



**UNIVERSITY
OF OULU**

FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

**Mikael Marin
Teemu Puro**

**IMPLEMENTING A TICKETING
MICROSERVICE FOR LOVELACE**

Bachelor's Thesis
Degree Programme in Computer Science and Engineering
April 2026

Marin M., Puro T. (2026) Implementing a Ticketing Microservice for Lovelace.
University of Oulu, Degree Programme in Computer Science and Engineering, 33 p.

ABSTRACT

This bachelor's degree thesis describes the process of designing and developing a ticketing microservice for the Lovelace learning environment. There are multiple aspects to this process, including related software, design choices, implementation, evaluation and discussion. This document dissects all the necessary and optional design choices when implementing the needed functionality for this microservice, including logic, data management, security and scalability. Although done for the Lovelace platform, the main focus of this project is not to implement an "out of the box" solution, but to design a robust structure so as to have the backend of the microservice ready for potential future implementation.

Keywords: B.Sc. degree, rest, api, asgi, python, database, async, sql, http

Marin M., Puro T. (2026) Tikettimikropalvelun toteutus Lovelace-alustaa varten.
Oulun yliopisto, Tietotekniikan tutkinto-ohjelma, 33 s.

TIIVISTELMÄ

Tässä kandidaattivaiheen opinnäytetyössä käydään läpi koko prosessi liittyen Lovelace-alustaa varten kehitetyn tiketöintimikropalvelun suunnittelusta ja toteutuksesta. Tämä prosessi sisältää päälliskatsauksen käytettyihin ohjelmistoihin, suunnitteluun, toteutukseen, työn arviointiin, sekä lopputuloksen pohdintaan. Tässä työssä käsitellään perusteellisesti tähän projektiin valitut suunnitteluperusteet ja arkkitehtuuripäätökset suhteessa haluttuun ja saatuun lopputulokseen. Vaikka Lovelace-alusta on keskeisessä roolissa projektia tarkastellessa, tavoiteltuun lopputulokseen ei kuulu täysin valmis käyttöliittymä tikettijärjestelmää varten. Lopputulos, johon tähtäämme on vakaa tausta-arkkitehtuuri tikettijärjestelmälle, jonka "päälle" voi toteuttaa halutun käyttöliittymän ja käyttötarkoituksen.

Avainsanat: kandityö, rest, api, asgi, python, tietokanta, async, sql, http

TABLE OF CONTENTS

ABSTRACT	
TIIVISTELMÄ	
TABLE OF CONTENTS	
FOREWORD	
LIST OF ABBREVIATIONS	
1. INTRODUCTION.....	8
2. RELATED WORKS.....	9
2.1. Ticketing Services.....	9
2.2. Microservices	9
2.3. Data Storage.....	10
2.4. API.....	10
2.5. REST	10
2.6. ASGI	11
2.7. Web Framework	12
2.7.1. Django.....	12
2.7.2. Flask	12
2.7.3. FastAPI	13
2.8. Uvicorn.....	13
2.9. Lovelace	13
2.10. MariaDB	13
2.11. Security Risks.....	14
3. DESIGN.....	15
3.1. Tickets	15
3.2. API Design.....	15
3.3. Interaction	17
3.4. Security Considerations.....	18
4. IMPLEMENTATION	19
4.1. Process.....	19
4.2. Technology Choices.....	19
4.3. Architecture	19
4.3.1. Data Structures.....	20
4.3.2. API Architecture	21
4.3.3. MariaDB Architecture	21
4.4. Security Implementation	23
4.4.1. Access Control.....	23
4.4.2. Database Security.....	23
5. EVALUATION	24
5.1. Evaluation Plan.....	24
5.2. Functionality	24
5.3. Performance	24
5.4. Quality	27
5.5. Security.....	27
5.6. Agility	27

- 6. DISCUSSION 28
 - 6.1. Reflection..... 28
 - 6.2. Future Work 28
- 7. CONCLUSION 30
- 8. REFERENCES 31
- 9. APPENDICES..... 33
 - 9.1. Project Time..... 33

FOREWORD

This project was started during the Applied Computing Project 1 course in University of Oulu. We would like to thank Timo Ojala for general instructions and guidance during the course. More importantly, we want to thank Mika Oja, who supervised this project and provided us with valuable feedback and insight.

Oulu, April 7th, 2026

Mikael Marin
Teemu Puro

LIST OF ABBREVIATIONS

CSE	Computer Science and Engineering
ASGI	asynchronous server gateway interface
WSGI	web server gateway interface
API	application programming interface
REST	representational state transfer
RPC	remote procedure call
RDBMS	relational database management system
HTTP	hypertext transfer protocol
SSL	secure socket layer
SQL	structured query language
DDoS	distributed denial-of-service
GUI	graphical user interface

1. INTRODUCTION

During this Bachelor's Thesis, a new ticket-based communication feature is designed and implemented for the learning environment Lovelace. The goal of this feature is to make it possible for students to communicate with teachers through the Lovelace platform, reducing current reliance on third-party communication platforms like email for service oriented requests (such as asking for course grades).

Lovelace already offers a feedback field for all exercises, but this ticketing feature is more interactive and it gathers feedback in one location for easier processing. The service will make it easier to send direct, course-related questions and report errors or other issues in the learning material. Simplifying this communication process helps teachers improve course quality and encourages students to participate and seek assistance, which has been shown to improve academic success[1].

This feature allows users to create support requests (tickets) concerning a specific course, which are then sent to a persistent data store, becoming available for retrieval by teachers linked to that course. When a user views their tickets, Lovelace provides the authorization token of the user, which is then used by the ticketing service to return all authorized tickets from the data store. Lovelace will then present these tickets to the user in its own format.

This feature will be implemented as a microservice. Microservice architecture is popular in modern software development, as it facilitates powerful freedom in how cooperating services can be built. Technological choices are not bound by the partner services, and each service can be developed and deployed individually. Due to this architecture choice, the ticketing service does not need to know how Lovelace operates and vice versa. Only communication between these two must use the same, agreed upon format. This microservice can also be used by other services, as long as the communication format is not changed.

As a microservice, the scope of this project is very specific. A data access handling application programming interface (API) will be developed, which will communicate with Lovelace on Lovelace's initiative. Lovelace will provide user authentication, and the ticketing service will provide secure handling of the tickets, until they are processed by Lovelace. User interface is not within the scope of this project, the reasoning behind this decision is explained in section 3 of this document, concerning design.

In this project, existing software and architecture implementation styles will be utilized; MariaDB[2] for data storage, FastAPI[3] for API development and representational state transfer (REST) as service architecture. Deployment will be handled with the asynchronous server framework Uvicorn[4]. These are described with more detail in the next chapter.

2. RELATED WORKS

2.1. Ticketing Services

Ticketing services are platforms for instanced, non-real-time communication that involves leaving tickets to be later handled by a separate team. These tickets have a specification detailing their content, which is curated by the service's owner to contain everything the team might need to resolve the tickets. Usually tickets store at least a message, sender identification, and a timestamp. Ticketing services are often used to serve customers or facilitate teamwork by tracking requests and issues.

