**FLEXIFAAS**

**CSIW ZG628T DISSERTATION**

**by**

**Abid Rafique Khan**

**2021WA86438**

**Dissertation Work carried out at**

**Wipro Technologies, Pune**

**BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE**

**Pilani (Rajasthan) India**

****

**November, 2025**

**CSIW ZG628T DISSERTATION**

**FLEXIFAAS**

**Submitted in partial fulfilment of the requirements of**

**M. Tech Computing Systems & Infrastructure Degree Program**

**by**

**Abid Rafique Khan**

**2021WA86438**

**Under the supervision of**

**Ayush Ranjan – Project Engineer**

**Dissertation work carried out at**

**Wipro Technologies, Pune**

**BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE**

**PILANI (RAJASTHAN)**

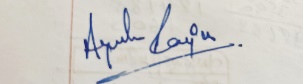
****

**November, 2025**

**BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI**

**CERTIFICATE**

This is to certify that the Dissertation entitled **FLEXIFAAS** and submitted by **Abid Rafique Khan** ID No: **2021WA86438** in partial fulfillment of the requirements of CSIW ZG628T Dissertation, embodies the work done by the him under my supervision



**Signature of the Supervisor**

**Ayush Ranjan**

**Project Engineer**

Date: 18/09/2025

**ACKNOWLEDGEMENT**

First and foremost, I would like to express my special thanks of gratitude to my supervisor, **Ayush Ranjan** who guided me in building **FlexiFaaS** Project. His experience helped me in building this application and in solving various issues as well as understanding the architecture.

I would like to thank my BITS Pilani Examiner for giving me this opportunity to work on this. It was possible because he accepted the idea of my project. My Sincere efforts have made me to accomplish the task of completing this project, I have taken efforts in this project. However, it would not have been possible without the kind support and help of many individuals.

Here I am extending my gratitude to my both the Wipro Examiners, who really helped me for this project. I came to know about so many new things, and I am thankful to them.

**TABLE OF CONTENTS**

|  |  |
| --- | --- |
| Abstract | 6 |
| Area of Research | 7 |
| Objectives | 8 |
| Scope of Work | 9 |
| Architecture Diagram | 10 |
| Literature Review | 11 |
| Methodology | 12-13 |
| High Level Architecture | 14 |
| High Level Diagram | 14 |
| Plan Of Work | 15-16 |
| Code Snippets | 17-21 |
| API Testing via Postman | 22-26 |
| References | 27 |

**ABSTRACT**

This dissertation is all about creating a platform for running user-defined serverless functions, which I’ve named FLEXIFAAS. The core goal was to let people upload, manage, and run little chunks of code (like Python, JavaScript, or Java) without having to worry about any of the complex backend setup. For me, it was a way to bring the cool features of cloud platforms like AWS Lambda or Google Cloud Functions into a simpler, self-hosted system that anyone could use—like for a college project, or maybe even for a small company.  
  
At the start, I dug into a lot of research papers and blogs, trying to figure out what’s actually important in serverless tech, and what’s just “buzzwords”. Most of them focus on things like scalability, multi-tenancy, and event-driven execution. Since I don’t have Google’s budget, I had to improvise—so the platform uses Spring Boot for backend logic, MySQL to keep data, RabbitMQ to handle function execution queues, and some basic React stuff for the user interface (still work in progress there). I also made a REST API, so everything is easy to test and can be connected with different clients.  
  
The main architecture lets users upload their functions, stores metadata and code files, and then allows execution requests. When you want to run something, the request gets queued in RabbitMQ, and then the worker service picks it up, executes the code based on its runtime, and logs everything—success, errors, and even the outputs. So far, I managed to get the function execution and queuing flow pretty stable. Executing results logging was also helpful for post-analysis and debugging, as it would be in actual projects. I did encounter some speed bumps, particularly with respecting various programming languages securely and dealing with file uploads, but I gained a great deal of experience debugging them.  
  
The following step will be security-oriented—such as encrypting files before they are saved, scanning for potential malware in uploaded files, and user authentication through JWT. I would also like to make the frontend UI more user-friendly and implement middleware for features such as authentication and authorization.  
  
Overall, this project isn’t just academic—it’s based a lot on my work experience in IT, but also gave me a chance to try out stuff I always wanted to, like cloud-native patterns and distributed messaging. I’m hoping FLEXIFAAS will show how serverless concepts can be adopted on a small scale, without needing a giant budget or complicated setup.

**AREA OF RESEARCH**

**Dissertation Topic: FLEXIFAAS - A Cloud-Native Approach for Secure and Scalable Function Execution**

The dissertation takes place in fast-changing world of cloud-native computing, especially looking at serverless architectures and secure function execution platforms. The research explores how cloud-native and serverless models can be used to allow for on-demand, scalable, and efficient execution of the user-defined code functions without relying on traditional servers.

In particular, the research explores:

* **Serverless Function Deployment:**
  + Investigating mechanisms and design patterns for supporting users in uploading, registering, and running code in several languages (Java, Python, JavaScript) on a platform-agnostic, cloud-native environment.
* **Event-driven and Asynchronous Execution:**
  + Utilizing message-oriented middleware (RabbitMQ) to separate function call from execution to achieve horizontal scalability, more efficient resource utilization, and enhanced fault tolerance.
* **Middleware Security Integration:**
  + Solving most significant security issues—such as authentication, authorization, secure code storage, and file-level encryption—by creating a modular middleware layer.
* **Full-stack Implementation:**
  + Embracing latest full-stack practices, with Java Spring Boot for server-side APIs, MySQL for storage, RabbitMQ for queuing messages, and ReactJS for the interface.

The research is inspired by top-tier FaaS providers (such as AWS Lambda and Azure Functions), yet set out to create an open, extensible, and self-hosted solution with added middleware and security properties applicable to enterprise use cases.

**OBJECTIVES**

The primary aim of this dissertation is to create and put into action a Serverless Function Deployment Engine which provides a secure, scalable, and language-independent function execution for the developers using a cloud-native environment.   
The project intends to:

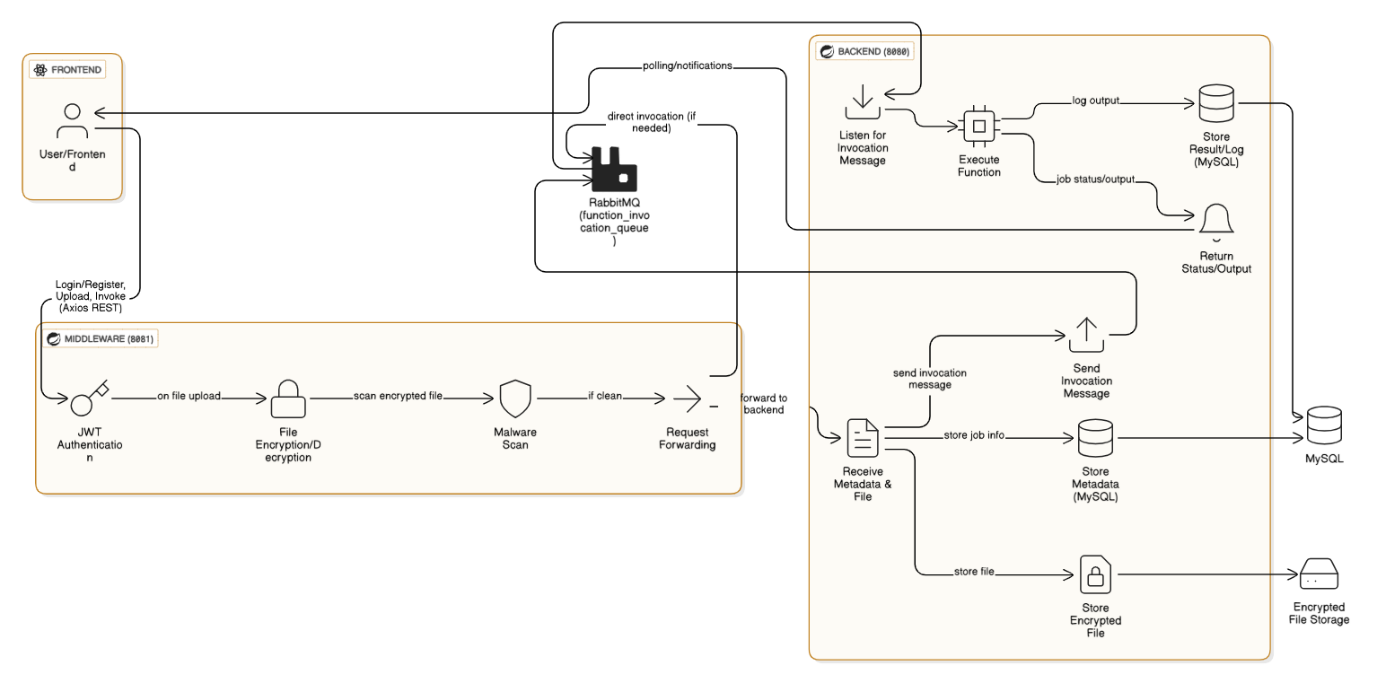
1. Develop a user-friendly platform where individuals can upload, register, and manage their own code functions in various programming languages (Java, Python, JavaScript).
2. Enable asynchronous and separate function execution through message queuing (RabbitMQ), mimicking the event-driven execution model found in commercial-grade FaaS platforms.
3. Incorporate robust middleware services to address essential security requirements—such as authentication, authorization, file encryption/decryption, and malware simulation verification checks—ensuring that function calls are secure and reliable.
4. Create a full-stack solution with a RESTful backend (Spring Boot + MySQL), middleware layer for message processing and security, and interactive ReactJS frontend for end-to-end user experience.
5. Make it modular and extensible, thus enabling the platform to be extended for research purposes in academia, enterprise deployment, or future extensions (e.g., more language support, enhanced logging, auto-scaling, etc.).

