BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION COMPUTER SCIENCE

DIPLOMA THESIS

Designing an Emergency Management System

Supervisors

Asist. Drd. Bogdan-Eduard-Mădălin Mursa
Prof. univ. dr. Anca Andreica

Author
Marco Munteanu

2024

UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ

LUCRARE DE LICENŢĂ

Proiectarea unui sistem de gestionare a urgențelor

Conducători științifici

Asist. Drd. Bogdan-Eduard-Mădălin Mursa
Prof. univ. dr. Anca Andreica

Absolvent
Marco Munteanu

2024

ABSTRACT

This thesis focuses on the design and development of the emergency response application named Sky Sentinel. The main objective of the system is to provide a useful and easy-to-use emergency management tool. The system's particular design is meant to benefit the general public and emergency responders.

Chapter 1 is a thorough description of the application and an introduction to the idea and significance of advanced emergency management systems.

In chapter 2 the capabilities and limitations of the emergency management systems in use today are reviewed. By highlighting particular requirements that are essential for an emergency response application, these informations establish the goal for the system.

Chapter 3 presents an examination of the prerequisites required for the application. Finding the functional and non-functional requirements that the system has to satisfy is the first step in the process. This methodology guarantees the early identification of all crucial functionality in order to plan the design and execution stages of the development process accordingly.

The architectural choices taken for the system are examined in Chapter 4, with an emphasis on introducing the microservices architecture. This chapter also shows how software development architecture has changed over time, moving from standalone programs to microservice-based platforms. This presents why modern software solutions in general and emergency response systems in particular benefit from the microservices design.

The system's chosen architecture and design are presented in Chapter 5, which also includes a description of the frontend and backend components intercommunication. The technology stack that will be employed is also shown.

The actual implementation of the previously discussed design and the main features are presented in the Chapter 6. This chapter covers the details of implementing the backend microservices, the user interface and integration with external APIs. It also covers the testing and validation of the implemented system.

Chapter 7 is the final chapter and summarizes the results of this thesis. It considers how this system can help emergency management, talks about potential future enhancements to increase functionality and boost efficiency.

In summary, this work demonstrates how the integration of advanced mapping, communications, and land cover analysis into a single platform can improve the efficiency of emergency management, supporting rapid response and strategic planning.

Contents

Chapter 1

Introduction

1.1 Overview of emergency management

No nation, community, or individual is exempt from the effects of calamities. However, disasters may and have been anticipated, responded to, and recovered from, and their effects have been lessened to a greater extent. The history of emergency management is longstanding. Early hieroglyphics represent cave dwellers trying to deal with calamities. The Bible mentions the numerous catastrophes that struck civilizations. Indeed, the narrative of Noah alerting his neighbors to the approaching flood and then building an ark to protect the planet's biodiversity could be seen as a preemptive lesson in risk management. People have looked for solutions to disasters for as long as there have been them.

Organized attempts at disaster recovery did not occur until much later in modern history. For instance, in the 18th century, the Great Fire of London in 1666 led to the development of fire insurance and building codes to prevent similar catastrophes. Moving forward, the 19th century saw the establishment of the American Red Cross in 1881 by Clara Barton, which played a significant role in providing disaster relief. In the 20th century, the 1906 San Francisco earthquake prompted the creation of more structured disaster response efforts, including improvements in building standards and emergency response protocols. The formation of the Federal Emergency Management Agency (FEMA) in 1979 marked a significant step in organizing and federalizing emergency management in the United States [1].

More recently, technological advances have further revolutionized emergency management. The use of satellite imagery and geographic information systems during Hurricane Katrina in 2005 improved situational awareness and response coordination. More recently, during the COVID-19 pandemic, digital tools and data analytics have been essential for tracking the virus, distributing vaccines, and managing medical resources.

Because emergency management concepts have been used differently over time, it is necessary to understand the history and evolution of emergency management. We can create more efficient plans and solutions to deal with future disasters by analyzing the achievements and shortcomings of these earlier attempts. Building resilience and making sure that communities are better equipped to handle the challenges presented by catastrophes depend heavily on this continuous process of learning and adjusting.

1.2 Role of technology in emergency management

In addition to using technology to assist many elements of our life, such as communication, transportation, food, healthcare, we can also efficiently use it to manage hazards and disasters. All aspects of disaster management starting from prevention, reaction, recovery, and mitigation, rely heavily on technology.

When disasters strike, technology becomes indispensable in the response phase. Real-time communication platforms, GPS, and drones are used to coordinate rescue efforts, deliver supplies, and assess damage. Social media and mobile apps can disseminate information quickly to the public, ensuring people know where to go for safety or how to get help.

During the recovery phase, technology assists in rebuilding and restoring affected areas. Geographic Information Systems (GIS) help map damage and plan reconstruction efforts. Data management systems track aid distribution and resource allocation, ensuring that help reaches those in need efficiently[2].

Even while food, shelter, water, and closeness to loved ones are essential resources during disasters, technology is still transforming disaster management readiness and efficiency.

For technology to be effective in emergency management, it needs to be user-friendly and accessible to everyone involved in the process, not just experts. This includes first responders, government officials, and even the general public. Apps with intuitive interfaces, automated alerts, and easy-to-understand data visualizations ensure that all users can quickly and effectively use the technology during a crisis.

1.3 Short Overview of the Application

The goal of the application, named Sky Sentinel, is to provide a user-friendly platform for managing emergency events. This platform is designed to make emergency management easier and more efficient by offering several important features:

Firstly, Sky Sentinel offers advanced mapping functionalities that allow users to visualize emergency situations in real-time. This includes the integration of geographic information systems to provide detailed and interactive maps, helping users understand the geographical context and extent of an emergency. The real-time maps enable quick assessment of the situation, allowing responders to make informed decisions and coordinate their efforts effectively.