The main advantages of using ticketing systems are agility and ease of use. When submitting or processing a ticket, people are not tied to any specific physical place and people do not have to worry that the ticket finds the right person. The agility of these systems also includes their modularity. The ticket parameters can be tailored for a specific use case, so that the tickets can have specific fields and attributes instead of just simple freeform text.

As a more technical and concrete example of a ticketing service, a document on designing an IT-helpdesk using microservices was discovered during research[5]. The ticketing system design in the document is somewhat similar to the one created during this project, since both have the same main fields; category, message, and header.

2.2. Microservices

Microservices are an important aspect of making modern software applications. The main idea behind microservices is to break down large, monolithic applications into smaller, more easily manageable and modular pieces. This allows the developers to build on top of existing frameworks without having to familiarize themselves with everything under the already existing infrastructure[6]. Communication between different applications is done through APIs, which simplifies the process of communication significantly, while being flexible enough to adapt to different scenarios and use-cases. Great advantages of microservices include, but are not limited to:

- Resilience, which means that the overall system can still work even though some services do not, due to the modular and decoupled design of microservices.
- Scalability, which allows the microservice to run more efficiently since resource requirements and demand can be scaled independently for each microservice.
- Diversity, which means that the microservice in question can be developed with whichever tools the developer sees fit, due to the fact that each microservice is a separate component.
- Flexibility, which was briefly touched on earlier. This allows for easier maintenance and upgrading possibilities, due to requiring minimal changes to the existing overall system[7].

However, there are still some relevant issues with microservices, most prominent ones being complexity and data consistency. Complexity can become a real issue when

the amount of microservices starts to be at a point where managing and coordinating the separate components requires a lot of additional work. Data consistency on the other hand starts to be at a higher risk when the data has to travel through multiple API layers[7].

2.3. Data Storage

Data storage is a key component in many software projects. Although there are a multitude of different databases and database structures, they can easily be divided into two different main types; Relational databases (RDB) and Non-relational databases (usually called NoSQL databases). In RDB's, the data is stored in tables, which consist of rows and columns (such as in a spreadsheet). Each table has a key, which can be linked to other tables, allowing for fairly complex relationships between tables. The main attributes of RDB's include its rigid structure, data consistency, and durability. Hence RDB's are a solid choice when storing non-complex, consistent data for a long period of time. Some popular RDB's are Oracle, MySQL, and PostgreSQL (which Lovelace uses). Oracle is the most popular option for commercial use, but it requires a paid license. MySQL and PostgreSQL are good open source alternatives, both of which are well suited for web applications.

The more modern (yet still quite old) NoSQL database structure is more focused on speed and scalability, often used for more performance-oriented use cases such as analytics, social networks, etc. The different types of NoSQL databases vary quite a lot in how they store and organize data, but they are generally grouped into four main categories: key-value stores, document databases, column-family databases, and graph databases. Unlike RDB's, the overall idea in NoSQL is to build a more "free-form", schema free database, so that the data structures do not have to strictly adhere to predefined constraints. In a nutshell, RDB's prioritize structure and consistency, while NoSQL's strength lies in flexibility, scalability and performance[8]. Popular choices for NoSQL databases include MongoDB and Redis, both of which are quite flexible, since they support multiple NoSQL types.

2.4. API

In general, API (Application Programming Interface) works as a "middle layer" for different software components. Most common types of APIs are web APIs (such as this project) and operating system APIs. Web APIs allow for interaction between different web services that would potentially otherwise be incompatible with each other due to differences in architecture. The core benefits of developing and using APIs is their flexibility for developers and simplicity for end-users[9].

2.5. REST

REST (REpresentational State Transfer) is an architectural style for distributed information systems. It is frequently built with the hypertext transfer protocol (HTTP)

as the communication medium, but it can work with other protocols too. REST is based on the idea that all singular or combinations of information, objects, concepts and services are represented as resources that the user may access[10]. These are communicated to users as “representations”, which includes the information and metadata with details about it. In comparison to remote procedure call (RPC), REST provides more relevant functionality for this project’s use-case. RPC is more about actions towards different components in the system, opposed to REST’s representation of components as accessible resources[10].

2.6. ASGI

ASGI stands for Asynchronous Server Gateway Interface, which is a protocol for communication between servers and Python web-applications. It allows for asynchronous programming for more scalable and efficient web applications. Its origins lie in the Web Server Gateway Interface (WSGI), which had shortcomings in long-lived connections and non-blocking I/O requirements[11].

ASGI uses an intermediate server to catch and decode HTTP communications, parsing these requests and responses into scopes and attaching communication events, which the main application will use to process and respond to the request. Scopes serve as a description, containing information like HTTP method and protocol version, paths and ports of both client and server, headers, and query strings. Events, most notably *send* and *receive*, handle communication by initiating transactions and transferring data. Figure 1 below illustrates how HTTP requests are handled in an ASGI-framework.

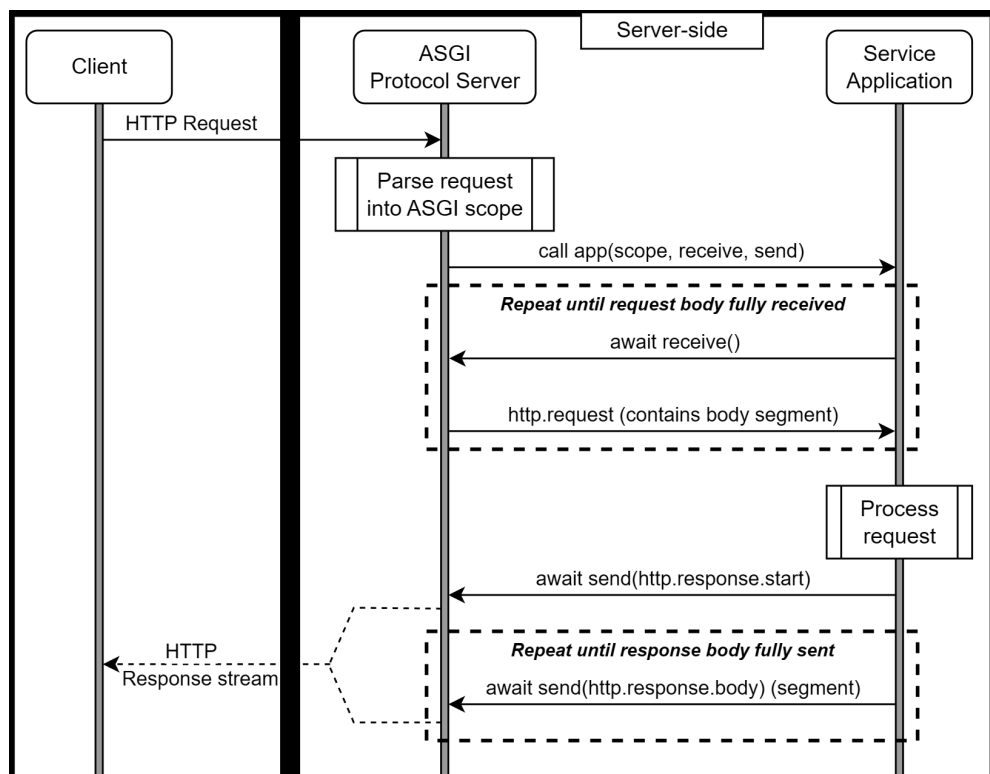


Figure 1. HTTP communication flowchart in an ASGI framework

2.7. Web Framework

The basic idea of a web framework in a software project is to provide a standardized way to build web applications, providing built-in tools to interact more easily with the different aspects of web development, such as handling HTTP requests, URL-routing, etc. In short, a framework allows for a consistent and (relatively) simple implementation of the web service, where mainly the application specific code is left for the developer to write[12].

