School of Engineering, Computing and Mathematics

Oxford Brookes University

COMP7033 (SOFT7011) Big data and The Cloud

(Semester 2, 2023-2024)

Coursework 2 Specification

Detailed Design and Implementation of

A Big Data and Cloud Application

# Introduction

Our architectural journey in the field of software engineering centres on creating a University Management System. We use microservices architecture, which divides difficult jobs into smaller, more manageable parts, such as the Survey Manager and University Account Manager. Our databases are painstakingly built to prioritise security and scalability while meeting critical data requirements. We prioritise designing user-friendly GUIs and APIs to improve user experience. Our system is built with fault tolerance techniques and stateless architectures, which are crucial for performance, reliability, and scalability. In the ever-changing field of software engineering, we aim for excellence, agility, and user-centricity through rigorous testing and execution.

Table of Contents

[Introduction 1](#_Toc166003522)

[Architectural design 3](#_Toc166003523)

[Specification of Microservices 5](#_Toc166003524)

[Specification of Databases 6](#_Toc166003525)

[GUI & API Designs 6](#_Toc166003526)

[Designs 7](#_Toc166003527)

[1. Login 7](#_Toc166003528)

[2. Create University Account 7](#_Toc166003529)

[3. Activate/Deactivate User 7](#_Toc166003530)

[4. Customize account 7](#_Toc166003531)

[5. Create Survey 7](#_Toc166003532)

[6. Submit Survey Response 8](#_Toc166003533)

[7. Publish results 8](#_Toc166003534)

[8. Perform Analysis 8](#_Toc166003535)

[Implementation 8](#_Toc166003536)

[Integration With Other Subsystems 8](#_Toc166003537)

[Unit Test 9](#_Toc166003538)

[Testing and Integration 9](#_Toc166003539)

[Test Cases 10](#_Toc166003540)

[1. Functional Test: 10](#_Toc166003541)

[2. Performance Test: 10](#_Toc166003542)

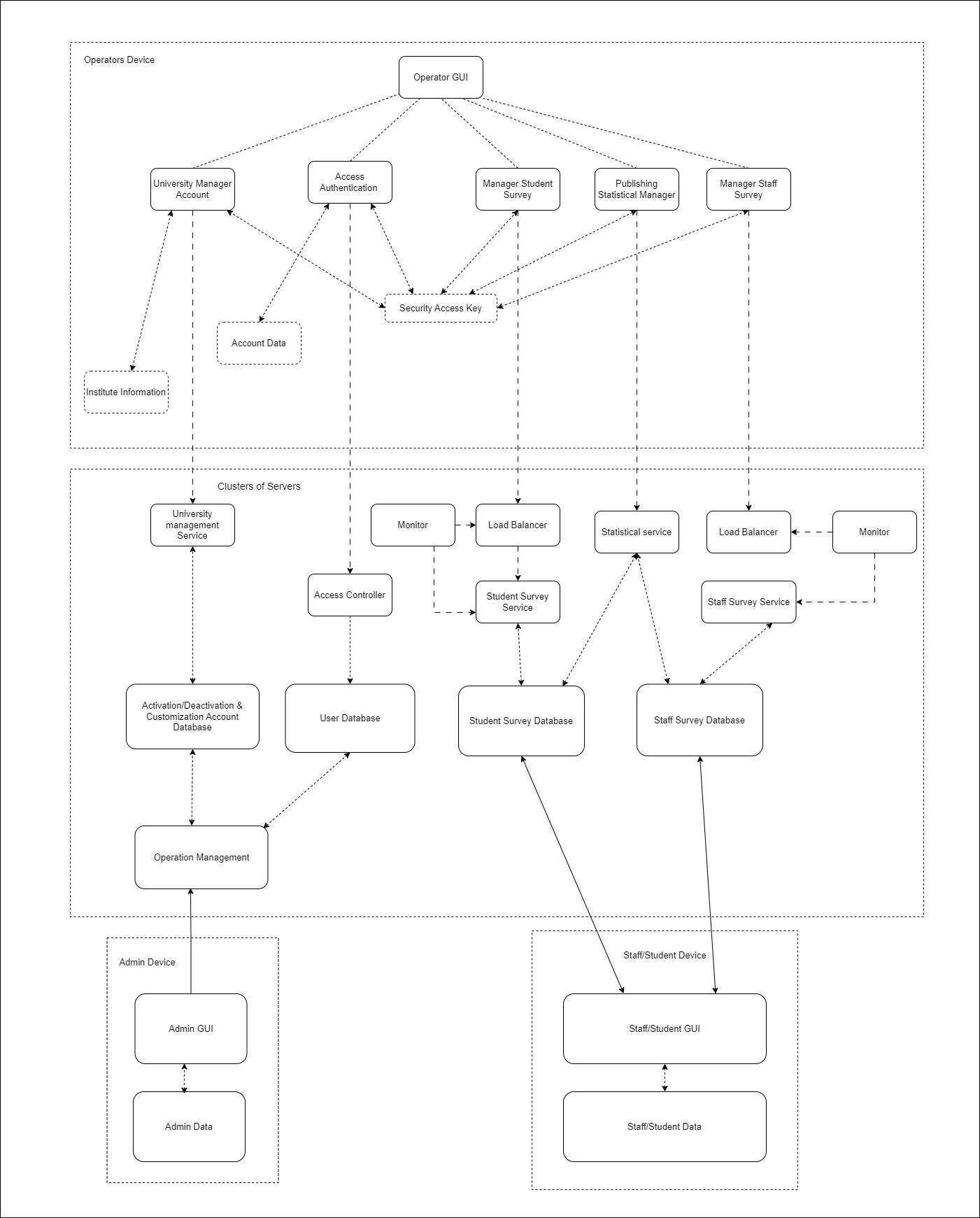
[3. Scalability Test 10](#_Toc166003543)