Sky Sentinel includes robust communication tools to facilitate seamless interaction between different users. This feature supports real-time chat, alerts, and notifications, enabling first responders, administrative staff, and the public to stay informed and coordinated during an emergency. The communication system is designed to handle high volumes of traffic, ensuring that critical messages are delivered promptly even under heavy loads. This ensures that everyone involved in the emergency response is on the same page and can act quickly based on the latest information.

The application incorporates landcover analysis through the use of AI and satellite imagery. By analyzing land cover, Sky Sentinel can provide valuable insights into the terrain and environment, which are crucial for planning and executing emergency responses.

Additionally, the platform features also an AI assistant that is equipped to guide users, respond to any requests, and possesses comprehensive knowledge of the system to provide assistance.

1.4 Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work, the author used GPT-4 model in order to ensure the text was as clear and coherent as possible. This process involved initially analyzing the original content, followed by generating improved phrasings to enhance readability and flow. After using this tool, the author reviewed and edited the content as needed and takes full responsobility for the content of the thesis.

Chapter 2

Review of Existing Emergency Management Systems

2.1 Analysis of existing emergency management systems

Emergency management systems are critical frameworks designed to facilitate the handling of emergencies and disasters. These systems integrate various technologies and methodologies to ensure rapid and effective responses, minimizing the impact of disasters on human life and property. Understanding the functionality and structure of existing systems provides essential insights for the development of new solutions. This review aims to assess the performance of different systems in real-world scenarios, focusing on their technological innovations, user interface designs, and operational strategies.

2.1.1 D4H

D4H Technologies provides a sophisticated emergency management platform known for its comprehensive approach to incident preparation, response, and recovery. D4H's Incident Management system is particularly notable for its ability to simplify the management of resources, personnel, and information during emergencies, making it an exemplary model in the industry of emergency management systems [3].

Features and Functionalities:

- **Real-Time Incident Tracking:** D4H enables real-time tracking of incidents, allowing emergency managers to capture critical details, assign tasks, and monitor ongoing activities. This feature ensures a comprehensive overview of all operations, helping to identify issues and coordinate resources effectively.

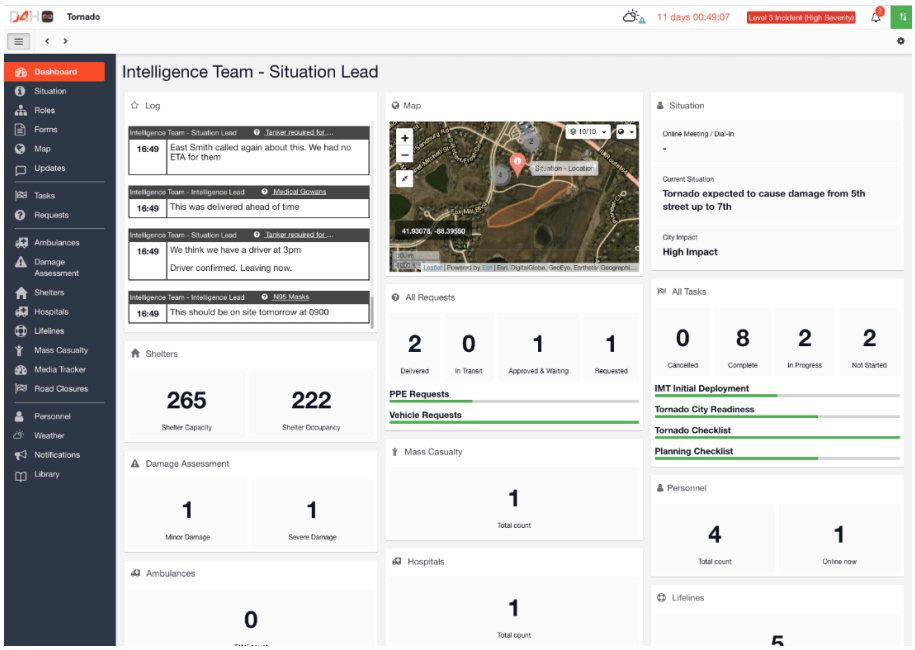


Figure 2.1: D4H dashboard [4]

- **Communication and Collaboration:** The platform includes built-in communication tools that support seamless information exchange among team members. These features are crucial for maintaining effective collaboration and ensuring that all participants are updated with the latest incident information.
- **Resource Management:** D4H offers robust resource management capabilities that help organize and deploy resources based on their availability and the needs of the incident. This system supports efficient decision-making to optimize response efforts.
- **Comprehensive Documentation:** The platform provides extensive documentation tools to record all details related to an incident, which is vital for post-incident analysis and compliance with regulatory requirements.

Technological Innovations: D4H incorporates advanced technologies that enhance its functionality, such as cloud-based data management for accessibility and scalability, and mobile integration that allows responders to access and input data on-the-go. These technological elements make D4H a state-of-the-art system in emergency management.

User Interface Design: The user interface of D4H is designed to be intuitive and user-friendly, ensuring that during high-stress situations, responders can quickly navigate and utilize the system without difficulties.

Impact on Emergency Management: The deployment of D4H has shown significant improvements in the efficiency and effectiveness of emergency management

operations. Agencies using D4H have reported quicker response times, better resource management, and enhanced coordination among different teams.

D4H exemplifies the potential impact of integrating comprehensive data handling and communication tools in emergency management systems.