**SCOPE OF WORK**

The dissertation has addressed a wide topic, with emphasis on the life cycle concerning serverless function management in a secure, new, and agile environment.  
  
The parameters specified are:

* Backend Services:
  + Building and implementing REST APIs to upload functions, register them, handle execution requests, and retrieve logs.
  + Long-term storage of execution, function, and user metadata using MySQL.
* Function Execution Engine:
  + Uploading files and managing metadata to facilitate code operations.
  + Simulating function execution in isolated environments based on runtime parameters (Java, Python, JavaScript).
  + Decoupled consumer and producer logic with asynchronous execution management via RabbitMQ.
* Middleware Security Layer:
  + JWT-based authentication and detailed authorization mechanisms.
  + AES encryption of data before storage, along with secure decryption during runtime.
  + Simulated virus/malware scanning for uploaded files (optional/advanced).
* Frontend (ReactJS):
  + Incorporating main UI components for function upload, execution, and displaying logs/results.
  + Integrating backend API through secure HTTP requests.
* Evaluation and Record-Keeping:
  + Conducting end-to-end testing of APIs using Postman and automated test suites.
  + Providing thorough documentation of design decisions, workflows, and implementation specifics.
* Out of Scope:
  + Actual deployment on public cloud providers (like AWS, Azure).
  + Native support for non-JVM/non-Node runtimes outside the specified languages.
  + Advanced security certifications or compliance frameworks.

The project’s objectives and scope are designed to deliver a fully working, academically sound, and demonstrably secure serverless function engine, showcasing best practices in cloud-native and middleware software design.



*System Architecture: High-level design of FlexiFaaS platform with frontend, middleware, backend, RabbitMQ, and MySQL integration.*

**LITERATURE REVIEW**

Before getting into this project, I spent a decent amount of time reviewing how the industry and academia both have tackled serverless computing, especially Function-as-a-Service (FaaS) platforms, over the past few years. The serverless paradigm itself really started to get popular after AWS Lambda launched back in 2014, and after that, a bunch of cloud giants like Azure and Google Cloud brought their own versions.

A lot of the core research (like Jonas et al., 2019) talks about how serverless is great for “event-driven” workloads and makes scaling almost automatic. But at the same time, there’s a lot of real-world talk about the challenges around cold start latency, vendor lock-in, and (especially in academic circles) the need for more open-source alternatives to big vendor solutions.

A lot of work is also published on secure function execution and resource isolation. For example, OpenFaaS and Apache OpenWhisk are both well-documented in terms of architecture and were super helpful to read about, since they focus on making serverless tech more “self-hosted” and less locked-in to any one company. But most of them assume you’ll always run on Kubernetes or Docker Swarm, and honestly, the setup for students or small teams is still kind of a pain.

On the academic side, papers like “Serverless Computing: Economic and Architectural Impact” by Spillner et al. (2017) and “A Berkeley View on Serverless Computing” (Jonas et al.) dig deep into why decoupling compute from infrastructure is a big deal, and how message queues (like RabbitMQ, Kafka, etc.) help keep everything asynchronous. Some researchers (Malawski et al., 2020) also highlight the need for stronger security and function isolation, which is a big gap in a lot of do-it-yourself systems.

Security-wise, there is a surprising lack of practical, open-source guides on adding encryption, file scanning, or even JWT authentication to student-level serverless projects. Most of the time, you get blog posts or half-baked GitHub repos, so I ended up referencing both academic stuff and professional docs (Spring, OWASP, etc.) to put together the middleware layer for this dissertation.

**METHODOLOGY (TECHNIQUES / TOOLS USED SO FAR)**

The research approach that was used in this dissertation project was non-traditional and exposed to my experience in dealing with the real world, in which flexibility is required in dealing with emerging challenges. I started by establishing the architecture and verifying compatibility between the core tools, which took significant time devoted to manual reading and prototyping prior to developing useful code.

**Step 1: Selection and Configuration of Technology**

* First, I utilized Java with Spring Boot on the backend since it's tried and tested and supports enterprise systems well (also, admittedly, to be honest, most businesses still utilize Java heavily). MySQL is utilized for the database since it's stable and free, and I already have experience with it through personal projects. RabbitMQ was utilized for asynchronous message processing since it's open-source, highly stable, and supports Spring well.
* Getting the project's basic setup took longer than expected. It was mainly because of the dependency hell and the requirement for ensuring the right versions were there. But once these problems were fixed, the backend started working on port 8080 and was able to communicate with MySQL locally successfully.

**Step 2: Entity Design & CRUD Endpoints**

* Then, I defined the basic entities (User, Function, ExecutionLog), mapped them to JPA, and added annotations to allow Hibernate to automatically create corresponding tables. After successful implementation of this module, I created repositories, services, and Data Transfer Objects (DTOs) for the main models. Most of the basic Create, Read, Update, Delete (CRUD) endpoints were tested via Postman, which, in all fairness, helped me catch many insignificant errors (like missing annotations and wrong DTO mappings).

**Step 3: File Upload & Function Registration**

* One very difficult part was handling file uploading and linking the uploaded files to certain functions. I used Spring's MultipartFile to handle this and decided to save the files on the server's local disk. I also had to add supporting code to handle file paths well and prevent any file from being uploaded over another person's function by mistake.

**Step 4: Simulation of Function Execution**

* Rather than executing the uploaded code directly, a solution was chosen to add logic that generates "execution" using ProcessBuilder for Java, JavaScript, and Python platforms. This makes the system more dynamic and closely follows the operational patterns of production Function as a Service (FaaS) systems. Considerable attention was given to exception handling, as a single typographical mistake in the code can make the functionality of the entire server useless.