[4. Fault Tolerance Test: 10](#_Toc166003544)

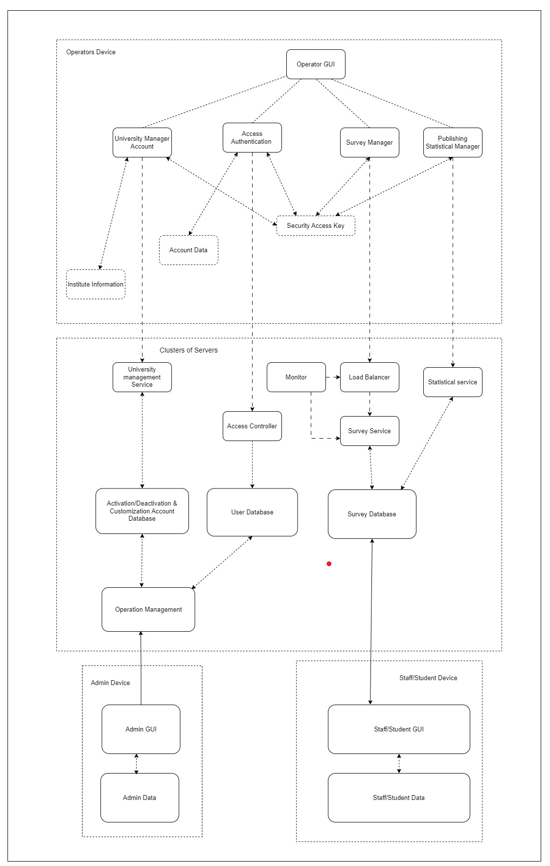
[Individual Reflection 11](#_Toc166003545)

[Conclusion 11](#_Toc166003546)

# Architectural design



Initial Design



Updated Design

## Specification of Microservices

|  |  |
| --- | --- |
| **Service Name** | **University Account Manager** |
| Description | This subsystem acts as the central component responsible for managing and coordinating interactions with other university services. |
| Provided Services - CreateAccounts | Function - Activate/Deactivate University Account; |
| Parameters - University ID; |
| Provided Services - ManageAccounts | Function - Customise University Account; |
| Parameters - University ID; |
| Requested Services - CustomizeAccounts | Service - University Account Manager |
| Operation - User authentication, profile management |
| Parameters - User credentials, profile information |
| Reply - Login Success Message/ Error |