2.1.2 Noggin

Noggin provides a robust emergency management platform that excels in offering integrated solutions for operational risk management, business continuity planning, and crisis management. It stands out for its holistic approach to managing emergencies and its ability to adapt to a wide range of needs [5].

Features and Functionalities:

- **Integrated Communications:** Noggin enhances communication channels including email, SMS, voice, and app notifications, which are vital for effective crisis management. It refines event communications to ensure all stakeholders are kept informed during emergencies.
- **Mapping and Visualization:** Utilizing advanced mapping functionalities, Noggin provides a visual understanding of assets and their relationship to events, which assists in resource allocation and impact assessment.
- **No-Code Customization:** The platform offers no-code customization options, allowing organizations to easily adjust the system to meet specific requirements without needing extensive technical skills.
- **Best Practices and Compliance:** Noggin comes equipped with a library of best-practice templates and supports compliance with international standards such as ISO 22301 and 22320, ensuring reliable and standardized emergency responses.

Technological Innovations: Noggin provides outstanding flexibility, enabling seamless adaptation to specific organizational requirements. The platform's intuitive no-code, drag-and-drop interface allows for precise customization, facilitating the use of tools such as dashboards, forms, templates, plans, charts, and reports to align closely with preferences and needs.

User Interface Design: The platform is designed to be accessible from any device, ensuring that users can operate effectively whether in the office, in the field, or on the go. This accessibility is crucial for maintaining operations during diverse emergency scenarios.

Impact on Emergency Management: Noggin has been instrumental in enhancing operational resilience across various industries. Organizations using Noggin

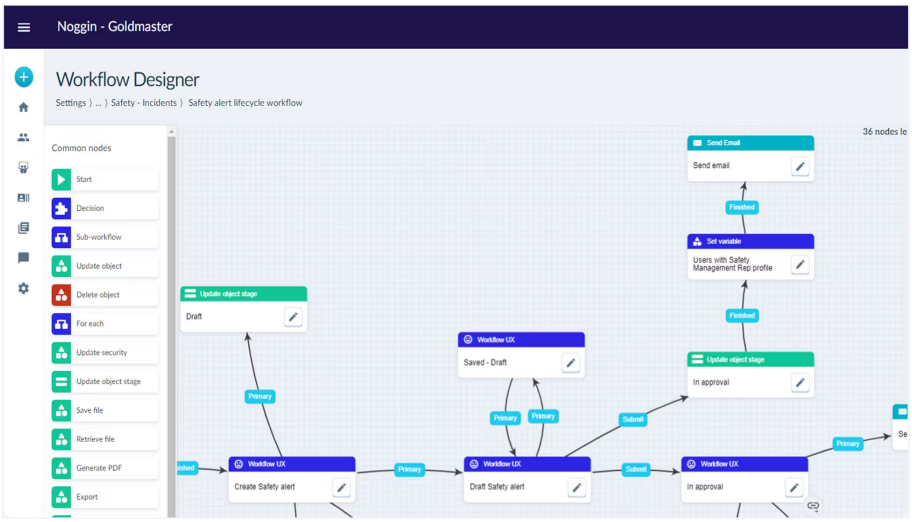


Figure 2.2: Noggin workflow designer [5]

report improved preparedness, quicker adaptation to crises, and enhanced overall resilience.

2.2 Conclusions

This chapter has explored the capabilities and functionalities of two prominent emergency management systems: D4H and Noggin. Below is a summary of the pros and cons of each system, presented in a comparative table format.

Feature	D4H	Noggin
Real-Time Incident Tracking	Offers comprehensive tracking capabilities that enhance situational awareness and resource coordination.	Provides integrated communication channels that facilitate swift information dissemination and response coordination.
User Interface	Intuitive and user-friendly, designed for rapid navigation during high-pressure situations.	Accessible from any device, designed for ease of use in diverse environments.
Customization	Supports efficient decision-making with robust resource management tools.	Features no-code customization, allowing precise adjustments to fit organizational needs.
Compliance	Extensive documentation features support compliance with regulatory requirements.	Supports international standards like ISO 22301 and 22320, ensuring best practices in emergency response.

Table 2.1: Comparative analysis of D4H and Noggin's features

Considerations and Limitations: While D4H and Noggin offer extensive features and capabilities, a common limitation is their complexity and the specialized training required to use them effectively. This can present challenges in environments where rapid deployment and user training are critical. Additionally, the sophistication of these platforms might overwhelm users who lack technical expertise, potentially limiting their wider adoption among the general public.

Future Enhancements and Innovations: Building upon the solid foundations laid by big systems such as D4H and Noggin, the proposed solution aims to extend their robust features while introducing innovative capabilities to better serve a wider audience. The implementation is designed not merely to fill gaps but to leverage and expand upon the proven strengths of these systems. Key innovations will include:

- **Greater Interoperability:** Expanding integration capabilities to include a more diverse range of third-party services and platforms, enhancing the system's utility and reach.
- **Road Segmentation Technology:** Introducing advanced road segmentation features that use AI to analyze and manage road-related emergencies, a unique addition tailored to the needs of everyday users.
- **Public Accessibility:** Crafting a user-friendly app designed not only for emergency responders but also for the general public, making advanced emergency management tools accessible to everyone.