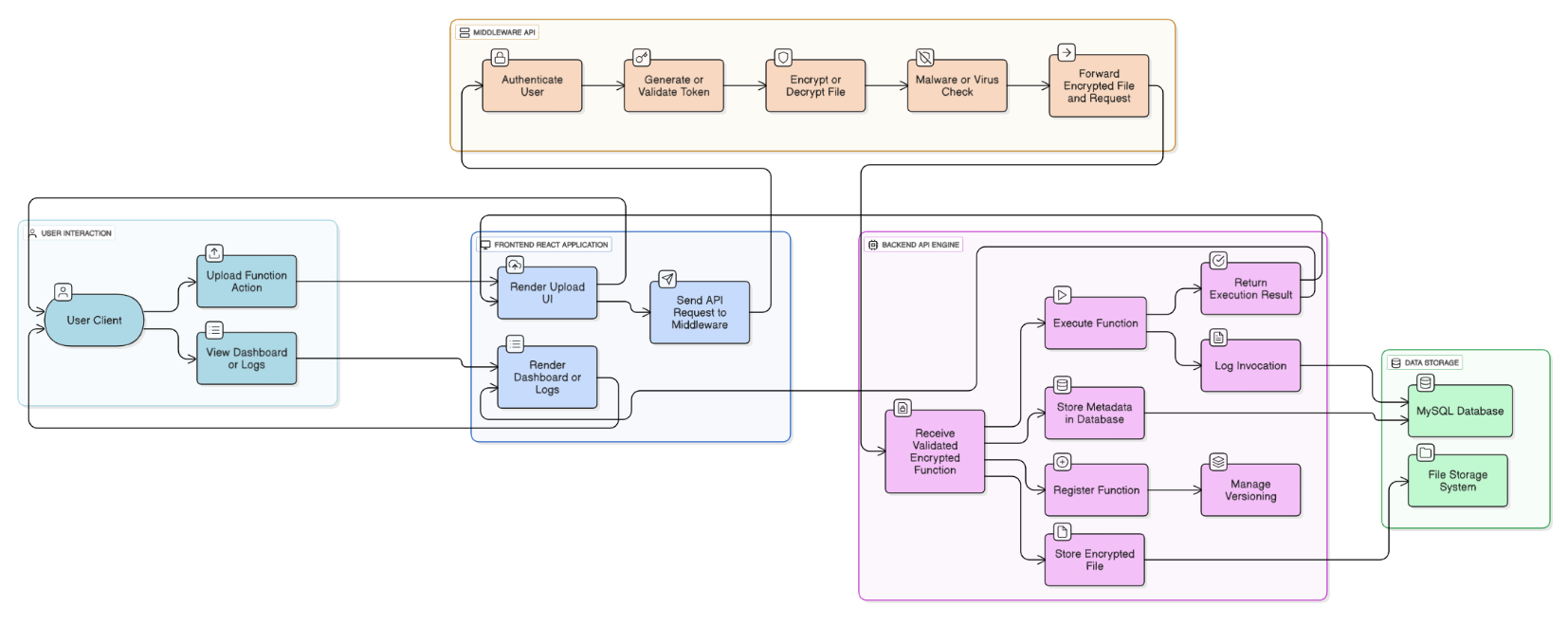
**Step 5: RabbitMQ Integration**

In order to enable asynchronous processing, I set up RabbitMQ and created a producer that is tasked with publishing function execution requests to a specified queue. A consumer module then pulls these requests and executes the necessary operations. Implementation of functions in this system was a trial-and-error process—mainly because of YAML configuration issues—though in the end, I managed to publish and consume messages and update the ExecutionLog in reflection. This system design is meant to improve the overall system via non-blocking and scalability enhancements.

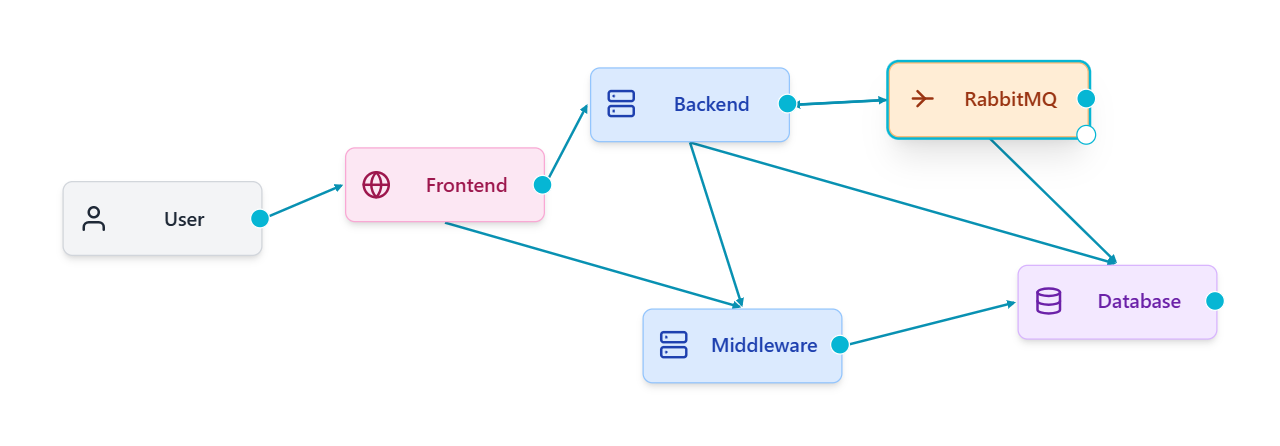
**Tools and Technologies Used Till Now:**

* Java (17) & Spring Boot.
* MySQL
* RabbitMQ
* Postman
* Hibernate / JPA
* Git & GitHub.
* Swagger.
* Maven.

I want to keep as high a level of modularity as possible, since the next stage will add middleware as a security layer, and eventually include a React frontend.



*High-Level Architecture: Overview of end-to-end workflow for function upload, execution, and result logging.*



*High-level Diagram*

**PLAN OF WORK**

So far, I would say that the project has followed closely the suggested plan, though with a number of unplanned events encountered along the way. The first step was the identification of the technological framework to be used and then the architectural design. I spent a week on this, which involved thorough reading, watching tutorial videos on YouTube, and seeking advice from some of the experienced colleagues in my office who specialize in creating similar projects. Once I had reached that point of comprehension, I then went ahead and built the backend using Spring Boot and MySQL since they are technologies that I am already familiar with.

The next big block of work was all data modeling and establishing the CRUD (create, read, update, delete) operations on users, serverless functions, and logs. I ensured all the core entities (e.g., User, Function, ExecutionLog) are correctly defined and laid out, so the database schema is future-proofed. We've done services and repositories and tested every endpoint using Postman to ensure nothing's breaking.

By the end of the first month, I had been able to implement file upload and user registration functionality, allowing users to upload files containing code that would then be linked to their respective accounts in the database. Next, I added a simulation functionality for the execution of functions where the backend would "execute" the uploaded scripts according to the respective environments of execution (for example, Python, Node.js, or Java). This has been the toughest phase, as leaving out edge cases would highly likely lead to security exploits or process failure under unexpected circumstances. However, the logic has been set well, and the initial execution is working fairly well. I recently added RabbitMQ to enable async function execution. I created a message producer that places requests onto a queue, and a consumer which takes them and runs them. It took a few tries to get the system properly set up (in all fairness, YAML is my personal worst), but now the system properly queues up execution requests and tracks their statuses. The backend is now properly set up for scaling and is actually more of an actual Function-as-a-Service (FaaS) platform than I originally thought when starting out.

**Work Completed till Date:**

* Finalized tech stack & architecture diagram.
* Spring Boot backend (running on port 8080), MySQL DB setup and connected.
* Entity classes, repositories, services, DTOs written and tested.
* REST APIs for function upload, metadata management, and retrieval.
* File upload logic & safe file handling.
* Function execution simulation for Python, Node.js, Java.
* RabbitMQ queue integration for async function execution.
* Execution logs recorded and status reflected in DB.
* Initial API documentation via Swagger.

**Plan for next phase:**

With backend execution and simple asynchronicity out of the way, the next thing on my plate is developing the middleware service. This will be run as an independent Spring Boot application (on port 8081), primarily to handle user authentication (JWT login), password hashing, and (depending on how much time there is left) spoofing malware scans of uploaded code. File encryption/decryption is another large security piece.

After the middleware is up, I will integrate it with the backend for secure API calls, and then start working on the ReactJS frontend. The UI should allow users to register/login, upload code, view the status of their functions, and check logs/results. If things stay on track, the last phase will be all about polishing, integration testing, and writing the final project report with screenshots and diagrams.

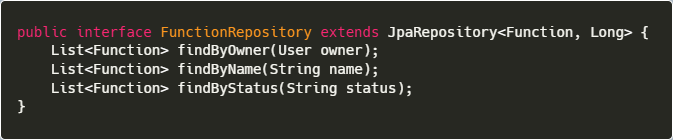
Honestly, the majority of these things may need to get tweaked on the fly, but the overall pipeline is fairly solid now. I'm trying to work smart—doing things such that features can be extended later, even if I don't have time to do everything now.

**CODE SNIPPETS**

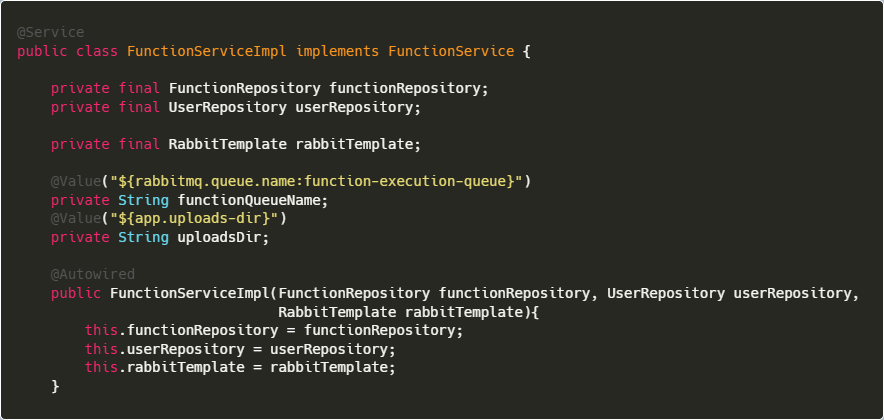
* **Function.java**- Defines the Function entity for database mapping, representing each serverless function’s metadata, owner, file details, and status.



