/app.py

```python
from flask import Flask, request, jsonify
import requests
import re

app = Flask(__name__)
RECURRENCE_SERVICE = "http://recurrence:5001/solve"

def naive_detect_recurrence(code: str):
    s = code.replace(" ", "")
    m = re.search(r"T\(n\)\s*=\s*(\d+)T\(n/(\d+)\)\+(.+)", s)
    if m:
        a, b, f = m.groups()
        return {"type": "divide-and-conquer", "a": int(a), "b": int(b), "f": f}
    if re.search(r"for\s+\w+\s+in\s+range|while\s*\(", code):
        return {"type": "loop", "hint": "may be O(n) or O(n^2) depending on nested
            ↪ loops"}
    return {"type": "unknown"}

@app.route("/analyze", methods=["POST"])
def analyze():
    payload = request.json or {}
    code = payload.get("code", "")
    if not code:
        return jsonify({"error": "no code provided"}), 400

    rec = naive_detect_recurrence(code)
    result = {"recurrence_detected": rec}

    if rec.get("type") == "divide-and-conquer":
        try:
            resp = requests.post(RECURRENCE_SERVICE, json={
                "a": rec["a"], "b": rec["b"], "f": rec["f"]
            }, timeout=5)
            result["recurrence_solution"] = resp.json()
        except Exception as e:
            result["recurrence_solution"] = {
                "error": "could not reach recurrence service",
                "detail": str(e)
```

```
            }

    if rec.get("type") == "loop":
        result["big_o_hint"] = "O(n) or O(n^2) - check nested loops."
    elif rec.get("type") == "divide-and-conquer":
        result["big_o_hint"] = "Use recurrence solution returned above."
    else:
        result["big_o_hint"] = "unknown"

    return jsonify(result)

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

File: recurrence/app.py

```
from flask import Flask, request, jsonify
import math

app = Flask(__name__)

def master_theorem(a, b, f_str):
    try:
        a = int(a); b = int(b)
    except:
        return {"error": "a and b must be integers"}
    log_term = math.log(a, b)
    import re
    m = re.search(r"n\^(\d+)", f_str)
    if m:
        c = int(m.group(1))
        if abs(c - log_term) < 1e-6:
            return {"theta": f"n^{c}", "case": "2 (f(n) = Theta(n^{log_b a}))"}
        elif c < log_term:
            return {"theta": f"n^{log_term:.3f}", "case": "1 (f(n) = O(n^{log_b a -
                ↪ eps}))"}
        else:
            return {"theta": f"f(n) dominates (approx n^{c})", "case": "3 (f(n) =
                ↪ Omega(n^{log_b a + eps}))"}
    return {"theta": f"n^{log_term:.3f} (master theorem baseline)", "note": "f(n) not
        ↪ parsed; returned baseline"}

@app.route("/solve", methods=["POST"])
def solve():
    data = request.json or {}
    a = data.get("a")
    b = data.get("b")
    f = data.get("f", "unknown")
    if a is None or b is None:
        return jsonify({"error": "missing a or b"}), 400
    sol = master_theorem(a, b, str(f))
    return jsonify({"a": a, "b": b, "f": f, "solution": sol})

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5001)
```

File: presentation/yeayea.py

```
from flask import Flask, request, jsonify
import requests
```

```python
app = Flask(__name__)
ANALYZER_SERVICE = "http://analyzer:5000/analyze"

@app.route("/present", methods=["POST"])
def present():
    payload = request.json or {}
    code = payload.get("code")
    if not code:
        return jsonify({"error": "no code provided"}), 400

    try:
        # Step 1: Send code to Analyzer
        resp = requests.post(ANALYZER_SERVICE, json={"code": code}, timeout=6)
        analysis = resp.json()
    except Exception as e:
        return jsonify({"error": "could not reach analyzer", "detail": str(e)}), 500

    # Step 2: Extract recurrence solution if available
    recurrence_solution = analysis.get("recurrence_solution", {})
    solution_data = recurrence_solution.get("solution", {})

    # Step 3: Fill big_o using recurrence solution if available
    if solution_data.get("theta"):
        big_o = solution_data["theta"]
    else:
        big_o = analysis.get("big_o_hint", "unknown")

    formatted = {
        "big_o": big_o,
        "explanation": "The recurrence was analyzed using the Recurrence service. See
            ↪ recurrence_solution for step-by-step details.",
        "recurrence_analysis": recurrence_solution
    }

    return jsonify({"analysis": formatted, "raw": analysis})

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5002)
```