Chapter 3

Requirements Analysis

The success of any software system heavily relies on the requirements analysis process. Traditionally, well-formed requirements are considered crucial at the inception of a project. However, many successful software projects have adopted a 'just-in-time' approach to requirements analysis, where requirements are captured less formally and elaborated primarily as implementation progresses [6]. This approach raises important considerations about the nature of requirements engineering and the tools and practices best suited for such environments. The analysis of the prerequisites for the Sky Sentinel emergency management system, covered in this chapter, acknowledges both traditional and non-traditional forms of requirements gathering. Understanding both functional and non-functional requirements is essential to ensure the system is efficient and meets expectations [7]. The provided analysis is crucial for guiding the subsequent phases of system design and development, ensuring all capabilities align with user needs. This flexibility in requirements engineering might be particularly beneficial in dynamically changing environments where decisions and their impacts need to be assessed continuously.

The goal of the Sky Sentinel system is to assist users in handling emergency situations. It functions similarly to a command center, gathering all the resources required to manage crises on a single, user-friendly platform. With the system's easy-to-use interface, anyone can report emergencies and get updates right away. You can easily navigate the system whether you're a concerned citizen or a first responder. The system makes sure that as soon as an emergency is reported, the appropriate persons are informed right away. This translates to improved coordination and quicker reaction times.

3.1 Functional Requirements

3.1.1 User Requirements:

- Users must be able to report new emergencies quickly and easily through a dedicated interface.
- Users should be able to view the status of ongoing and past emergency events.
- The system should allow users to volunteer for emergencies and join the real-time communication threads.

3.1.2 System Requirements:

- The system must authenticate users and manage user sessions securely.
- Real-time updates must be pushed to the users regarding emergency status changes.
- The system must store all emergency events, user actions, and communications securely and retrieve them efficiently as required.

3.2 Non-functional Requirements

3.2.1 Usability

The user interface (UI) of the Sky Sentinel system is a critical non-functional requirement, designed to ensure optimal usability during emergency situations. The UI must:

- **Facilitate Ease of Use:** Enable users to navigate the system efficiently, crucial during high-pressure situations where response time is critical.
- **Reduce Cognitive Load:** Incorporate intuitive layouts and visual cues that help minimize cognitive effort, allowing users to focus on the task at hand without distractions.
- **Ensure Accessibility:** Feature high-contrast visual elements and large, easily accessible buttons to accommodate all users, including those with limited technological proficiency.
- **Promote Quick Response:** Design the interface to facilitate a quick understanding and rapid input, reducing the likelihood of errors and enhancing the system's overall strength in emergency response.

3.2.2 Performance

- The system should be capable of handling up to 10,000 simultaneous user connections initially, with the ability to scale further based on demand without performance degradation.
- Response time for all user interactions should not exceed 2 seconds under normal operation conditions.

3.2.3 Reliability and Availability

- The system must be operational 24/7, with a targeted uptime of 99.9% outside of planned maintenance windows.
- Automatic failover mechanisms should be in place to ensure continuous service availability during system failures.

3.2.4 Security

- All user data must be encrypted in transit and at rest using industry-standard encryption protocols.
- Regular security checks and vulnerability assessments should be conducted to maintain system integrity.

3.2.5 Scalability

- The system should be designed to scale horizontally to sustain increasing loads and user numbers without requiring significant changes to the system architecture.
- Elastic scaling strategies should be employed to handle sudden spikes in demand, particularly during large-scale emergency situations.

3.2.6 Maintainability

- The system should be designed with modular components to facilitate easy updates and maintenance.
- Comprehensive logging and monitoring should be implemented to detect and resolve issues promptly.

These non-functional requirements are critical for ensuring the system not only meets its functional purposes but also adheres to performance, reliability, security, and scalability standards necessary for robust emergency management systems.

3.3 Constraints

- **Development Timeframe:** The system is required to be developed within a constrained timeframe by a single developer.
- **Budgetary Constraints:** The development and operational budget is limited, which may restrict the selection of tools like cloud-native technologies to enable autoscaling and improve resource efficiency.
- **Regulatory Constraints:** Adherence to data protection and privacy laws requires careful handling of user data, influencing system design and functionality.

3.4 Validation Criteria

The system will go through a number of organized testing to make sure it complies with all requirements:

3.4.1 Functional Requirements Validation

- **Automated Unit Tests:** Every module will undergo independent testing to ensure that its functionality corresponds with the anticipated results.
- **End-to-End Tests:** This type of testing will mimic user interactions from beginning to end, guaranteeing that every component of the system functions as intended.

3.4.2 Non-functional Requirements Validation

- **Performance Testing:** To make sure the system functions well under stress, tests will be conducted on its response times and data handling capabilities under various loads.

The purpose of these tests is to thoroughly evaluate the system as a whole as well as its individual parts, making sure that every feature satisfies design requirements and operates at peak efficiency in all scenarios.

Chapter 4

Microservices Architecture in Web Development

4.1 Evolution of architectural patterns in software development

In the constantly changing world of software development, the amount and complexity of software systems is on the rise [8]. The concept of architectural pattern plays a crucial role in the structuring and organization of these systems. They provide a way for representing, sharing and reutilizing knowledge that is crucial for building and maintaining robust software architectures.

4.1.1 Standalone Architecture

In the early stages of software development, the standalone or single-tier architecture was the main pattern of application design. This architectural pattern is characterized by the entire application, starting from the user interface to data processing and data storage, being placed within a single platform or device [9].

Standalone applications are self-contained units that operate independently of other systems. They do not require network connectivity for functionality.

The primary advantage of standalone applications lies in their simplicity and reliability. With all resources locally available, these applications typically have quick response times. Furthermore, they offer privacy and security advantage since the data remains tied to the user's device.

Their limitations are obvious in the context of today's world. Scalability is a significant challenge; as standalone applications cannot easily distribute processing or storage tasks across multiple servers. This limitation affects both the performance of the application under heavy loads and its ability to handle complex operations.