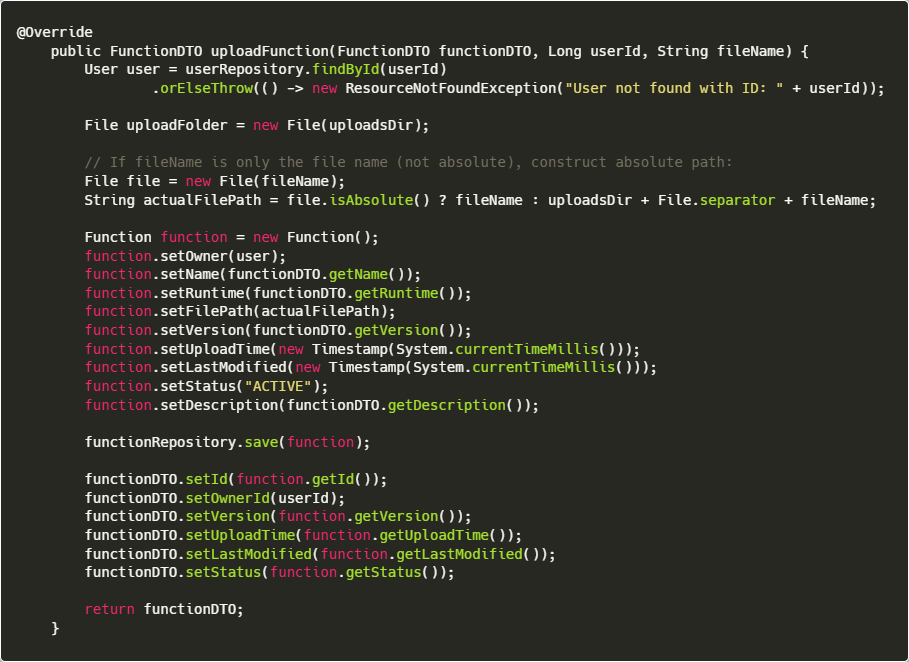
* **FunctionRepository.java** - JPA repository interface to provide CRUD operations and custom queries for the Function entity.



* **FunctionServiceImpl.java -** Implements the business logic for function operations, integrating repository and RabbitMQ for execution flow.



* Handles the storage and metadata registration for newly uploaded serverless functions.

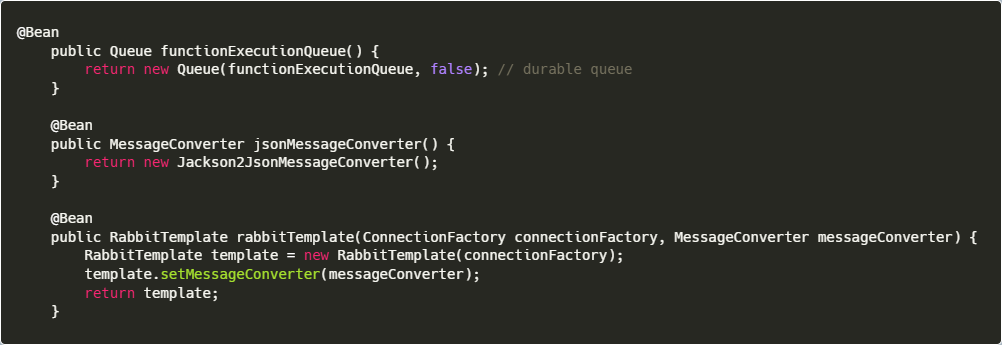


* Sends function execution requests to RabbitMQ, decoupling execution via asynchronous message queuing.



* **RabbitMQConfig.java -** Spring configuration class for setting up RabbitMQ connection, queue, and JSON message serialization.

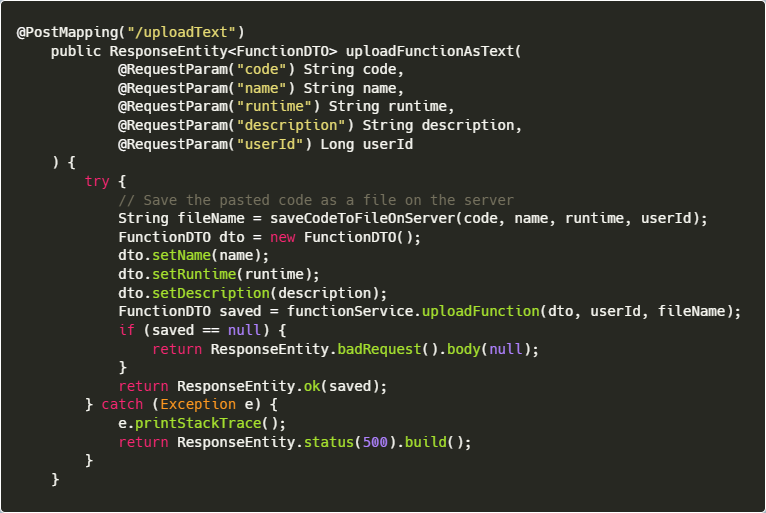




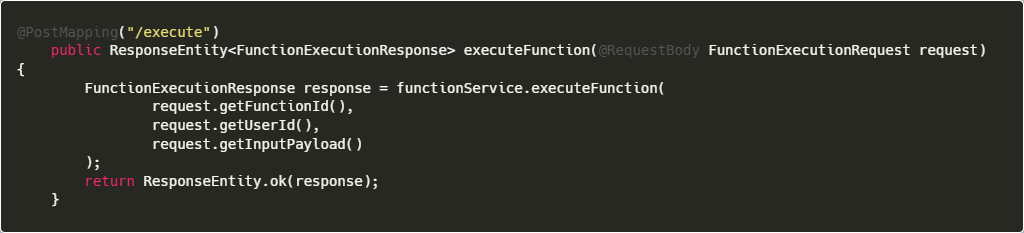
* **FunctionController.java** - API endpoint to upload new serverless functions with file and metadata.



* API endpoint allowing function code uploads via direct text input for quick deployment

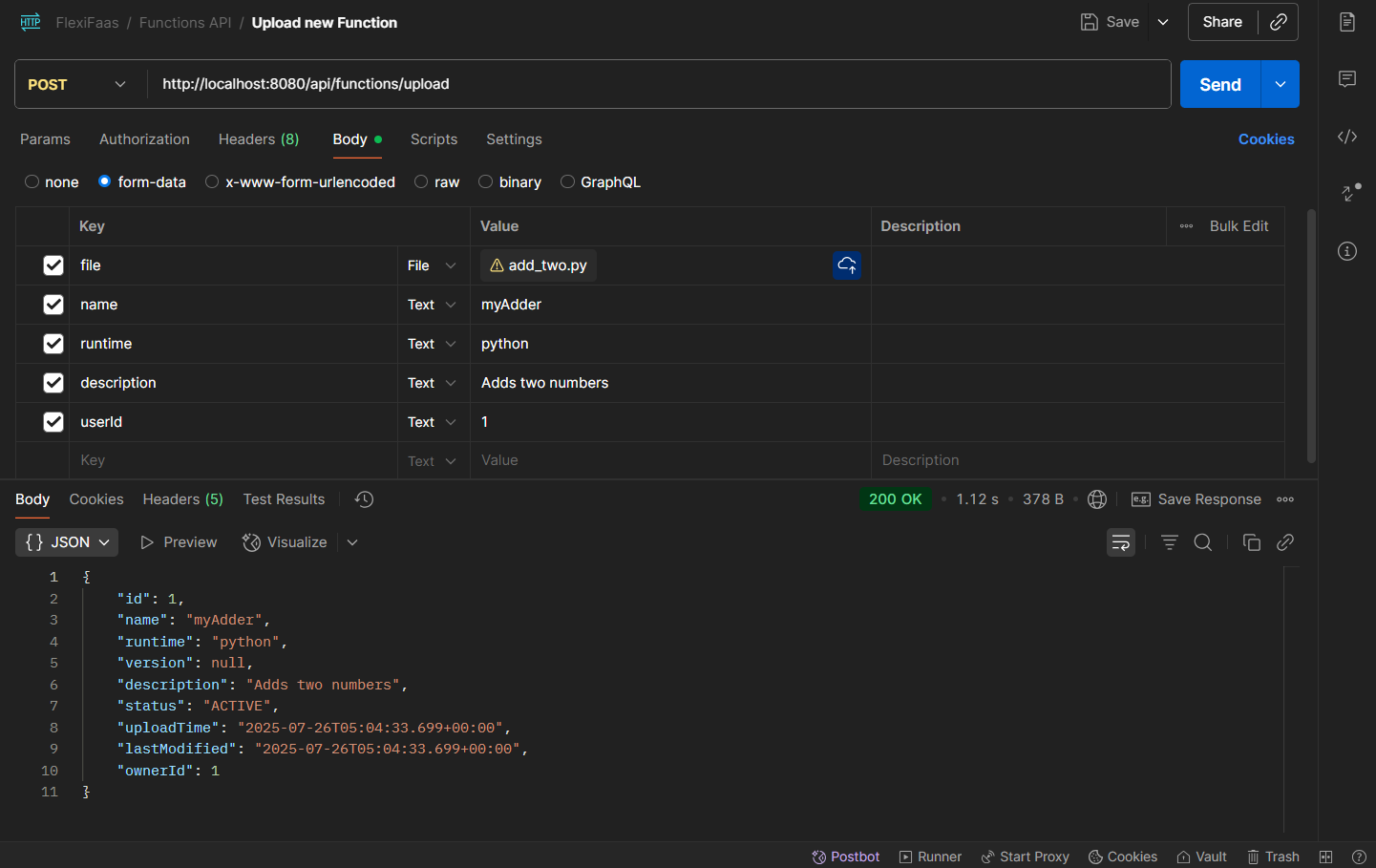


* API endpoint for triggering asynchronous execution of a deployed function via RabbitMQ.