Based on our limited experience and the scale of this project, developing this service without a pre-existing framework for guidance was out of the question. When choosing an existing framework for our API, we started by conducting surface-level research on popular options, quickly honing in on Python-based solutions due to the coding language's simplicity and the fact that Lovelace itself is based on Python, avoiding unnecessary additions to its tech stack and maintenance difficulty.

From the Python-based web frameworks, three candidates seem to outshine the competition; Django, Flask, and FastAPI. Following subsections cover their main advantages and weaknesses. All three share the benefits of good documentation and scalability.

2.7.1. *Django*

Django[13] (2005) is a massive full-stack framework that has useful features for almost any project, which would likely reduce the list of features we need to implement ourselves. One of these simplified features is database management with Django's ORM system, which would help with conversions between Python objects and database data. Despite the large feature-set, Django maintains sufficient performance for web applications.

Due to the apparent goal of satisfying as many needs as possible, Django is unavoidably quite "heavy". It has many features that would only get use in a large project that also covers client functionality, unlike our relatively small web service. This leads to reduced performance and more complex development process for suboptimal benefits. Ultimately, it is unlikely our project would benefit enough to balance the added complexity.

2.7.2. *Flask*

Flask[14] (2010) is a very lightweight framework that tries to avoid unnecessary features, resulting in a more flexible and customizable development experience. Simplicity and lightweight design make Flask a great option for user friendliness and easy performance.

An obvious consequence of the extreme lightweight design is lack of pre-built features. Flask provides basic web functionalities and optional plugins for security and data-handling features. Despite Flask vastly reduced size compared to Django, its performance is only moderately better, and FastAPI is still faster.

2.7.3. *FastAPI*

FastAPI[3][15] (2018) is the newest of these three frameworks, and so its developers have had the opportunity to learn from the older frameworks' mistakes and successes, focusing their efforts on performance and ease of use. It fully supports the modern OpenAPI specification for easier API documentation and extends an ASGI-based framework Starlette, which should make writing asynchronous web service code easier.

Being a relatively new framework, FastAPI hasn't had as much time to refine and improve as the older options. This also affects the amount of major web services using it, which in turn reduces the amount of available examples and user experiences to learn from. This can be seen in the reduced variety of compatible modules. As a lightweight framework, FastAPI has the expected difficulty of lacking more specialized features.

2.8. **Uvicorn**

Uvicorn is a fast, high-performance ASGI-server, most commonly used to run Python web applications that use the ASGI-framework. Such frameworks being for example FastAPI, Quart and Starlette. It is built on top of asyncio framework, providing developers with the ability to write asynchronous code that can handle a large number of simultaneous events and connections[4]. This is crucial for a fast and reliable API.

Other potential ASGI-server alternatives include servers such as Hypercorn and Daphne. These servers have their own strengths, but are not suited for this project as well as Uvicorn. Key point being that Hypercorn has many advanced extra features that are not required in this project, and Daphne is overall more suited for Django-based frameworks[11].

2.9. **Lovelace**

Lovelace is a virtual learning environment originally developed by Miiikka Salminen, used in University of Oulu to teach different programming courses, mostly related to computer science and engineering (CSE). It is based on Django, which is a Python web framework most commonly used by high-level developers and even large organizations to build web applications[13]. Lovelace also uses other services such as RabbitMQ, Redis, Apache, Nginx and PostGres to support different functionalities of the platform[16]. The main focus in this project does not lie in Lovelace, but there is some interaction. Most notably Lovelace is bound to be the interaction interface between students, their tickets, and staff. The API also needs authorization tokens provided through Lovelace to know which tickets can be accessed.

2.10. **MariaDB**

This is the data persistence solution used in this project. It is a popular relational database management system (RDBMS) developed as an open source project using C

and C++ programming languages. It is based on MySQL, and was started by some of the original developers of MySQL. MariaDB offers a large amount of features important for this project. Security is improved with features such as encryption of both storage and connections[17], proxy service MaxScale[18] and user validation. Data integrity is ensured through backups[19] as well as various logs for errors and activity[20]. These features and more are well documented in the knowledge base cited above, which helps us implement the most fitting features into this project, for example potential encryption and backups.

2.11. Security Risks

Security concerns must be addressed to the full extent in almost every software project. In our this project, API security is crucial. Common API security risks include authentication-based attacks, authorization errors, denial-of-service attacks, and a wide scope of miscellaneous vulnerability exploits[21]. Authentication-based attacks require the attacker to obtain an authenticated client's credentials, most commonly the API key or token. This in turn exposes the API to malicious use. Authorization errors, on the other hand, can give the user broader access than intended, opening up the possibility of unauthorized actions. Denial-of-service attacks remain a widely used method, with the aim to disrupt and block access to a certain service. In theory, these attacks may seem quite harmless, but they can pose a serious threat when directed at critical services, such as national infrastructure. As our API includes a structured query language (SQL) database, we must also be aware of and minimize possible SQL injection attacks, which rank among the most common database attack types. In worst case scenarios, these injection attacks have the potential of wiping out entire databases.

3. DESIGN

Due to this service being implemented as an API for services instead of providing tickets straight to end users, focus during the design process was on architecture instead of interface. The technological choices derived from initial requirements for the service, which are described below. Tickets are a fixed data type with consistent fields (sender, state etc.), so SQL was chosen instead of noSQL as the project's data storage style. SQL performs better when the data is stored in an unchanging format. From available SQL databases, MariaDB was the most fitting, due to its focus on security and speed, as well as availability and scalability to allow for expansion in the future.

3.1. Tickets

When querying and updating the database, tickets are processed as separated data fields, such as user, message and date. When communicating with Lovelace, tickets are formatted as JSON objects, which are easy to unpack for the user or database. We decided to go with JSON for this project due to its lightweight design and simple parsing.

The initial requirements for the tickets are as follows:

- All tickets have a unique ID assigned to them.
- The tickets can be accessed according to the user's authority level.
- The tickets are assigned to a category, and new categories can be created.
- The ticket includes a subject and a short message explaining the problem at hand.
- The tickets are assigned to a specific course.
- The tickets have the ability to be in a specific state (pending, processing, delayed, done).
- Students can see the details of their own tickets.
- Staff members can select pending tickets to personally start processing them.
- Staff members can retrieve pending tickets and those they have selected for processing.
- Specific authorized staff members can retrieve all tickets.

