# 4.2C - Extended Calculator Microservice – Documentation

**Github Link** : <https://github.com/Pasindufdo98/sit737-2025-prac4c>

This project builds upon the work done in Practical Task 4.1P. To start, I made a copy of the original 4.1P code and used it as the base for this extended version. The goal was to introduce a few additional mathematical operations and improve overall functionality, while still maintaining proper logging and error handling practices.

What’s New

In addition to the basic operations (addition, subtraction, multiplication, and division), the microservice now supports:

* Exponentiation – calculates the result of a number raised to a power.
* Square root – finds the square root of a single number.
* Modulo – returns the remainder after dividing one number by another.

**Steps Taken to Extend the Microservice**

1. Copied the Code from 4.1P

I began by creating a copy of the original Practical 4.1P project files so that I could build on top of a working version without affecting the original.

2. Added New Math Functions

In the main server file, I added the logic for three new operations: exponentiation, square root, and modulo. These functions were written in a similar structure to the existing math operations.

3. Created New Endpoints

To make these new functions accessible, I created new API endpoints for each one:

/exponentiation

/square-root

/modulo

Each endpoint accepts query parameters and calls the relevant function, just like the original four operations.

4. Updated Input Validation

I updated the validation logic to handle cases where only one parameter is needed (e.g., square root), and to provide meaningful error messages if the input is invalid or missing.

5. Maintained Error Handling

All operations were wrapped in try-catch blocks to ensure errors (like division by zero or invalid inputs) are caught and handled gracefully.

6. Continued Using Winston for Logging

The Winston logger from 4.1P was kept and used throughout the new operations. Each request is logged with details such as input values, type of operation, and any error messages if something goes wrong.