As the internet grew, people needed more ways to work together and share information. There was the need for systems that could connect and communicate over the internet. This shift led to the development of client-server architecture. By separating the data layer from the client interface, this allowed multiple clients to interact with the server, fixing many of the limitations of the single-tier applications.

4.1.2 Client-Server Architecture

The move to networked computing significantly changed how we design software, moving from self-contained, single-use applications to the more interactive client-server model. This change introduced a system where user-facing parts of applications, known as clients, ask for services or data, and the back-end parts, known as servers, provide them. This setup allowed for more users to access the same data or services at the same time, making it essential for creating websites and business software that many people could use together.

Clients are the parts you interact with, like the app on your phone or the web browser on your computer. Servers are powerful computers that store data and applications, responding to requests from clients. This setup has several benefits.

It's scalable, meaning it can handle more and more users by adding more servers. It allows for centralized updates and management, making it easier to keep software up-to-date and secure. Servers can be optimized to handle specific tasks efficiently, which is great for managing data and processing requests from many users at once.

However, there are also challenges. The system relies heavily on network connectivity, which can introduce delays or outages. Centralizing data on servers can create security risks, requiring strong protections against hacking and data breaches. As the system grows, it can become complex to manage, requiring careful planning to ensure everything runs smoothly. The client-server architecture laid the groundwork for the internet and many applications we use daily, and it's a key part of how cloud computing works today [10].

4.1.3 The Rise of Microservices

A microservice is a self-contained unit of limited extent that can be deployed independently and facilitates interaction via message-based communication. Microservice architecture refers to a method of building highly automated, adaptable software systems composed of microservices that are aligned with specific capabilities. The rise of microservices represents a significant shift in software architecture, moving away from large, monolithic applications towards smaller, independently deployable services. This trend reflects a broader evolution in software development towards systems that are more modular, scalable, and capable of rapid iteration [11].

Microservices architecture offers a solution to the challenges posed by traditional monolithic systems, particularly in terms of maintainability and scalability. By breaking down applications into smaller components that perform distinct functions, developers can update, deploy, and scale these services independently, facilitating continuous delivery and improving system resilience.

4.2 Advantages of microservices architecture

The adoption of microservices has been driven by the need to manage increasingly complex software systems more efficiently. As applications grow, making changes or adding features to a monolithic system can become cumbersome and risky, slowing down development cycles.

Microservices, by contrast, allow teams to work on different services concurrently, reducing dependencies and potential conflicts. This architectural style not only accelerates development but also enables organizations to leverage cloud computing more effectively, taking advantage of on-demand resource scaling.

Netflix and Amazon are often cited as pioneers in adopting microservices, transitioning from large, unified codebases to architectures where services communicate through well-defined APIs. This approach has enabled them to scale their operations dramatically, supporting millions of users worldwide. The success of these companies has underscored the viability of microservices for building complex, high-traffic web applications [12].

The technical foundation of microservices includes a range of tools and practices designed to facilitate service deployment and management. Lightweight containerization technologies, such as Docker, play a crucial role in encapsulating services and their dependencies, making them easier to deploy across different environments.

4.3 Challenges and considerations in adopting microservices

One of the significant challenges in adopting microservices is managing the increased complexity of operations. Microservices architecture distributes functionality across multiple smaller services, which can complicate deployment, monitoring, and maintenance processes. The costs associated with these operations can escalate if not managed effectively, requiring investments in sophisticated tooling and skilled team members to handle the distributed nature of the system.

Deciding how to appropriately decompose an application into microservices is non-trivial and involves understanding the domain deeply enough to identify bounded contexts correctly. Data management becomes particularly challenging in a microservice architecture due to the distributed nature of services. Ensuring data consistency and integrity across services without creating tight coupling can be difficult.

Microservices are designed to be independently deployable and scalable, which allows for rapid iteration and evolution of individual components without impacting others. However, this continuous evolution can lead to challenges in maintaining system coherence and ensuring that all services are aligned with the current business needs and technology standards [12].

Chapter 5

System Design and Architecture

5.1 Proposed system overview

The primary goal is to provide a robust and user-friendly interface that allows various users, ranging from the general public to first responders and administrative staff to manage and respond to emergencies effectively.

The system will be structured around a microservices architecture, which is important in building robust and scalable applications that can support high loads and maintain resilience. This architecture style breaks down the application into smaller, independent services that perform specific functions. By using microservices, the system ensures that components such as the Engine Service, Landcover Analysis Service, and Communication Service can be developed, deployed, and scaled independently.

This modularity allows for better fault isolation, as issues in one service can be addressed without impacting the overall system's availability. Moreover, microservices facilitate rapid development cycles and continuous deployment, which is crucial for adapting to the ever-changing requirements of emergency management. The use of this architecture supports the system's ability to handle extensive data and user load, maintaining performance and uptime during critical operations.

5.2 Detailed architecture design

In this section, the architecture of the system is presented, which is a microservices architecture to ensure scalability, flexibility, and resilience. Each component of the system is designed to operate independently and communicates effectively through well-defined interfaces and protocols. The architecture diagram, shown below, illustrates how these components interact within the system.