In the late stages of development, the ticket structure received a timestamp as to when the ticket was last updated, and an "update seen" boolean value. In addition, a logging system was implemented so that any errors and other important information is saved. Staff members (later named professors) and admins have different access, where professors have their access linked to their course and admins have access to everything. Students can only see their own tickets. The authority level based on numbers was changed to named roles.

3.2. API Design

The service will respond with a HTTP status code and either the requested content or a string explaining why the content couldn't be returned. Each call to the service must

include sufficient information for user authentication, which will restrict the returned content. The service's interaction endpoints are listed below, with the last column indicating returned HTTP status codes with short explanation of the response.

Table 1. API endpoints

Address	Method	Usage & Function	Output
/tickets	POST	Request must contain a JSON-object representing the added ticket. This ticket will be added to the database if user is authorized and the ticket fields have correct format.	200: Success: Contains the added ticket's identifier 401: Unauthorized 500: Internal error
/tickets /tickets/{ticket_id}	GET	If ticket_id is included, only one ticket matching that identifier is returned. Otherwise all available tickets are returned, based on user's authorization.	200: Success: No message, only tickets are returned 401: Unauthorized 404: Not found 500: Internal error
/tickets/{ticket_id}	PATCH	Request must contain a JSON-object representing a partial ticket, with only the modified fields set. These fields of the identified ticket will be updated if the user is authorized and the updated fields have correct format.	200: Success 401: Unauthorized ticket or fields 404: Not found 500: Internal error
	DELETE	Ticket matching given identifier will be deleted if the user is authorized.	200: Success 401: Unauthorized 404: Not found 500: Internal error
/categories	GET	Returns all created ticket categories.	200: Success 500: Internal error
	POST	Creates new category if user is authorized.	200: Success 400: Already exists 500: Internal error
/categories/{category}	DELETE	Deletes existing category.	200: Success 401: Unauthorized 404: Not found 500: Internal error

3.3. Interaction

The basic process chain design of this ticketing service is as follows: User navigates to Lovelace's ticketing provider. Lovelace detects that the user wishes to view their tickets and sends a request to the ticketing service, including the user's authorization token. See figure2 below for expected handling flow of this token. The service will query the database, listing all the tickets which this token gives access to, and sends this list back to Lovelace. As this list is received, Lovelace formats and otherwise adapts the data for the user, displaying it. The user can then select a ticket to manipulate. Lovelace will detect changes made by the user, and send them to the service, accompanied by the user's token. The service receives the changes and propagates them into the database, after which users will then have to request the data again to get updates.

The choice not to develop a user interface was influenced by the idea that Lovelace is more able to provide a good user experience that will fit its style and format. As Lovelace's theme is expected to evolve in the future, providing a separate interface would restrict this advance or worsen overall user experience. In addition, the maintenance required by this service should be minimized, as due to the nature of the project, continued updates will not be provided after initial deployment.

Prototyping of the API component began with Flask, which is a very lightweight API framework for Python. Flask was chosen due to its ease of use. Later in the design process it was switched for a more performant alternative, FastAPI, which focuses on speed while still maintaining usability. Uvicorn was used to host the prototypes.

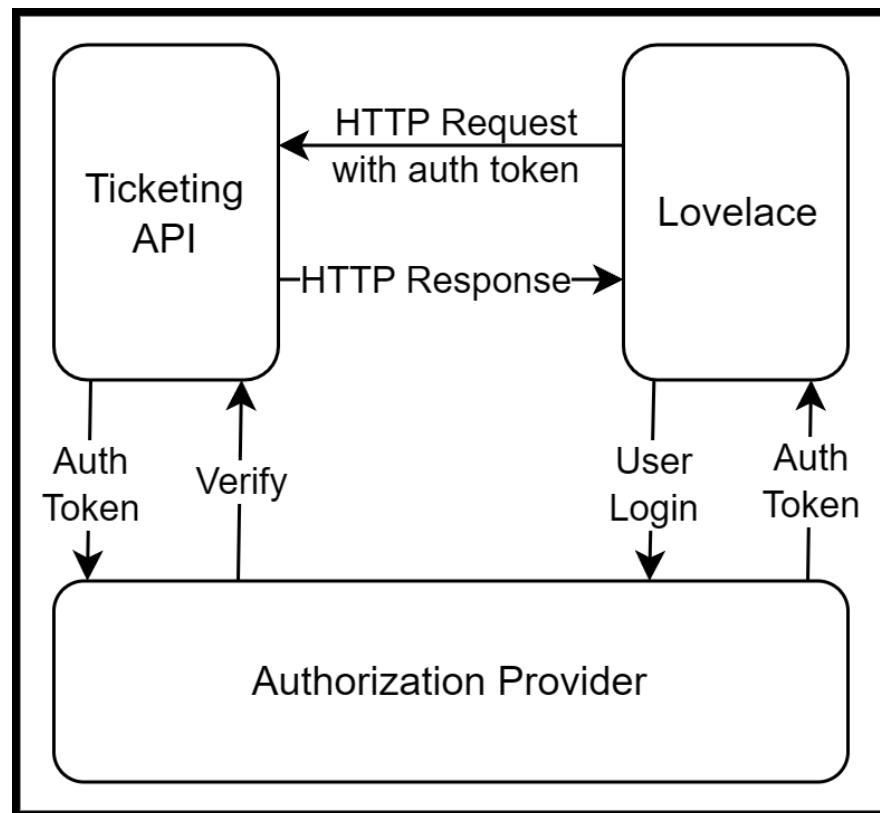


Figure 2. User authentication flow

3.4. Security Considerations

In our implementation, we have to take into account the biggest risks presented in the previous chapter. Our API design includes user authentication, user authorization, and prevention of SQL-injections. We will also have the option to support encrypted communication between the API and client(s), improving the confidentiality of the data. Mitigation of DoS and DDoS-attacks does not fall into the scope of our project.

Authentication is designed in a way that anyone who wishes to use the API must have a role assigned to them; student, professor or admin. Only these users can access and use the ticketing system. Authorization on our part is designed by limiting the student role. Students would have limited access only to their own tickets. In addition to own tickets, professors would have access to the tickets of their course and the possibility to add/delete categories. Naturally, people with the admin role would have unlimited access to all the tickets and methods. When it comes to the security of our database, we have decided to use prepared statements to prevent injection attacks[22]. This is fairly simple to implement, and using prepared statements drastically reduces the likelihood of an SQL-injection attack.

4. IMPLEMENTATION

4.1. Process

We started the implementation process by researching the necessary technologies, software, techniques and modules for the API, database, and their combined use. Especially MariaDB and Docker required extensive study due to the database system's large quantity of fairly complex features and use of SQL language, and the virtual environment manager's unique workflow.

After conducting shared research, workload was roughly split between writing the report and programming the API and database. Both of us participated in most major decisions relating to content and structure of both report and the program, and all work was focused on the report after programming concluded.

Focus in this project was on implementing the database and a simple API. Certain functionalities, such as user validation and deployment to Lovelace, will be implemented by a different team.