|  |  |  |
| --- | --- | --- |
| **Service Name** | | **Survey Manager** |
| Description | | This service works on the computer or mobile device, allowing the Student/Staff to conduct annual surveys. Managers can get more statistical data. |
| Provided Services – CreateSurveys,  GetSurveys,  SubmitPublishSurveys | | Function - Create new Survey, Submit Survey Response, Get Surveys, Publish Survey Results; |
| Parameters - University ID, Student ID, Staff ID, Survey Details, Survey Results, Statistical Data, Edit Survey; |
| Requested Services  StatisticalAnalysis | - | Service - Publishing Statistical Manager |
| Operation - Post/Analyse |
| Parameters - Data Analysis |
| Reply - Analysis Results/Error |

## Specification of Databases

|  |  |
| --- | --- |
| **Database Name** | **University Account Database** |
| Description | This Database is a critical component and used to securely manage and organize the accounts of various universities hosted on the platform. This database serves as a centralized repository, storing essential information related to each university's account, including user credentials, account settings, and access permissions. |
| Design | Tables - University Account, Status, Setting, Security |
| Type - Account | University ID, University Name |
| Type – University ID, Status | Active, Inactive |
| Type - Security | University ID, Password |

|  |  |
| --- | --- |
| **Database Name** | **Survey Database** |
| Description | The database stores the reviews and feedback of the students &staff. |
| Design | Aggregate MongoDB Database |
| Table - Surveys | University ID, Rank |

# GUI & API Designs

To tackle scalability requirements, I opted for a microservices architecture, which provided the flexibility needed to scale individual components independently based on demand. This approach meant that each functional aspect, such as account management and survey operations, was encapsulated within its own microservice. Consequently, when the system experienced increased load, it could scale horizontally, distributing the workload across multiple instances of these microservices. This not only ensured that the system could handle higher traffic volumes but also minimized the risk of downtime during periods of high demand.

In terms of performance, I prioritized the design of lightweight microservices. By utilizing optimized communication protocols, I aimed to reduce latency and enhance the system's responsiveness. To ensure that the system met performance expectations, I established specific benchmarks for actions such as account management and survey operations. Through rigorous testing procedures, I verified that these benchmarks were consistently achieved under various load conditions. This proactive approach not only ensured that users experienced minimal delays but also laid the foundation for a robust and reliable system.

For reliability, I implemented fault tolerance mechanisms within the microservices architecture. This included redundancy and failover strategies to mitigate the impact of potential failures. By designing the system with fault tolerance in mind, I aimed to minimize disruptions and maintain service availability even in the face of unexpected errors. To validate the system's resilience, I conducted comprehensive fault injection and failure scenario testing, ensuring that it could withstand a range of adverse conditions.

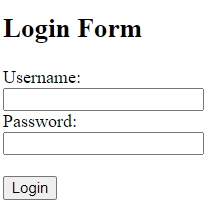
Furthermore, I adopted a stateless design approach to enhance scalability and performance. By eliminating the need to store session history and state, each service request could be processed independently by any instance of the microservice. This not only simplified horizontal scaling but also improved fault tolerance by reducing dependencies between components.

To facilitate maximal parallel processing and optimize system efficiency, I specified non-blocking service requests, particularly for inter-microservices APIs. This allowed multiple requests to be processed concurrently, minimizing bottlenecks and maximizing throughput.

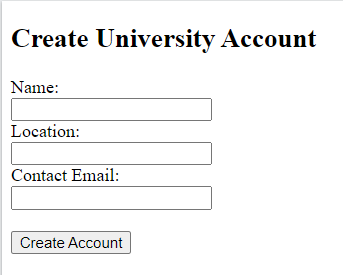
Finally, I ensured that response messages provided comprehensive information about any failure modes encountered. By offering detailed insights into the cause of errors, these messages enhanced transparency and aided in troubleshooting, ultimately contributing to a more resilient and user-friendly system.