**Screenshots**A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

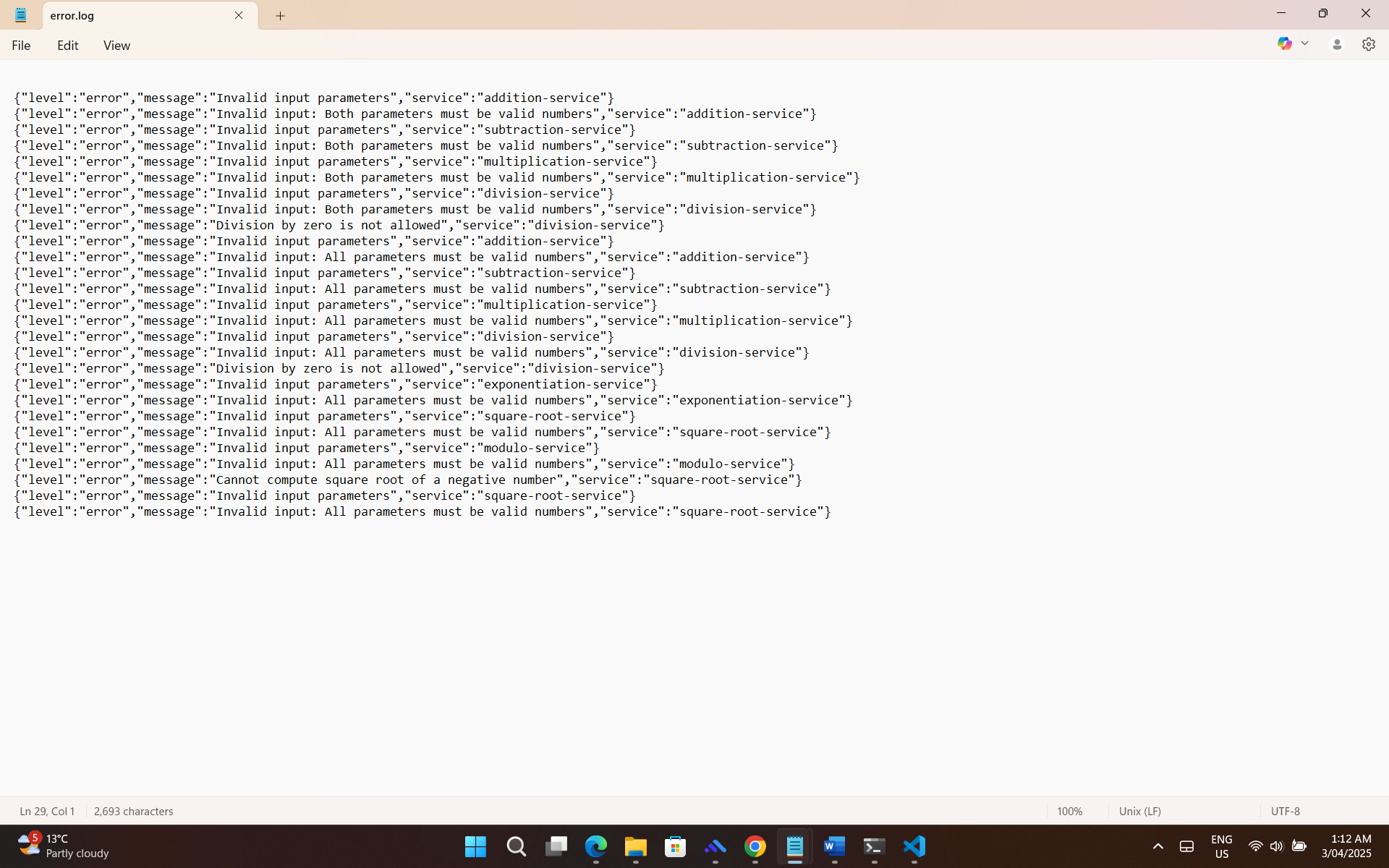
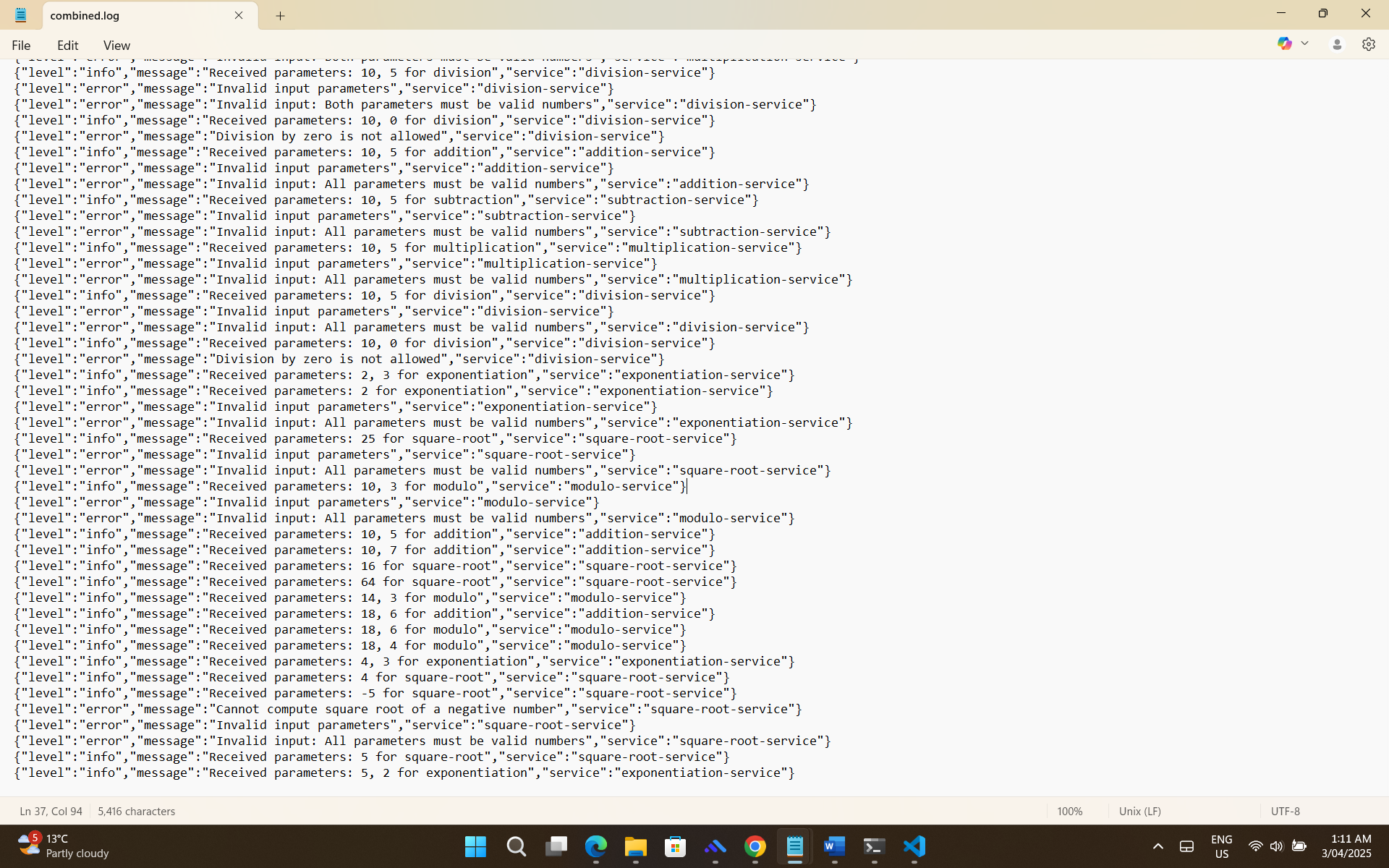
AI-generated content may be incorrect.A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