4.2. Technology Choices

We chose to implement asynchronous Python code for this project because it has many benefits compared to synchronous code in regards to the API design. The main advantages of asynchronous implementation in a web API are scalability, performance and responsiveness.

Using SQL instead of noSQL was the better choice for this project, due to the fact that the data received (tickets) is simply structured, does not require a lot of flexibility/scalability, yet benefits from good data consistency, which SQL provides. This was already decided during the designing of the software and it works well with our implementation.

4.3. Architecture

The basic concept of the architecture and logic can be seen in the following figure3. The figure shows how the service handles requests from Lovelace web clients. The rounded rectangles represent start/endpoints, regular rectangles represent operations that the API/database executes, where green ones represent the API and blue ones represent the database. Purple squares represent API authorization decisions and the purple diamond expresses the user validation step provided by a separate service within Lovelace.

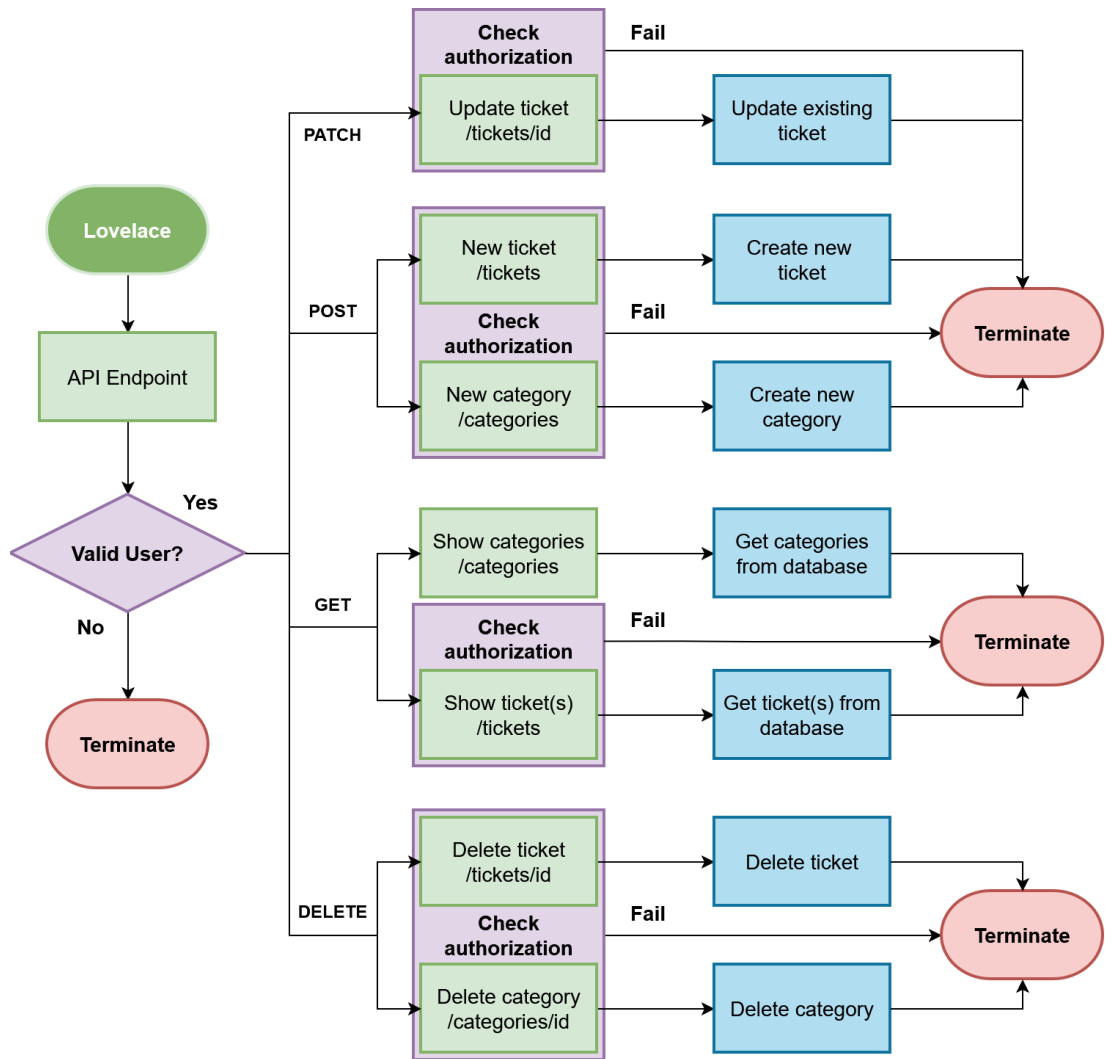


Figure 3. Ticketing Microservice process.

4.3.1. Data Structures

The tickets are sent to our API as JSON-objects. The API transforms the JSON into an instance of the Ticket class. The class holds all the information the received tickets have. Other crucial classes in the API are Authority and Update_Form. The Authority class represents the authority that a user has regarding the ticket database. It includes everything needed to check any user's access to a ticket. The Update_Form class is used to update the tickets. It contains the id of the updated ticket, as well as all fields that are being updated. Data is validated using pydantic, which is a data validation and type handling module for Python.

4.3.2. API Architecture

The service implements a simple database-api architecture, in which users contact the API with credentials, which validates the user with Lovelace authentication provider. Depending on the received authorization, users are granted access to a specific set of tickets in the database. This authentication happens with each request and session data is not saved, which is characteristic for a stateless REST API.

4.3.3. MariaDB Architecture

The ticket data in MariaDB is stored in a single table, as primitive types such as integer and string (specifically VARCHAR). In addition, ticket categories are stored in a separate table. This is a simple, but effective design for this project's use case. Ticket id functions as the primary key of the ticket table and it is not assignable by the user, as the database automatically assigns it whenever a ticket is added. Only when handling existing tickets may the user provide a ticket id, to identify the target of the update.

Ticket attributes are stored as integer, string, and boolean (which is ultimately stored as an integer with value of 0 or 1). Each attribute requires a value, empty fields are prevented. Ticket identifier and timestamps for creation and update dates are filled automatically by the database engine using the "AUTO_INCREMENT", "CURRENT_TIMESTAMP", and "ON UPDATE" features.

Field value lengths are restricted manually in the case of text, and by 32-bit integer maximum length for integers and timestamps. In the following table, VARCHAR(X) means text with limit of X characters. All INT fields are unsigned, so they can store two times larger positive values at the cost of negative values, with range of 0 to 4294967295. TIMESTAMP fields are stored as seconds from UNIX epoch, with maximum value of 2147483647, analogous to date 19.1.2038 at time 3:14:07.