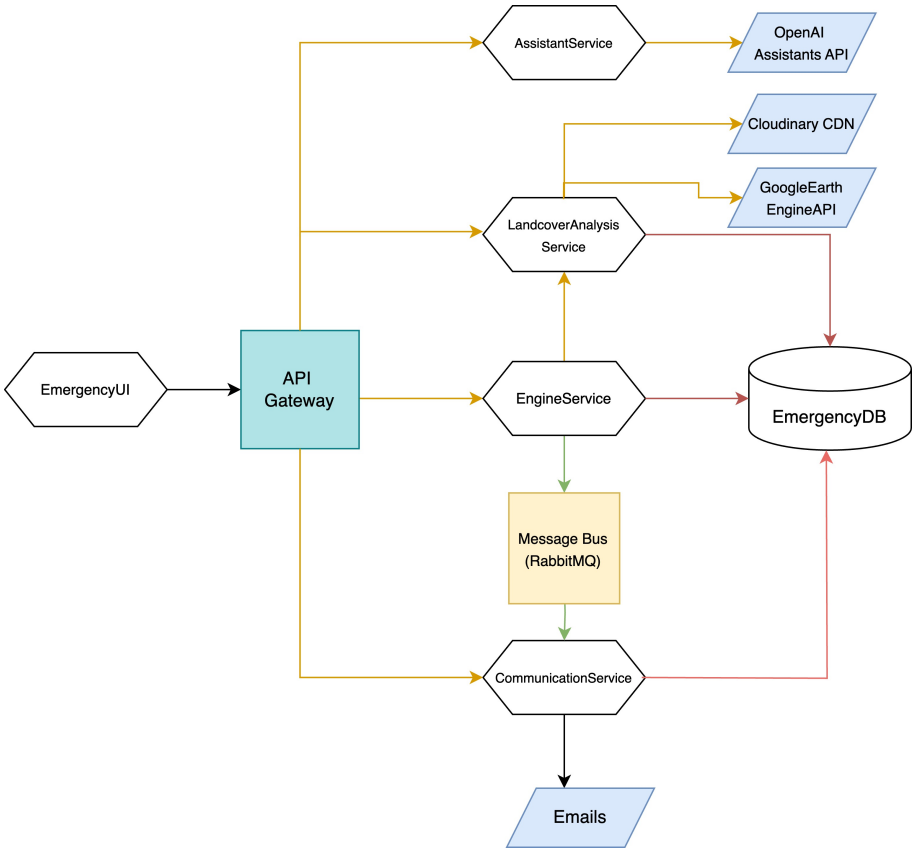


Figure 5.1: System Architecture Diagram

5.2.1 Backend components

- **API Gateway**: The API Gateway is the entry point for all requests targeting one or more of the backend services. This architectural choice enhances system scalability by decoupling the clients from the backend implementation. With this extra layer of abstraction, interactions between the clients and the services are simplified and made more straightforward.

- **Engine Service**: Acts as the backbone of the backend, handling core functionalities such as user management, event registration, and resource management. It directly interacts with the Emergency DB to perform data retrieval, updates, and storage operations. The Engine Service also plays a crucial role in facilitating communication within the Emergency Response System. It interacts with the Communication Service through a Message Bus, based on RabbitMQ, to ensure seamless and efficient message exchange across different components of the system. This interaction primarily supports the synchronization of user actions and system responses, enabling real-time updates to be communicated promptly to the relevant users. In addition, this service is also linked with the *Landcover Analysis Service* for incorporating AI-driven image processing into the emergency management workflow.

- **Landcover Analysis Service**: This service handles the processing of satellite imagery, delivering essential land cover analysis for emergency management. It retrieves images through the Google Earth Engine API and utilizes semantic segmentation models to examine the terrain and extract the roads, vital for effective emergency response strategies.

- **Communication Service**: Manages all communication channels within the system, particularly focusing on real-time interactions during emergencies. It supports features like real-time chat, notifications, and alerts, which are vital for effective communication during crisis management. This service is designed to receive events from the Message Bus. Upon receiving these events, which could include updates on ongoing emergencies, changes in resource allocations, or new emergency alerts, the *Communication Service* reacts by sending out targeted notifications and emails to relevant users.

- **Assistant Service**: This service abstracts away the communication with the OpenAI assistants API, providing specialized assistants that possess specific knowledge about the application. It acts as an intermediary, allowing users to interact with AI-driven assistants that can offer expert advice, automated responses, and guidance tailored to various aspects of the emergency management system.

5.2.2 Frontend: User interface and interaction

The frontend of the system is designed with a focus on user interaction and responsiveness, particularly under the demanding conditions of emergency management. The user interface is engineered to provide a seamless and intuitive experience, enabling users to navigate the system efficiently during high-stress situations.

The *Emergency UI*, the system's frontend component, is structured to facilitate quick access to all necessary functionalities with minimal interaction. This ensures that users can report emergencies, access resources, and communicate with minimal delays. The design prioritizes clarity and accessibility, incorporating features like a dashboard where users can visualize active emergencies on an interactive map. This map-based interface is not only intuitive but also provides immediate situational awareness, which is crucial during an emergency.

The frontend is closely integrated with the backend services through RESTful APIs and Web Sockets , which allows for real-time updates to the user interface. Changes in emergency status, resource availability, and other critical alerts are pushed to the user's interface instantaneously. This capability ensures that all users are informed promptly and can react quickly to evolving situations.

The goal is to minimize the complexity of interactions, therefore reducing the cognitive load on users during emergencies, which is crucial for fast decision-making and effective response.

5.2.3 Database design and schema

The database of the system is built on PostgreSQL, a robust and scalable relational database management system that supports complex queries and extensive data integrity features. This choice reflects the system's need for reliable transaction processing and efficient data retrieval capabilities, which are crucial for the responsive and dynamic nature of emergency management applications.

The database is accessed through an Object-Relational Mapping (ORM) layer by the EngineService, which simplifies the interaction with the database by abstracting SQL commands into object-oriented code. This approach not only enhances the development speed but also maintains the code's clarity and reduces the likelihood of SQL injection attacks, ensuring the application's security.