# Designs

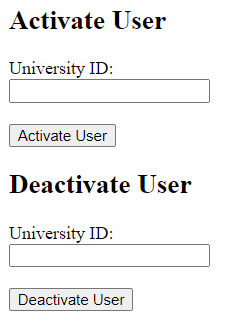
## Login

****

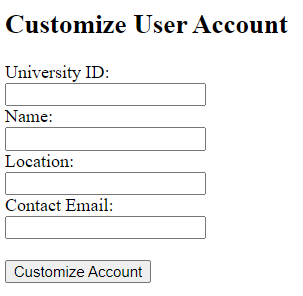
## Create University Account



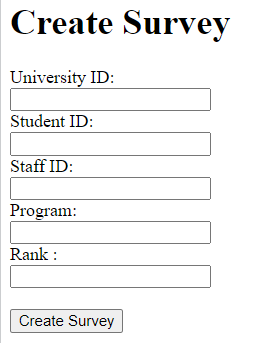
## Activate/Deactivate User



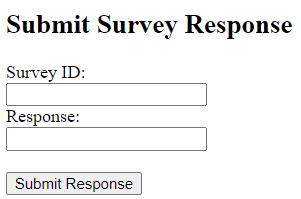
## Customize account



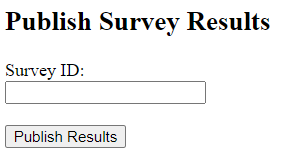
## Create Survey



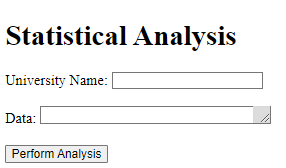
## Submit Survey Response



## Publish results



## Perform Analysis



# Implementation

In terms of completeness, I ensured that all the essential functionalities specified in the coursework were implemented for my selected user type. This included implementing user registration and authentication, as well as additional main functions of the subsystem such as university account management, user activation and deactivation, and customization of user accounts. To leverage appropriate Big Data and Cloud computing techniques, I opted for MongoDB Atlas, a cloud-based NoSQL database, to store and manage user, university, and survey data effectively. Each endpoint in my implementation serves as a microservice, handling specific functions related to user authentication, university management, and survey handling. While I haven't explicitly incorporated microservice design patterns like load balancer or circuit breaker, I've designed the system with flexibility to include these patterns in the future for scalability and fault tolerance. As for the quality of coding, I adhered to Python coding conventions and FastAPI framework guidelines, maintaining consistency, readability, and clarity through well-structured code and informative comments.

# Integration With Other Subsystems

I ensured that my implementation took into account the data and services that would be used by other subsystems while offering APIs. In the design paper, I specified the APIs that each of the other subsystems would utilise, as well as the features to which they would need access. Then, in the implementation code, I included APIs for these subsystems, ensuring that they were accessible and well-documented. For example, the Survey Manager subsystem provides APIs for producing and managing surveys, which are essential for gathering information from students and staff.

# Unit Test

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case Description** | **Test Data** | **Expected Result** | **Actual Result** | **Status** |
| Login with valid credentials | {"username": "Sagar", "password": "Sagar@123"} | Successful login | {"message": "Login successful"} | Passed |
| Login with invalid credentials | {"username": "user1", "password": "invalid"} | Error message: "Invalid username or password" | {"detail": "Invalid username or password"} | Failed |
| Create university account | {"name": "University A", "location": "City A", "contact\_email": "email@example.com", "username": "user1", "university\_id": "123", "university\_name": "University A", "is\_active": true} | University created successfully | {"message": "University created successfully", "university\_id": "xyz"} | Passed |
| Attempt to create duplicate university account | {"name": "University A", "location": "City A", "contact\_email": "email@example.com", "username": "user1", "university\_id": "123", "university\_name": "University A", "is\_active": true} | Error message: "University with this name already exists" | {"detail": "University with this name already exists"} | Failed |
| Submit survey response with valid survey ID | Survey ID: "1234567890", Response: "Agree" | Survey response submitted successfully | {"message": "Survey response submitted successfully"} | Passed |
| Submit survey response with invalid survey ID | Survey ID: "invalid\_id", Response: "Disagree" | Error message: "Survey not found" | {"detail": "Survey not found"} | Failed |
| Publish survey results with valid survey ID | Survey ID: "1234567890" | Survey results published successfully | {"message": "Survey results published successfully"} | Passed |
| Publish survey results with invalid survey ID | Survey ID: "invalid\_id" | Error message: "Survey not found" | {"detail": "Survey not found"} | Failed |

# Testing and Integration

To test the system as it is used, I would employ a method that simulates user interactions with the system for a specific use case. This entails sending a series of service requests, receiving responses, and monitoring changes in the system and messages transferred between microservices.  
  