Table 2. Database table: tickets

Field	Type	Parameters
ticket_id	INT	UNSIGNED AUTO_INCREMENT NOT NULL
user_id	INT	UNSIGNED NOT NULL
heading	VARCHAR(100)	NOT NULL
message	VARCHAR(500)	NOT NULL
category	VARCHAR(50)	NOT NULL
course	VARCHAR (50)	NOT NULL
state	VARCHAR(10)	NOT NULL
update_seen	BOOLEAN	NOT NULL
created	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP NOT NULL
updated	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP NOT NULL
assigned	INT	UNSIGNED NOT NULL

Table 3. Database table: categories

Field	Type	Parameters
category	VARCHAR(30)	NOT NULL

4.4. Security Implementation

Regarding the security aspect of our microservice, we have implemented user authentication, authorization, and database security based on the design outlined in the previous chapter. Implementation of these features did not pose any notable issues, and so far, they have worked as intended. Implementing HTTPS as an option in our API was also successful, as it now only requires for the host machine to provide secure socket layer (SSL) files to use it. Overall, we have implemented basic, adequate security measures for our microservice, but much of the security relies on the authentication implementation, which is not within the scope of this project.

4.4.1. Access Control

Our implementation uses double access control. First the access is verified through the API by contacting Lovelace authority provider, after which the access is also verified when storing/receiving data to/from the database. Implementing communication with the Lovelace authority provider is outside the scope of this project. Access to the database is limited by usernames and passwords, which will exist as a temporary text file that gets stored as a protected Docker secret upon launching the service container. These are sufficient for providing all 3 required levels of access to the database; admins have unrestricted access to all data, professors only have the options required by course teachers, while students are limited to accessing their own tickets.

4.4.2. Database Security

To mitigate SQL-injection attacks, we implemented the use of prepared statements for critical operations and basically for every operation where it is viable. Some cases (non-critical commands) required a hybrid approach using prepared statements with Python's f-string. Database security becomes noticeably better when implementing good user input sanitation, as in our case.

MariaDB supports a variety of options for data encryption and backups which can be implemented in accordance with wanted security requirements. At the moment, the database uses password protection, default encryption, and indexes to keep the data safe and consistent. Most likely danger comes from malicious access within the Docker container, in which case the attacker would already have admin-level access, nullifying almost all defenses. The likelihood of such an attack happening mostly depends on the host machine and the physical security of its location.

5. EVALUATION

5.1. Evaluation Plan

Evaluation of the service will be conducted using the client functionality of HTTPX[23]. As it is designed to work with FastApi services, it is able to test our implemented functionality extensively, including asynchronous operation. Some performance variables we can measure include response time to queries in various situations and the effect of asynchronous versus synchronous operation. In addition, we can inspect the security features applied to the implementation, trying to find weaker areas.

In addition to unit tests, curl[24] is used for quick testing functionality from the command console. Talend API Tester[25] was used for similar purposes, especially for its capability to save and visualize requests.

5.2. Functionality

The API receives HTTP-requests, which are then handled by the corresponding decorators, telling the application what to do. For example, during a HTTP POST request to the “/tickets/new”-URL triggers the “post_ticket” function, which in turn triggers a command for the database to create a new ticket. For every HTTP action, such as adding, modifying, or deleting tickets, the application responds with a HTTP status code and an explanation string, providing basic information of whether the action was successful or not. There were no issues aside from a few code mistakes noticed during testing, which were simple to fix. Testing was concluded on every supported method, with both correct and incorrect inputs.

5.3. Performance

Early performance evaluation targeted a locally hosted API, using the FastApi client. The basic methods of the API were tested using pytest, with a tailored script measuring response times and response correctness. The tests showed promising results: The tasks did their purpose in under 30 milliseconds consistently, except for one anomaly where sending a ticket took almost 400 milliseconds.

Later stages have the API hosted in a Docker container, with the test script sending requests from outside the container, as will be the case after deployment. After the API was updated to function asynchronously, a concurrent connection stress test was added. With 100 simultaneous sequences of post-, patch-, and delete-ticket requests, response time of POST increased from 40ms to around 400ms, which is still within acceptable bounds. Note that these numbers heavily depend on host hardware and amount of computation resources assigned to the Docker container, since testing on another machine with more high-end components, the response time was nearly halved.

A key factor in these fast speeds is the usage of asynchronous programming, which helps performance especially in tasks related to modifying the database. At the

moment, we haven't identified any major bottlenecks hindering the system, but there has not yet been performance testing under great network stress.

The tables below present a rough estimation for the service's response speed, tested on a lower end personal computer, which also hosted the service in a Docker container. First table4 results from running 100 simultaneous processes, each posting a similar ticket, modifying it with a patch call, then deleting it. The data describes the service's performance under unusually large load, leading to slower but still acceptable responsiveness. The second table5 shows results from 100 singular calls to each method, one method at a time. The calls no longer compete for computing time and these durations are a more accurate representation of the service's performance. All GET calls took from ~0ms to 10ms, with an average of 2ms.

Figure4 shows method response times for each of the 100 processes, sorted roughly in process launch order. It seems the first step, POST, serves as a bottleneck that helps distribute the load of later steps over time, so the first step's process time varies the most by process order. Competition for processing resources can be seen in some of the coinciding spikes in different step durations. Total process completion times can be seen in figure5, note that the duration range does not begin from zero.

Table 4. Test duration for 100 parallel sequences (in milliseconds)

Statistic	POST	PATCH	DELETE	Full Sequence
Minimum	153	280	209	1580
Average	498	750	645	1892
Maximum	826	1339	758	2170

Table 5. Test duration for 100 consecutive method calls (in milliseconds)

Method target	Statistic	POST	PATCH	DELETE
Tickets	Minimum	52	55	20
	Average	69	70	26
	Maximum	116	98	114
Categories	Minimum	12	-	16
	Average	26	-	27
	Maximum	61	-	78