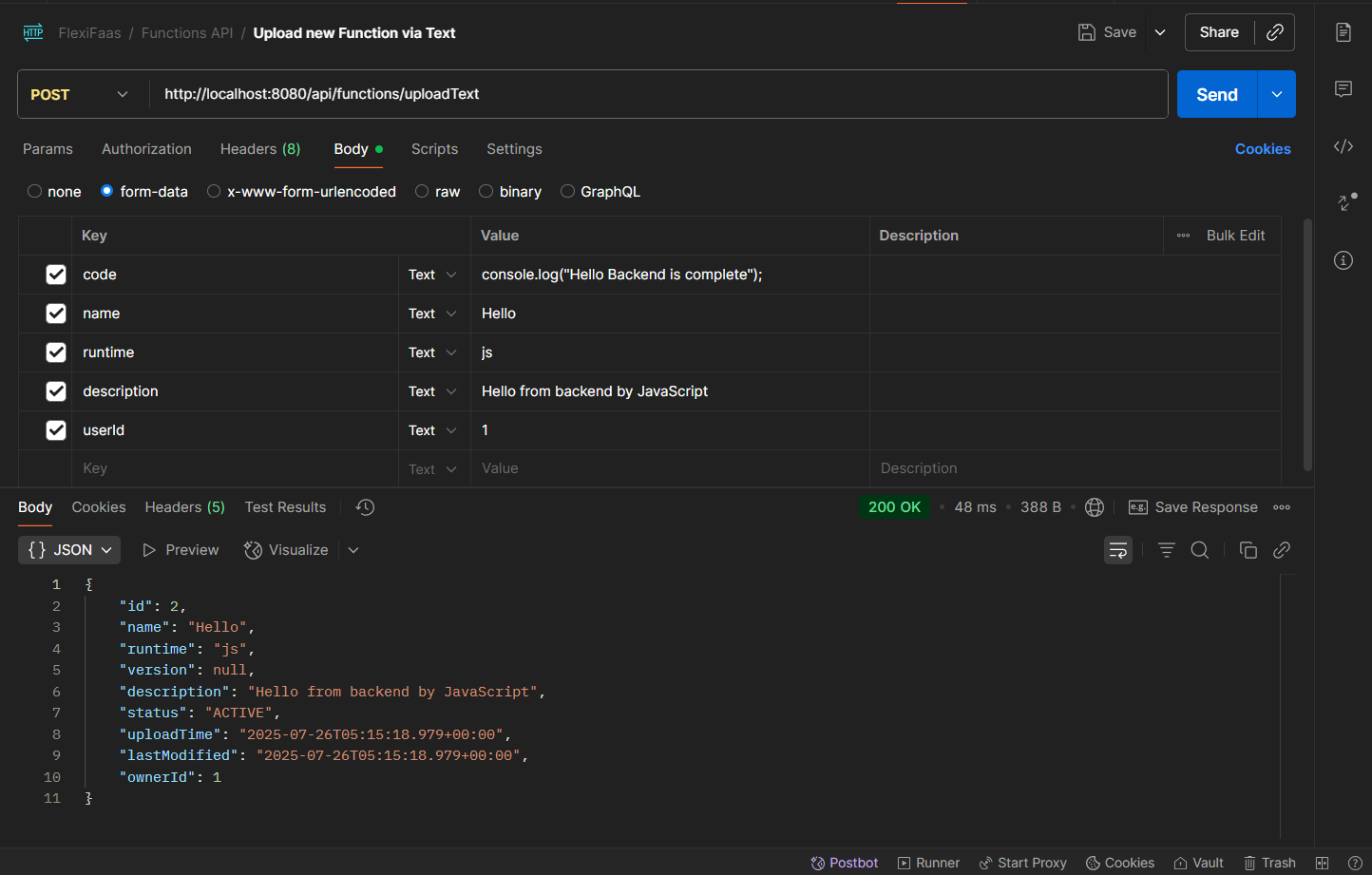


**API TESTING VIA POSTMAN**

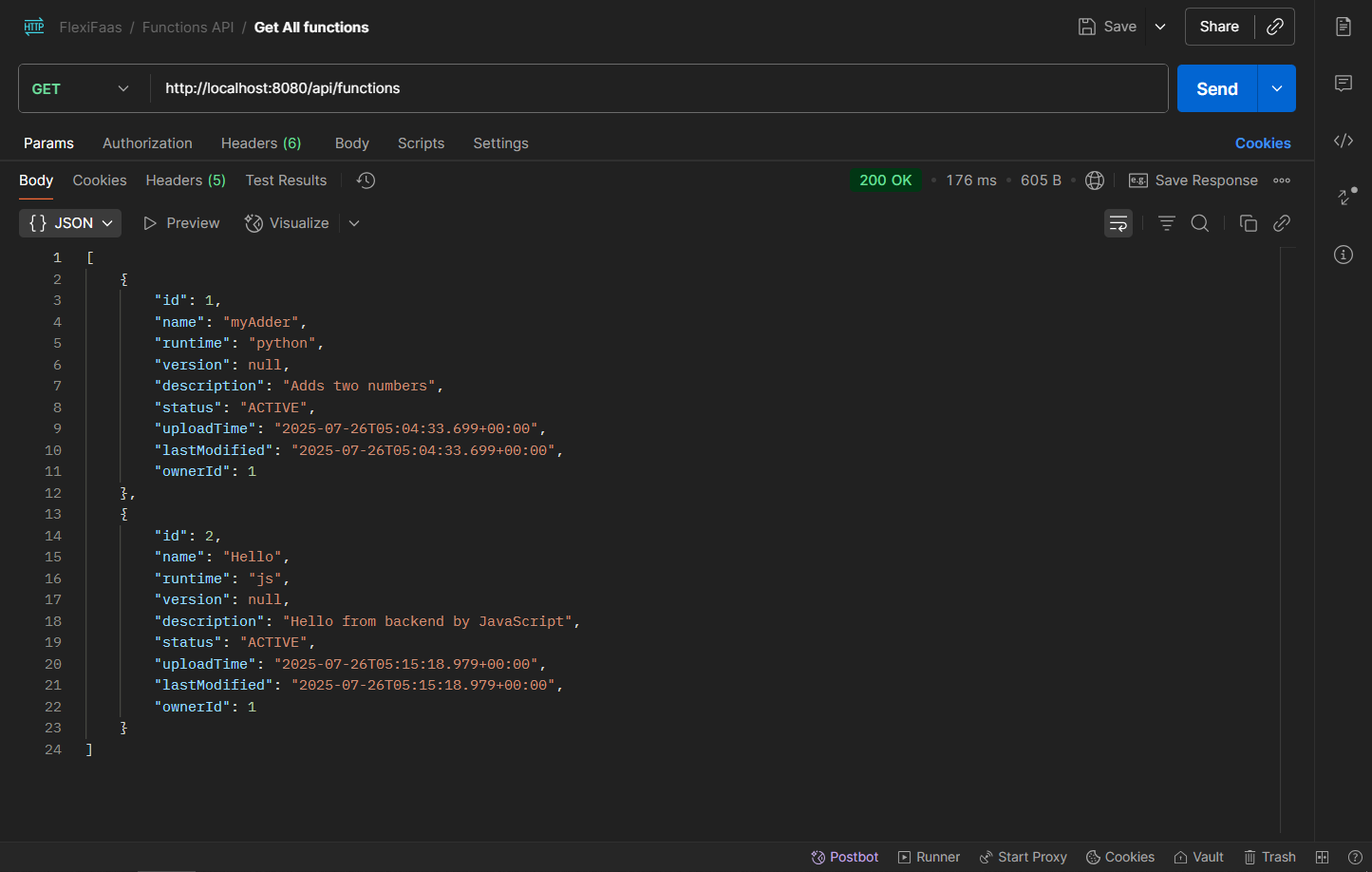
* API test for uploading a new function with file using POST /api/functions/upload