First, I would define a typical user scenario, such as a student creating a university account, customising their account settings, and then submitting a survey response. I would run each step of this scenario using test code or by manually using the system's GUI (if accessible), confirming that the system reacts appropriately and accurately to each request.   
  
By addressing every potential use case and scenario inside the selected user scenario, I would make sure the testing was adequate throughout. To guarantee complete coverage, I would test variations of the registration procedure, such as valid and invalid inputs.   
  
For tools, I may use custom-written test code to automate the interaction process or manually run the GUI if it is available. GUI-based test scripts could also be used if necessary.   
  
By extensively evaluating the system in this manner, I would develop confidence in its functionality and guarantee that it performs as predicted in real-world settings.

# Test Cases

## Functional Test:

* Purpose: I conduct functional tests to ensure that the implemented functionality works as expected and meets the requirements.
* Method: I send various requests to the API endpoints and check if the responses and internal state changes align with the expected behavior. This includes verifying login functionality, creating university accounts, submitting survey responses, etc.

## Performance Test:

* Purpose: I evaluate how well the system performs under normal load conditions.
* Method: I send a large number of requests to the API endpoints and measure the response times. I monitor the server resources (CPU, memory, network usage) to ensure they are within acceptable limits. I use tools like Apache JMeter or Locust to simulate concurrent user requests and observe the system's behaviour.

1. Scalability Test:

* Purpose: I assess the system's ability to handle increased load by scaling out.
* Method: I gradually increase the number of concurrent user requests and observe how the system responds. I monitor metrics such as response time, throughput, and resource utilization as the load increases. I use Kubernetes to scale the application horizontally by adding more instances of the microservices and observe if the system can handle the increased load effectively.

## Fault Tolerance Test:

* Purpose: I evaluate if the system can recover from failures gracefully without affecting overall functionality.
* Method: I introduce faults into the microservices by manually crashing them or simulating network issues. I observe how the system responds to these failures. I check if the system can handle errors and failures without crashing completely. I monitor log files and error messages to identify any issues and ensure proper error handling mechanisms are in place. I use Kubernetes to automatically restart failed microservices and verify if the system can recover without human intervention.

# Individual Reflection

This architectural project has been a fascinating journey, providing vital insights into the complexities of creating sophisticated software systems. Throughout the process, I improved my skills in system design, microservices architecture, and database management. The experience of transforming conceptual designs into tangible implementations has helped me gain a better knowledge of software engineering principles and best practices.  
  
Furthermore, this research demonstrated the value of collaboration and interdisciplinary teamwork. Working directly with colleagues and stakeholders allowed me to obtain a comprehensive understanding of system requirements and user needs, which improved the quality of our design decisions.

Looking back, I recognise the issues we faced, such as combining performance optimisation with fault tolerance and maintaining seamless integration across subsystems. However, each difficulty provided an opportunity for development, promoting resilience, adaptability, and problem-solving abilities.   
  
As I reflect on this journey, I feel a deep feeling of satisfaction and a renewed commitment to software engineering excellence. I carry on the lessons acquired and experiences gained, ready to take on new architectural challenges with confidence and excitement.

# Conclusion

In our drive to create a strong University Management System, we navigated architectural complexities, thorough planning, and rigorous implementation. Our use of the microservices architecture gave us with the flexibility we needed to deal with the challenges of university management. By encapsulating core functionality in distinct microservices, we laid the groundwork for a scalable, dependable, user-centric system.   
  
Our attention on performance optimisation guaranteed that the system could respond quickly even under heavy loads. We strengthened the system against probable failures by incorporating fault tolerance techniques and stateless architectures, ensuring its availability and reliability. Our commitment to transparency and usability is shown in our detailed error messages and user interface designs.

As we conclude this architectural project, we are pleased of the system we have created—a system capable of meeting the changing needs of university management with agility, resilience, and excellence.

GitHub Link - <https://github.com/sagar0135/Operator>