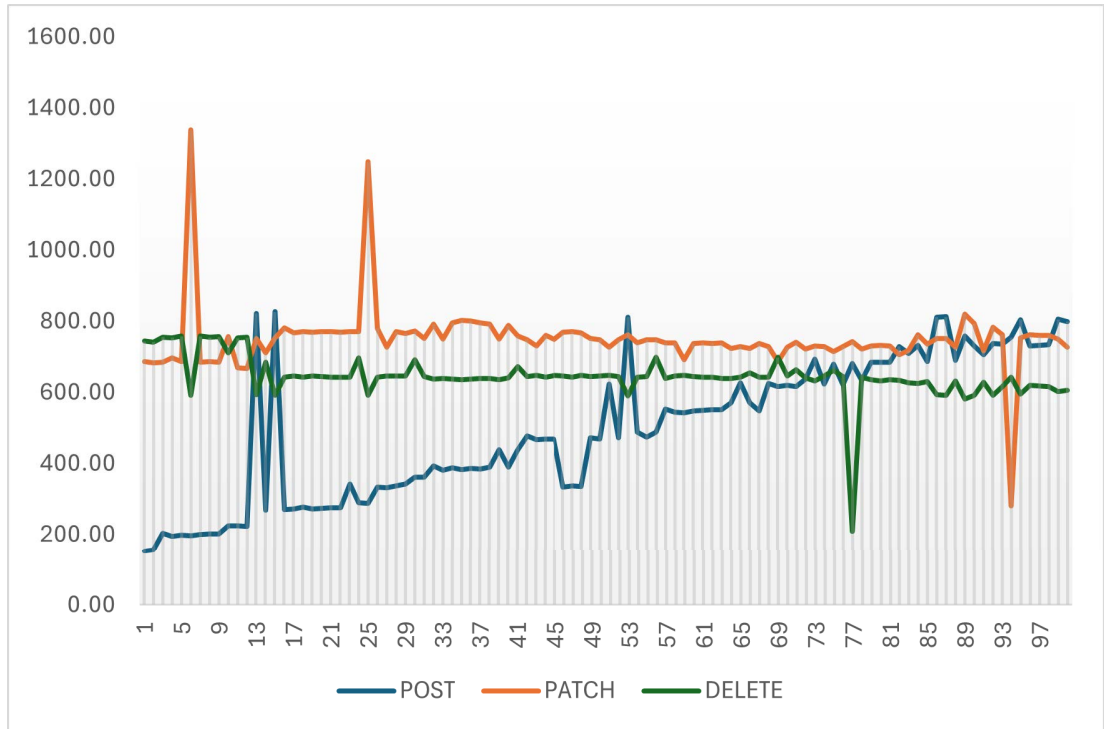


Figure 4. Parallel sequence step durations for each process.

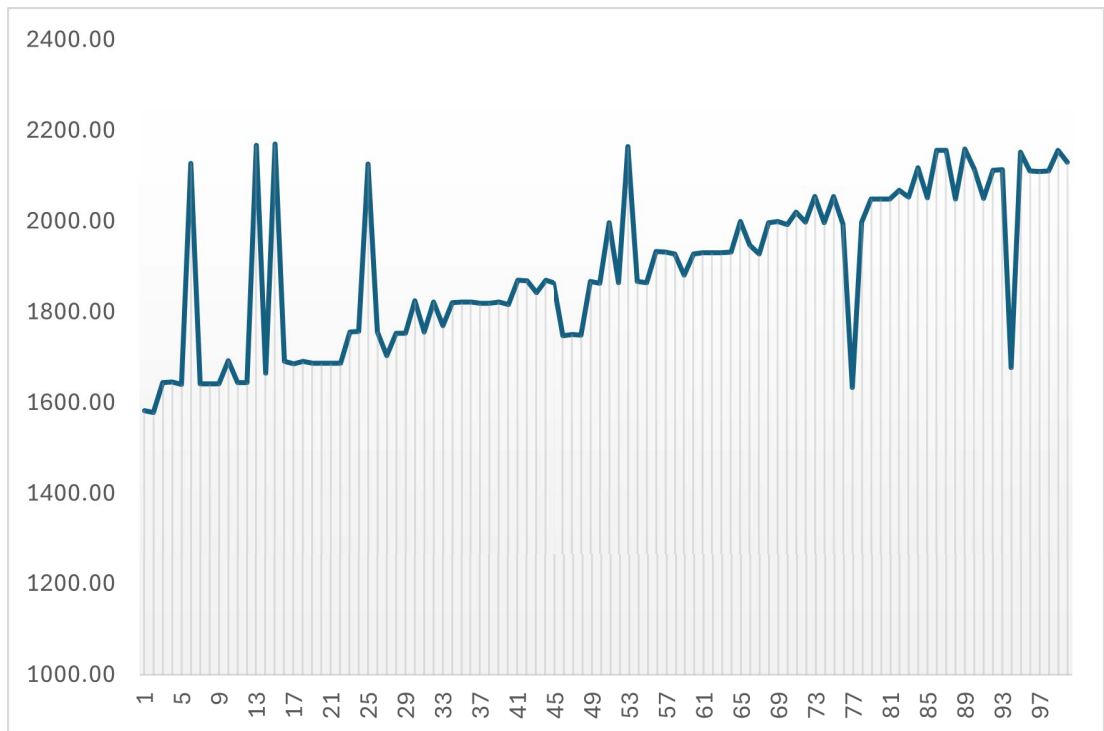


Figure 5. Parallel sequence total duration for each process.

5.4. Quality

Usually code quality is measured using performance, security, and maintainability as key factors. Our main goal regarding code quality is to make the ticketing service maintainable, limiting complexity and improving readability while still retaining all the functionality. The code in this project is extensively documented using comments and documentation strings to explain intended behavior and interaction parameters. The code is divided into files and segments based on functionality to reduce complexity and line count of single files. Python code reviewer tool Pylint[26] was used to analyze code quality.

5.5. Security

Security in the application is ensured by checking the authority of the user, by using prepared statements for the database and by encrypting the data in the database. The application uses tokens provided by Lovelace to check the authentication of the user, after which the API gives corresponding access level. Input sanitization is used to prevent any user from executing malicious code in the service. The API also supports HTTPS (encrypted HTTP) instead of HTTP to ensure adequate network security for the service. SQL-injection attacks are prevented using input sanitization and separation of SQL-statements from statement fields.

5.6. Agility

Good documentation and the modular nature of this microservice support simple and quick development cycles. Lightweight code, small number of files, and minimized external module usage make it easy to find specific code segments and get a full picture of the service, making it easy to start development without prior knowledge.

In the early stages of the project, rapid iteration was used for as long as possible, until we decided to start using Docker virtual environments. This step massively increased build times, slowing iteration. This can be avoided by developing inside the Docker container, but that has difficulties of its own and the negatives outweighed the positives in our situation.

6. DISCUSSION

6.1. Reflection

Overall, the project was a success. We managed to design and implement a ticketing microservice for Lovelace. The scope of the project was also fulfilled since it was not within our confines to actually have a final working version integrated into the Lovelace learning platform. This project is a solid framework that is open and ready for future work and live implementation.

By far, the biggest issue in this project was time management. Both of us had many other courses and projects to attend to, so there were a few times where we had to work for several hours through the night. We believe that both of us have gathered valuable experience in time management regarding large projects, albeit the hard way.

Designing and implementing a modern microservice was a very useful learning experience. Microservices and APIs were quite a new field for both of us and it took us many hours to familiarize ourselves with all the new information. We learned about basic web-API design, database design and usage, some important security features and the core philosophies behind microservice design.

Another important aspect of this project was to learn collaborative work on a project and delegating tasks. Neither of us had experience leading similar projects, nor a particular drive to become a leader, but in the end we were relatively successful in dividing labor and getting work done, though our overall communication could definitely use some improvement.

6.2. Future Work

Future work of the project will probably be conducted by other people who will take on the task of deploying the ticketing microservice to Lovelace. The code is modular, quite self-explanatory, and well documented with the goal of making the project easy to work with. Docker implementation makes developing and deploying the project flexible, regardless of operating system or development environment.

Although the project code is relatively easy to work with, real-world deployment is a bit more complex if not using Docker. First of all, a process manager should be used to ensure resiliency when running multiple processes. In Uvicorn's case this is done by using the built-in "workers" option, Gunicorn[27], which is a separate worker model for running web frameworks in production, or supervisor[28], which is a client/server system that aims for a convenient and simple way to run and control multiple processes. In addition, it is recommended to run the Uvicorn processes behind a proxy, such as nginx[29], the most popular web server in the world. Using a proxy server provides additional robustness when running the service in production. Requests and responses between the client and the server are sent through the proxy, providing a controlled environment for communication[30].