****

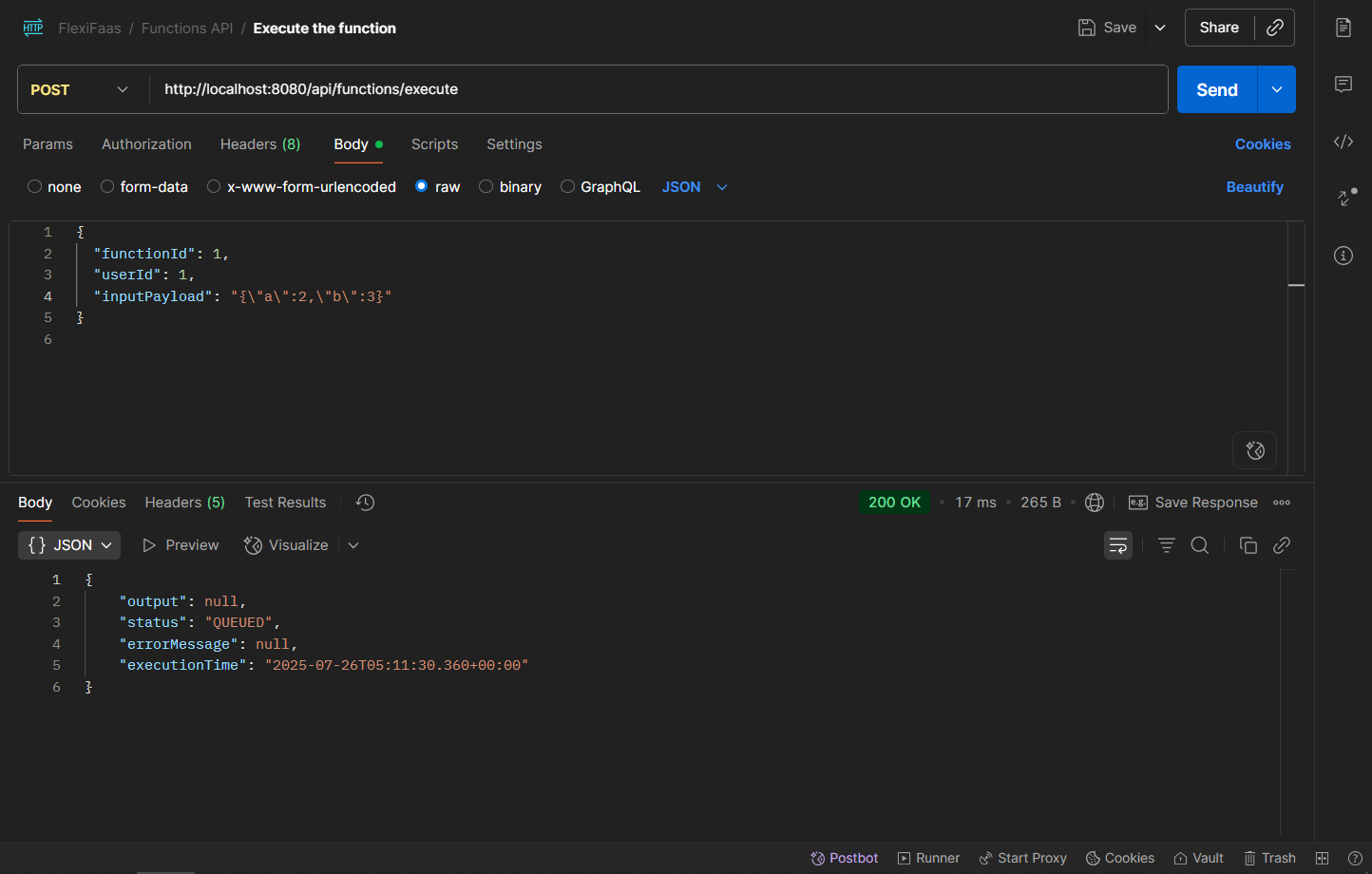
* API test for uploading a new function by source code using POST /api/functions/uploadText

****

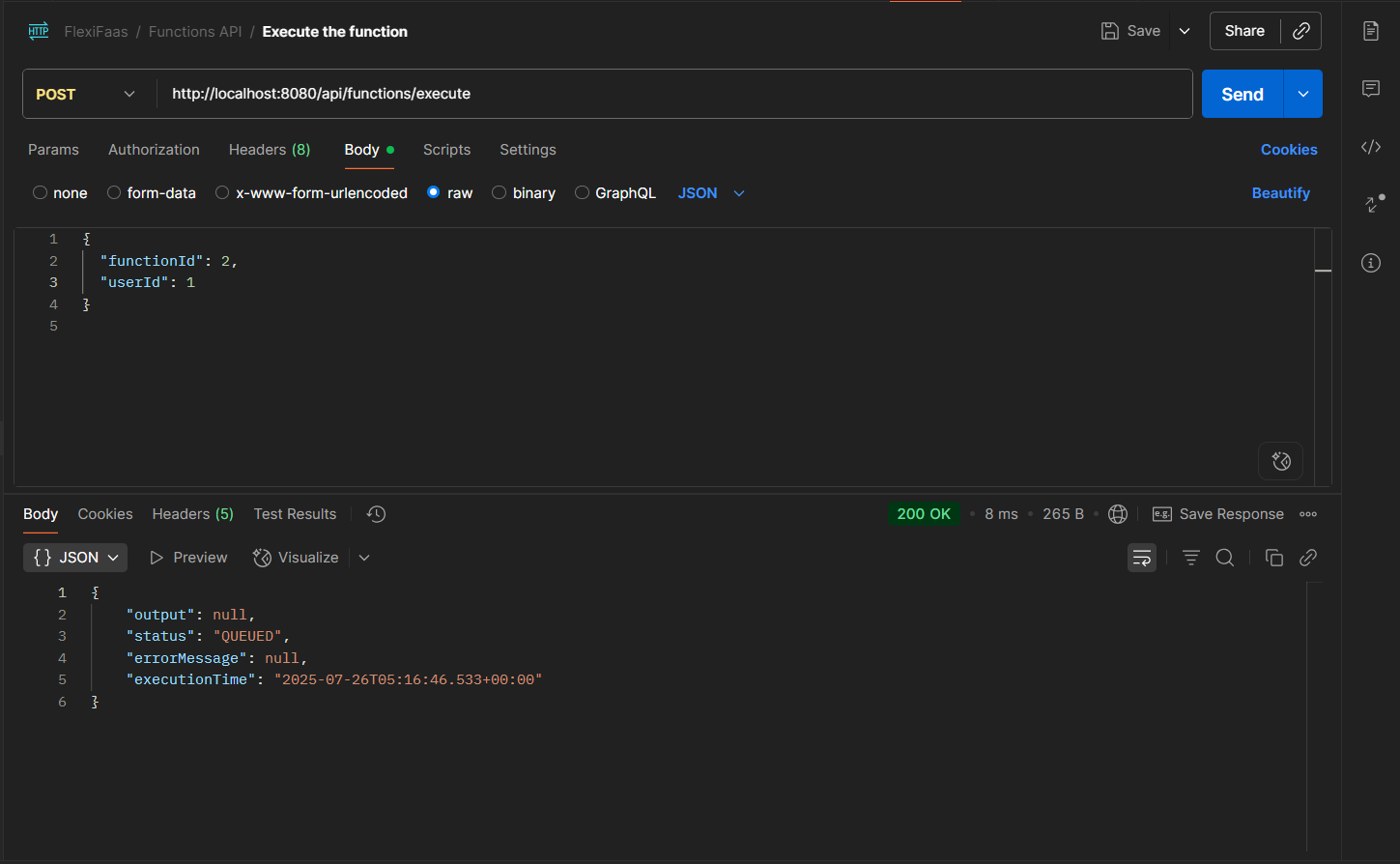
* API response for retrieving all registered functions with GET /api/functions