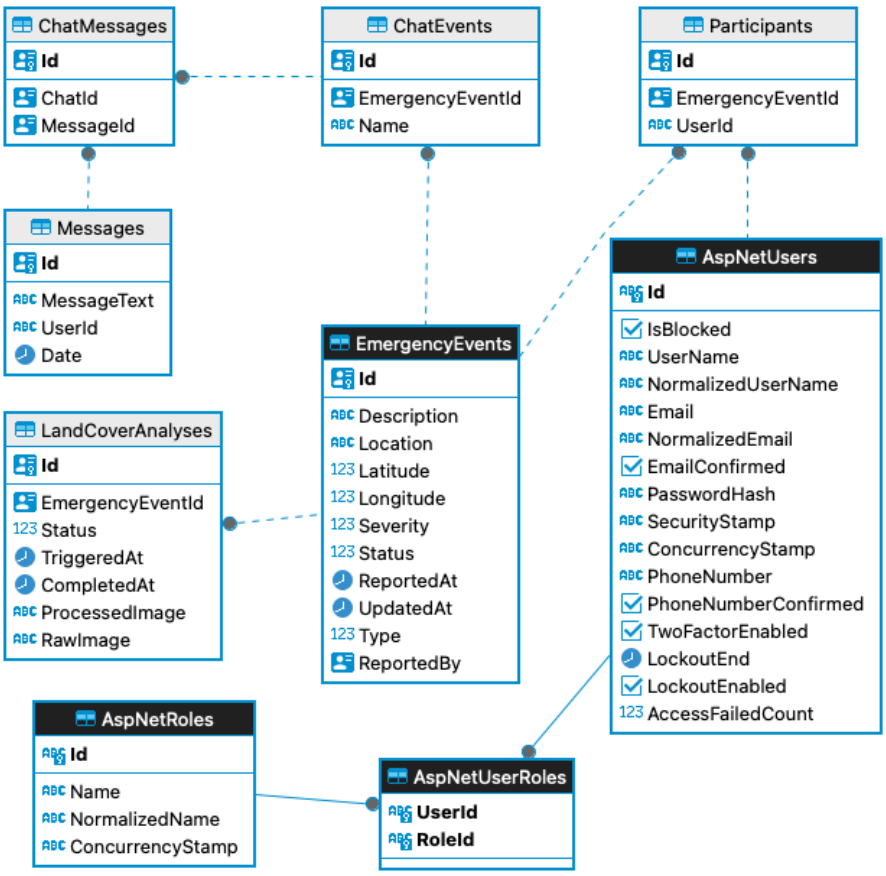


Figure 5.2: Data Model

5.3 Technology Stack Overview

5.3.1 Angular

Angular is a versatile and powerful platform for building dynamic single-page applications (SPAs). Developed and maintained by Google, Angular leverages TypeScript to enhance development productivity through static typing, classes, and interface features.

After being entirely redesigned and relaunched as Angular 2 (later simply Angular) in 2016, Angular was first released as AngularJS in 2010. This change signified a substantial departure from JavaScript to TypeScript, improving its performance, maintainability, and scalability. The purpose of the modification was to solve the problems with AngularJS that large-scale applications encountered, especially with regard to testability and performance [13].

In this project, Angular is the the primary interface through which users interact with our system. It offers a well-designed user interface that works smoothly alongside the backend services. This integration ensures a seamless user experience, enabling efficient and dynamic interactions with the application.

5.3.2 ASP.NET Core

ASP.NET Core is a modern, high-performance, open-source framework developed by Microsoft for building a wide range of applications, including web applications and services, IoT apps, and mobile backends. It is a cross-platform tool, meaning it can run on Windows, Linux, and macOS, which provides a flexible and accessible environment for developers. ASP.NET Core was first released in 2016 and is a redesign of earlier Windows-only versions of ASP.NET [14].

Statistics and performance benchmarks often highlight ASP.NET Core as one of the fastest web frameworks available. In industry comparisons, such as those conducted by TechEmpower, ASP.NET Core consistently ranks at the top for raw throughput and response times across various types of web applications [15].

In Sky Sentinel, ASP.NET Core is used to implement the Engine Service and the Communication Service, which make up the core of the system.

5.3.3 Python

Python is known for its simplicity and readability, which significantly accelerates the development process and has established it as a favorite among programmers for tasks ranging from simple scripting to complex system development.

It was created by Guido van Rossum and first released in 1991. The name "Python" doesn't come from the snake but rather from the British comedy series "Monty Python's Flying Circus," a show van Rossum enjoyed during the development of the language.

Python is designed around a philosophy that emphasizes code readability and simplicity, often referred to as "The Zen of Python." This set of aphorisms includes principles like "Beautiful is better than ugly," "Explicit is better than implicit," and "Simple is better than complex," guiding Python developers towards writing clear and logical code [16].

In the current project, Python's role extends beyond mere utility scripting; it is integral for data processing and backend logic, interacting effectively with the ASP.NET Core and external APIs (Google Earth Engine API) to manage data flows and application logic.

5.3.4 Node.js

Node.js is renowned for its non-blocking, event-driven architecture that makes it particularly suited for building scalable network applications. It was developed by Ryan Dahl and initially released in 2009. Node.js operates on a single-threaded event loop, using non-blocking I/O calls, allowing it to handle tens of thousands of concurrent connections efficiently [17].

Node.js includes a rich ecosystem of open-source libraries available through the Node Package Manager (NPM), which further accelerates development and deployment processes.

In the current system, Node.js is used for the Assistant Service implementation, leveraging the existing Node.js OpenAI library to interact with the API. This setup enables simple communication with the OpenAI services.