In a way, we ended up doing some future work ourselves, since this project was on a considerable hiatus. During that time, some of the packages used became obsolete, and updating the code taught us a bit about maintainability and versioning of used libraries. Due to the modularity of the microservice architecture and well documented code, it

was relatively easy to start working on the project again. It would be an exciting experience for both of us to see this microservice in real use in a future version of Lovelace.

7. CONCLUSION

The goal of this project was to design and implement a ticketing microservice for Lovelace and it was a success. In general, the ticketing system is a powerful tool for supporting the users of various systems, such as Lovelace. The ticketing system can be used by students to report issues and ask for help and it can also be used by staff to report bugs, errors etc. to other staff members and system administrators.

Pre-existing tools, guides, and documentation were a paramount part of this project and we could not have made the ticketing service happen without all the useful information, mostly in the form of documentation. While there was no “out of the box” template for this project, many APIs tend to be similar to what we implemented, so guidance on the concept was easy to find. Although APIs tend to be designed and developed for a very specific use case (such as this one), they do share a big amount of similar design choices.

During this project we learned a lot about designing and implementing microservices and got a concrete perspective on what happens “under the hood” when interacting with different kinds of microservices, which are commonly used nowadays. The project also taught us about different security features that have to be accounted for when designing online services.

8. REFERENCES

- [1] Algharaibeh S.A.S. (2020) Should i ask for help? the role of motivation and help-seeking in students' academic achievement: A path analysis model. *Cypriot Journal of Educational Sciences* 15, pp. 1128–1145. URL: <https://doi.org/10.18844/cjes.v15i5.5193>.
- [2] Official MariaDB website. URL: <https://mariadb.com/>.
- [3] Official FastAPI website. URL: <https://fastapi.tiangolo.com/>.
- [4] Official Uvicorn documentation (n. d.), [introduction to uvicorn]. URL: <https://uvicorn.dev/>, (Retrieved 13.12.2025).
- [5] Wongsakthawom R. & Limpiyakorn Y. (2018) Development of it helpdesk with microservices. In: 2018 8th International Conference on Electronics Information and Emergency Communication (ICEIEC), pp. 31–34. URL: <https://ieeexplore.ieee.org/document/8473557>.
- [6] Newman S. (2014) *Building microservices: Designing fine-grained systems*. (1st Edition). O'Reilly Media Inc.
- [7] Lewis J. & Fowler M. (2014), *Microservices: A definition of this new architectural term*. URL: <https://martinfowler.com/articles/microservices.html>.
- [8] Geeks For Geeks, Difference between relational database and nosql. URL: <https://www.geeksforgeeks.org/dbms/difference-between-relational-database-and-nosql/>.
- [9] Goodwin M. (n. d.), What is an api (application programming interface)? URL: <https://www.ibm.com/think/topics/api>, (Retrieved 13.12.2025).
- [10] Fielding R.T. (2000) *Architectural Styles and the Design of Network-based Software Architectures*. Ph. d. dissertation, University of California, Irvine. URL: <https://roy.gbiv.com/pubs/dissertation/top.htm>.
- [11] ASGI Team (2018), Introduction to asgi. URL: <https://asgi.readthedocs.io/en/latest/introduction.html>, (Retrieved 13.12.2025).
- [12] Mozilla for Developers, Server-side web frameworks. URL: https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side/First_steps/Web_frameworks.
- [13] Django Software Foundation (n. d.), About django. URL: <https://www.djangoproject.com/start/overview/>.
- [14] Official Flask website. URL: <https://flask.palletsprojects.com/en/stable/>.

- [15] Rafiqul Hasan (2024), Fastapi deconstructed: Anatomy of a modern asgi framework. URL: <https://rafiqul.dev/posts/fastapi-deconstructed/>, (Retrieved 4.2.2026).
- [16] Suutari L. & Holappa J. (2022) Future proofing Lovelace system development environment. Bachelor's thesis, University of Oulu, Finland. URL: <https://oulurepo.oulu.fi/handle/10024/20277>.
- [17] MariaDB Foundation (n. d.), Encryption. URL: <https://mariadb.com/docs/server/security/securing-mariadb/encryption>, (Retrieved 12.12.2025).
- [18] MariaDB Foundation (2025), About mariadb maxscale. URL: <https://mariadb.com/docs/maxscale/maxscale-architecture/mariadb-maxscale-guide>, (Retrieved 12.12.2025).
- [19] MariaDB Foundation (2025), Backup and restore overview. URL: <https://mariadb.com/docs/maxscale/maxscale-architecture/mariadb-maxscale-guide>, (Retrieved 12.12.2025).
- [20] MariaDB Foundation (2025), Server monitoring & logs. URL: <https://mariadb.com/docs/server/server-management/server-monitoring-logs>, (Retrieved 13.12.2025).
- [21] Cloudflare (n. d.), What is api security? URL: <https://www.cloudflare.com/learning/security/api/what-is-api-security/>, (Retrieved 31.1.2026).
- [22] Geeks for geeks, Mitigation of sql injection attack using prepared statements (parameterized queries). URL: <https://www.geeksforgeeks.org/sql/mitigation-sql-injection-attack-using-prepared-statements-parameterized-queries/>.
- [23] HTTPX Async Support. URL: <https://www.python-httpx.org/async/>.
- [24] Official curl website. URL: <https://curl.se/>.
- [25] Talend API tester - Free Edition. URL: <https://chromewebstore.google.com/detail/talend-api-tester-free-ed/aejoelaoggembcagimdiliamlcdfm?pli=1>.
- [26] Official Pylint website. URL: <https://www.pylint.org/>.
- [27] Official Gunicorn website, Gunicorn. URL: <https://gunicorn.org/>.
- [28] Official Supervisor website, Supervisor. URL: <https://supervisord.org/>.
- [29] nginx official website, nginx. URL: <https://nginx.org/en/>.
- [30] Official Uvicorn documentation, Uvicorn deployment. URL: <https://uvicorn.dev/deployment/>.

9. APPENDICES

9.1. Project Time

Table 6. Timetable (Final stage represents work done after the course concluded.)

Student	Hours	Contribution
	Course stage 1	
Mikael Marin	109	Research, Writing thesis
Teemu Puro	94	Research, Writing thesis
	Course stage 2	
Mikael Marin	52	Mostly coding, some research and thesis writing
Teemu Puro	50	Research, coding and thesis writing
	Course stage 3	
Mikael Marin	50	Mostly coding, some research and thesis writing
Teemu Puro	40	Testing, evaluation and thesis writing
	Course stage 4	
Mikael Marin	7	Thesis writing, Making the presentation
Teemu Puro	27	Thesis writing, Making the presentation
	Final stage	
Mikael Marin	60	Coding (40h), thesis writing (20h)
Teemu Puro	36	Thesis writing, research, checking citations, revisions