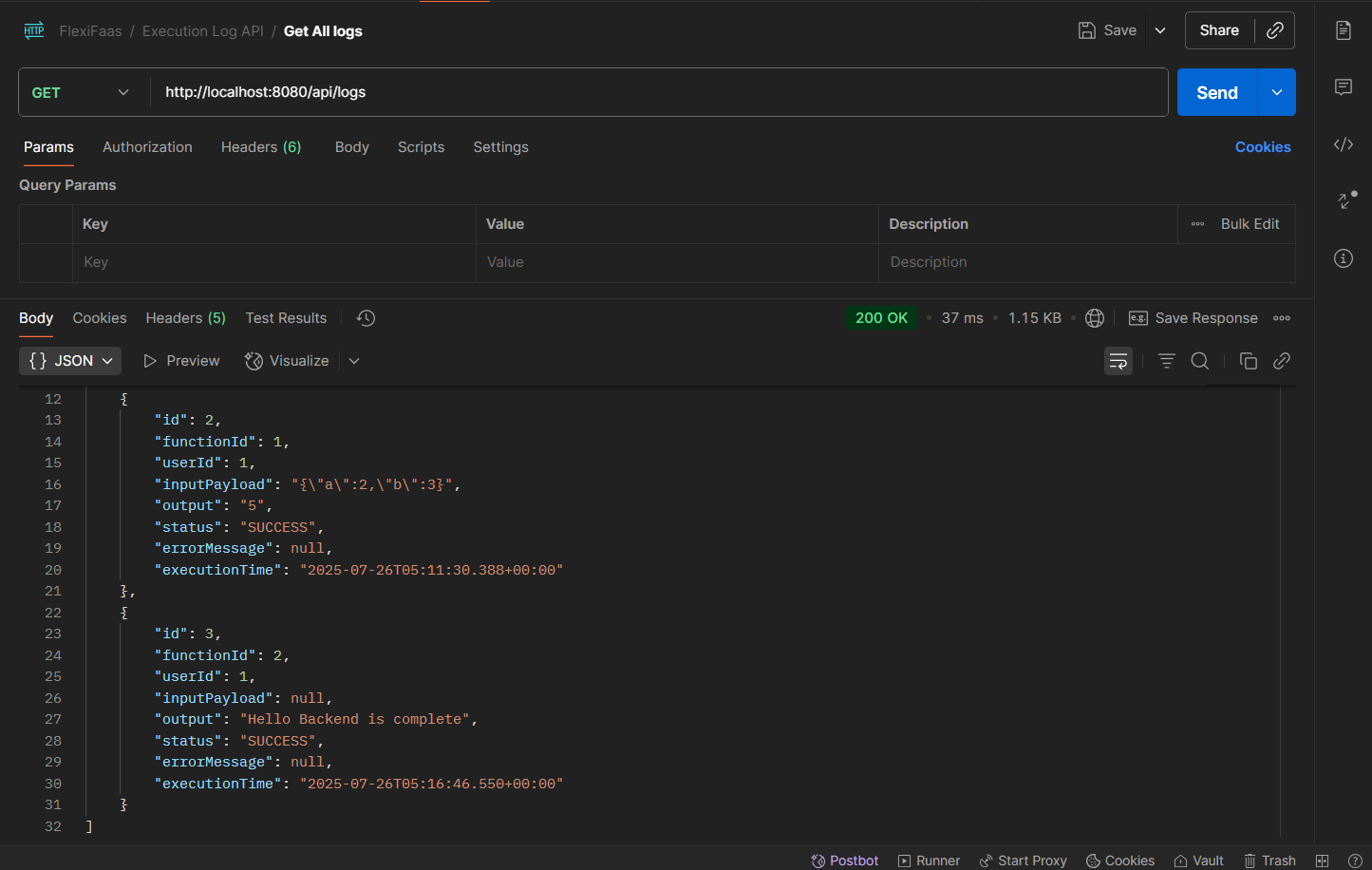
* API call to execute a function with input payload using POST /api/functions/execute

****

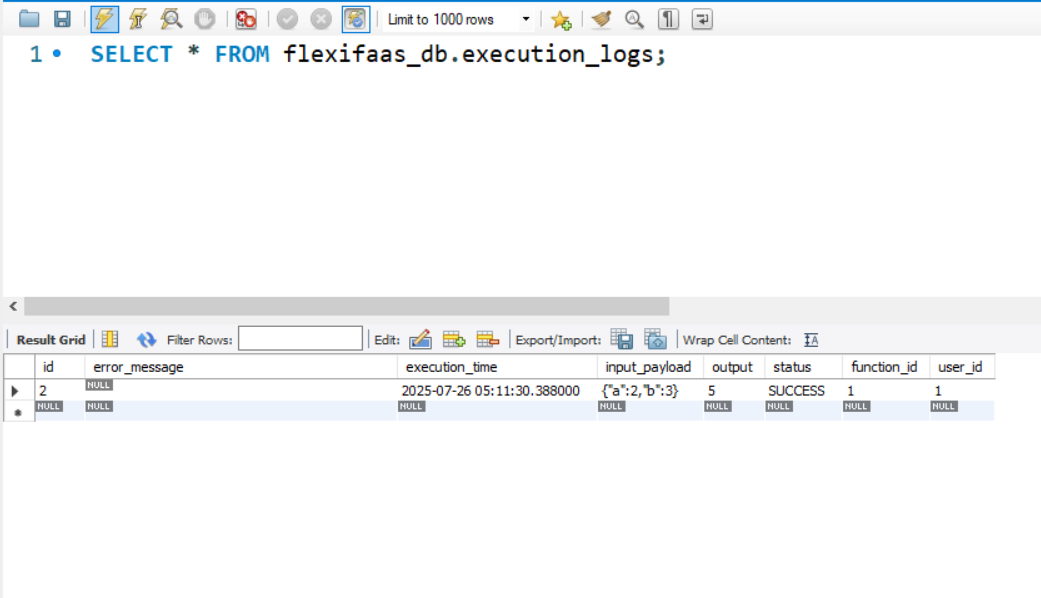
* API call to execute a function without input payload using POST /api/functions/execute

****

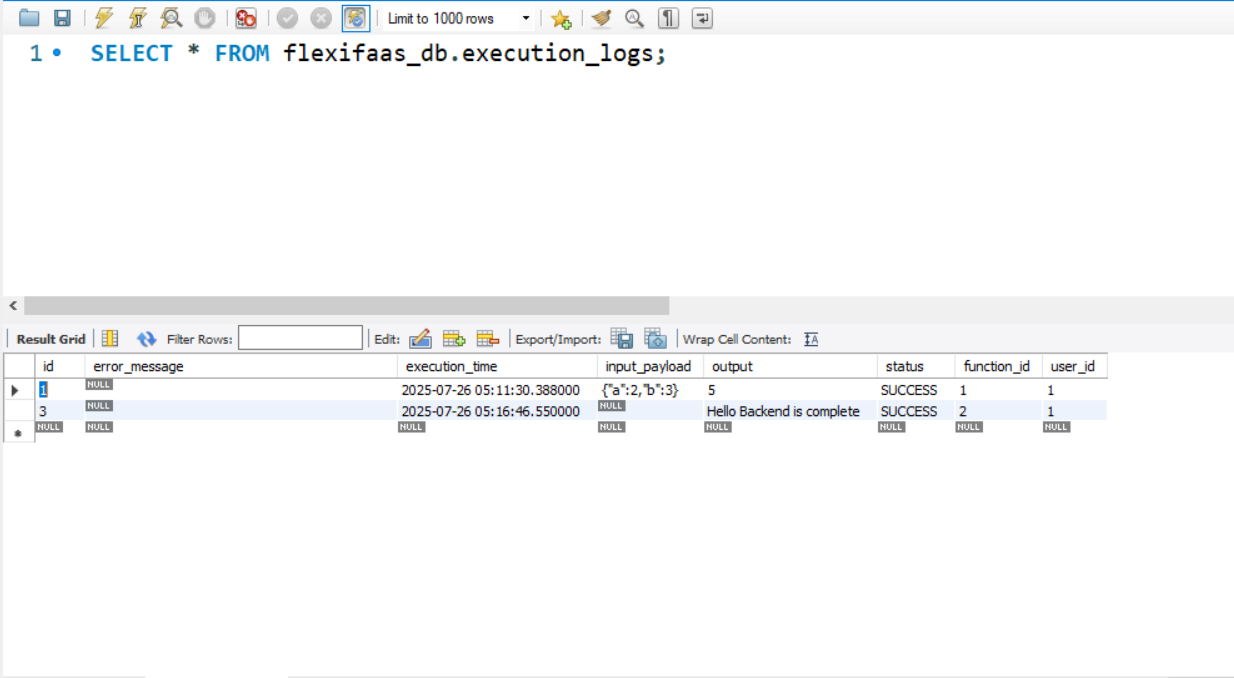
* API call to fetch all function execution logs with GET /api/logs