## Part II: Various error handling strategies prevalent in microservices architecture

In microservices architecture, errors are expected due to the distributed nature of the system, so having strong error-handling strategies is essential to keep services reliable and user-friendly. One commonly used method is the Circuit Breaker pattern, which prevents a failing service from being overwhelmed by stopping calls to it once a failure threshold is reached. It switches between states “closed”, “open”, and “half-open” to test when the service is healthy again and helps avoid full system crashes. This not only protects system resources but also allows dependent services to continue functioning without being dragged down. Another supportive pattern is the Retry pattern, which automatically re-executes failed operations that might be caused by transient problems, such as network latency or timeouts. It’s important to use techniques like exponential backoff (increasing wait times) and jitter (adding randomness) to avoid retry storms that can overwhelm the network and backend systems. Developers should also ensure that operations are idempotent, meaning repeated requests won’t cause unintended effects.

Fallback mechanisms are also essential, they act as a backup by causing users to see some content even if a service is down. This can include displaying cached content, using default values, or silently masking non-critical functionality while preserving core functions. These mechanisms are especially useful in maintaining user trust and lessening frustration during partial service outages. Together, these patterns form a multi-layered defence that enables microservices to recover gracefully and fail safely. With centralized logging, real-time monitoring, and custom exception handling, they help development teams detect and resolve issues faster, leading to much less downtime. Overall, using a combination of circuit breakers, retries, and fallbacks helps build resilient microservices that can recover from failures neatly without impacting the user experience adversely, even on unforeseen failures.  
  
References:

* MultiGenesys author, 2023. Handling Exceptions in Microservices: Best Practices. [online] MultiGenesys. Available at: <https://multigenesys.com/blog/how-to-handle-exceptions-in-microservices> [Accessed 31 March. 2025].
* Swift, D., 2024. Best Practices for Handling Exceptions in Java Microservices. [online] Springfuse. Available at: <https://www.springfuse.com/exception-handling-best-practices-in-microservices/> [Accessed 31 March. 2025].
* Microsoft Learn, 2025. Circuit Breaker Pattern - Azure Architecture Center. [online] 22 March. Available at: [https://learn.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker](https://learn.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker%20) [Accessed 1 Apr. 2025].
* Mezo Code, 2024. Resilience in Microservices: Implementing the Retry Pattern in Java. [online] 25 June. Available at: <https://mezocode.com/implementing-retry-pattern-in-java/> [Accessed 1 Apr. 2025].