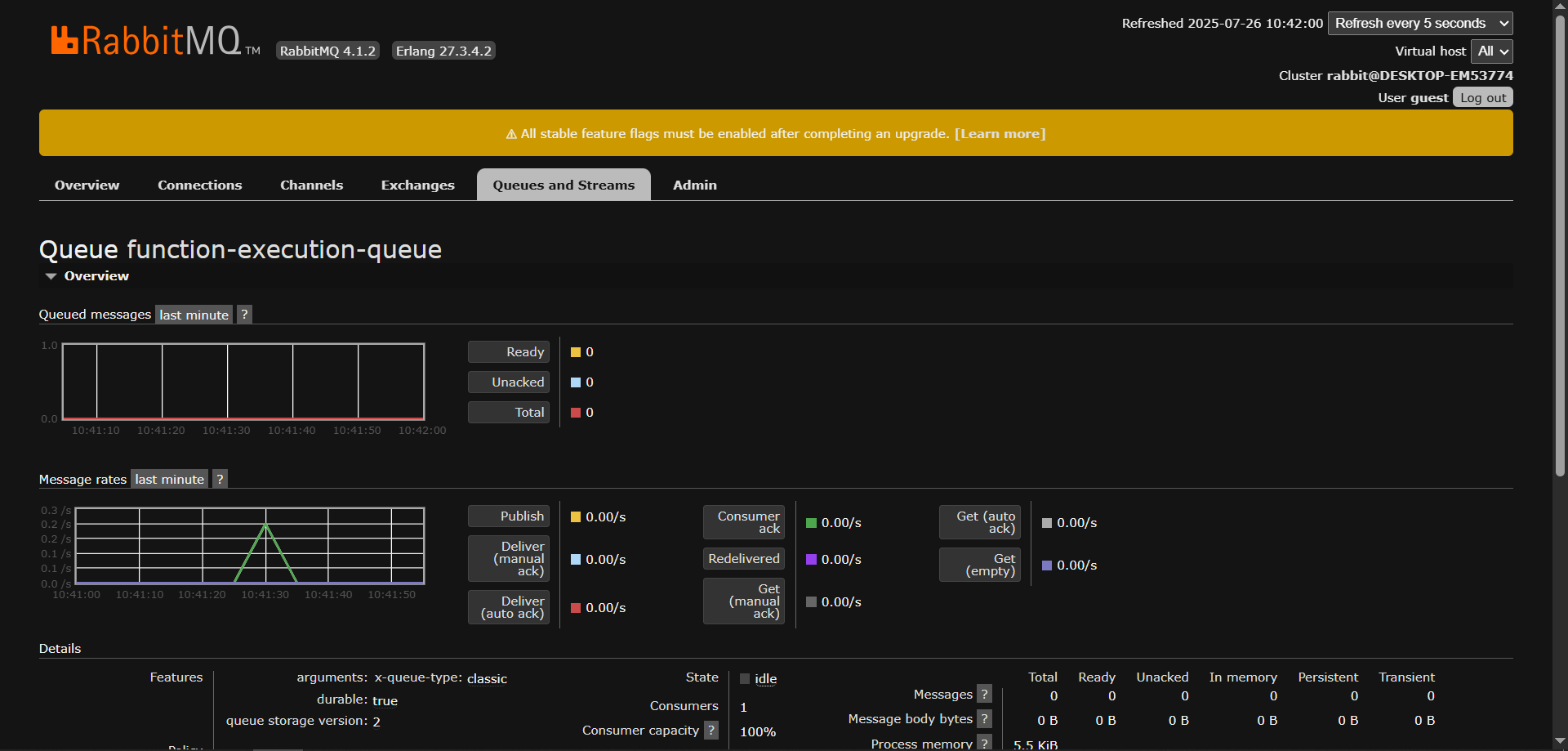
* Execution log entry stored in MySQL database after function execution



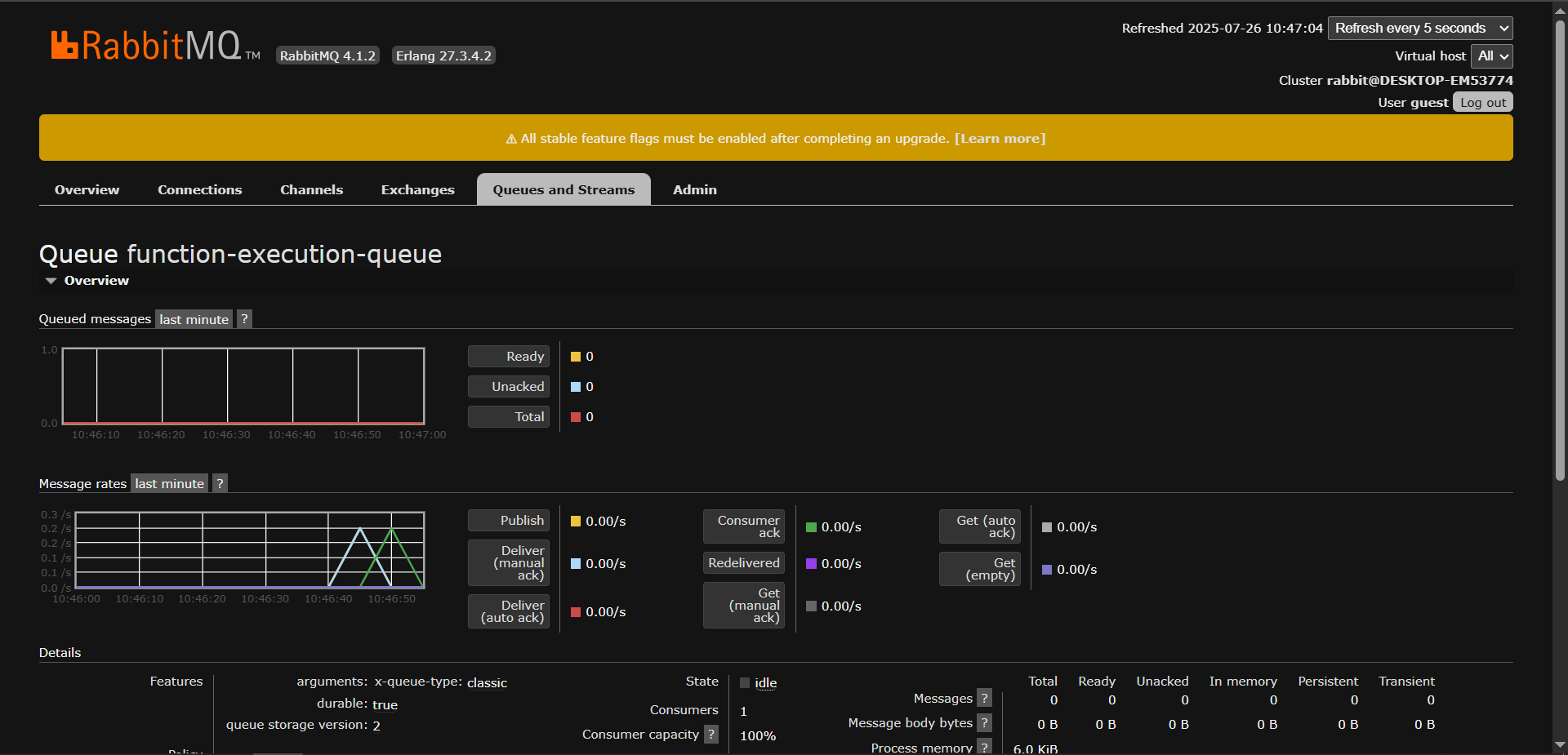
* Another sample execution log entry in MySQL database



* RabbitMQ management UI showing message queue activity for function-execution-queue



* RabbitMQ management UI confirming queue status after execution



**REFERENCES**

1. Jonas, E., Schleier-Smith, J., Sreekanti, V., et al. (2019). “Cloud Programming Simplified: A Berkeley View on Serverless Computing.”  
   <https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.pdf>
2. Spillner, J., Mateos, C., Monge, D. A. (2017). “Serverless Computing: Economic and Architectural Impact.”  
   In: IEEE Internet Computing, 21(4), 66-71.
3. Malawski, M., et al. (2020). “Serverless Execution of Scientific Workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions.”  
   Future Generation Computer Systems, 110, 502-514.
4. Documentation:
   * OpenFaaS: <https://docs.openfaas.com/>
   * Apache OpenWhisk: <https://openwhisk.apache.org/documentation.html>
   * AWS Lambda: <https://docs.aws.amazon.com/lambda/latest/dg/welcome.html>
   * Spring Security: <https://docs.spring.io/spring-security/reference/>
   * OWASP Foundation: <https://owasp.org/>

Technology & Implementation References:

1. Oracle. (2024). Java SE Documentation.  
   Retrieved from: <https://docs.oracle.com/en/java/javase/17/docs/>
2. Spring Boot Reference Documentation.  
   The Spring Team. (2024).  
   <https://docs.spring.io/spring-boot/docs/current/reference/html/>
3. RabbitMQ Official Documentation.  
   Pivotal Software, Inc. (2024).  
   <https://www.rabbitmq.com/documentation.html>
4. Maven – Project Management and Comprehension Tool.  
   Apache Software Foundation. (2024).  
   <https://maven.apache.org/guides/index.html>
5. Hibernate ORM Documentation.  
   Red Hat, Inc. (2024).  
   <https://hibernate.org/orm/documentation/>
6. MySQL 8.0 Reference Manual.  
   Oracle Corporation. (2024).  
   <https://dev.mysql.com/doc/refman/8.0/en/>
7. Baeldung. (2023). *Guide to Spring Boot and RabbitMQ*.  
   <https://www.baeldung.com/spring-amqp-rabbitmq>
8. Stack Overflow. (n.d.).  
   Various posts on error handling, Spring Boot config, and RabbitMQ setup.  
   <https://stackoverflow.com>
9. GeeksforGeeks. (2024). *Spring Boot Tutorials & Examples*.  
   <https://www.geeksforgeeks.org/spring-boot-tutorial/>
10. W3Schools. (2024). *Java File Handling*.  
    <https://www.w3schools.com/java/java_files.asp>