5.3.5 MassTransit

MassTransit is a high-performance, open-source message bus framework for .NET, designed to facilitate communication between services in a distributed system. Developed primarily to make scalable applications easier to build, MassTransit provides an extensive set of features for handling complex messaging patterns and workflows. It supports a variety of transport layers, including RabbitMQ, Azure Service Bus, and Amazon SQS, among others. MassTransit simplifies the development of event-driven services by abstracting the complexities associated with raw messaging infrastructures.

One of the major advantages of using MassTransit is this provider agnosticism. It allows developers to switch or upgrade the underlying messaging infrastructure

without major code changes, enabling better adaptability and future-proofing of the system architecture. Whether the application uses RabbitMQ, Azure Service Bus, or Amazon SQS, MassTransit interfaces with them through a consistent API, minimizing disruptions and learning curves when changes to the infrastructure are made [18].

In the current system, MassTransit ensures that our microservices, such as the Engine Service and Communication Service remain decoupled and modular. This separation of concerns allows for modifications or replacements of parts of the infrastructure with minimal impact on the overall system, leading to more scalable and maintainable architecture.

5.3.6 Docker and Docker Compose

Software is delivered in packages known as containers using OS-level virtualization via a group of platform-as-a-service (PaaS) technologies named Docker. Containers can communicate with one other over well-defined channels; they are segregated from one another and come with their own software, libraries, and configuration files. Compared to typical virtual machines, containers are lighter since they are all powered by a single operating system kernel [19]. It was as developed by Solomon Hykes as an internal project within dotCloud. It was officially introduced to the public in March 2013 during the PyCon conference [20].

Docker Compose, on the other hand, is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services, which allows you to create and start all the services from your configuration with just a single command. It was initially introduced under the name "Fig" in 2013, developed by a company called Orchard Laboratories Ltd. Docker later acquired Orchard in 2014, and Fig was rebranded as Docker Compose.

In this system, Docker and Docker Compose play critical roles by ensuring that the application is environment-agnostic and can be deployed consistently across any platform, enhancing both development and production efficiencies. This setup simplifies complexities related to installing and configuring different components of our technology stack and ensures that team members have an easy-to-reproduce local development environment.

Docker's lightweight containerization technology also improves the scalability and portability of applications, making it an invaluable tool in modern software development.

5.3.7 PostgreSQL

PostgreSQL, often simply called Postgres, is an advanced open-source relational database management system (RDBMS) that is widely praised for its proven architecture and robust feature set.

PostgreSQL originated from the POSTGRES project, which was led by Professor Michael Stonebraker at the University of California, Berkeley, starting in 1986. The project's goal was to build upon the ideas of the earlier Ingres database and to break new ground in database concepts such as the handling of null values and ensuring data integrity through primary keys.

The name changed to PostgreSQL to reflect its support for SQL, which was added in the mid-1990s. This adaptation allowed it to take advantage of the burgeoning popularity of the SQL language in business and academia, thus broadening its use case and user base [21].

Unlike many other database management systems which are controlled by single corporate entities, PostgreSQL is not owned by any one company. It is developed and maintained by a diverse group of software developers and companies who are passionate about database technology. This community-driven approach ensures that PostgreSQL continues to evolve in response to real-world use and modern technology trends.

Chapter 6

Implementation

This chapter will present the practical execution of the theoretical designs and architectural frameworks outlined in previous section of this thesis. The focus here is on how the proposed emergency management system is brought to life, highlighting the specific technologies, methodologies, and coding practices used. The implementation phase is crucial as it transforms conceptual models and architectural plans into a real world system.

Also, in this chapter are presented the challenges encountered during the implementation process and the strategies adopted to overcome them, providing a real-world perspective on the application of software engineering principles.

6.1 Implementation of Backend Services

As stated in the previous chapter, the system employs a microservices architecture, with the backend logic of the application distributed across multiple components. Each component plays a specific role and communicates effectively with the others to provide a robust and scalable system. Now, each of the services will be presented along with its implementation challenges and trade-offs.

6.1.1 API Gateway

The API Gateway, a important component in the architecture of our emergency management system, was implemented using Nginx.

Nginx functions as a reverse proxy, directing client requests to the appropriate backend services efficiently. This decouples the client from the backend components and therefore it maintains a clean architecture and separation of concerns. Nginx was chosen for its high performance, stability, and low resource consumption, which are critical in handling high traffic volumes and maintaining quick response times during emergencies. Although currently the system does not handle

scaling based on load, the API Gateway was introduced with the future in mind. This layer of abstraction will allow the system to scale independently, and enable the API Gateway to act as a load balancer as well. Here is a subsection from the nginx.conf file that configures how the requests should be forwarded to the Engine Service.

```
http {
   upstream engine_service {
      server host.docker.internal:8181;
   }

   upstream communication_service {
      server host.docker.internal:8282;
   }

   upstream assistant_service {
      server host.docker.internal:3001;
   }

   server {
      listen 80;
      server_name localhost;

      location /engine/ {
         if ($request_method = 'OPTIONS') {
            add_header 'Access-Control-Allow-Origin' 'http://localhost:4200';
            add_header 'Access-Control-Allow-Methods' 'GET, POST, OPTIONS, DELETE, PUT';
            add_header 'Access-Control-Allow-Headers' 'Content-Type, Access-Control-Allow-Headers, Authorization,
                  X-Requested-With';
            add_header 'Access-Control-Allow-Credentials' 'true';
            return 204;
         }

         proxy_pass http://engine_service/;
         proxy_http_version 1.1;
         proxy_set_header Origin $http_origin;
         proxy_set_header Upgrade $http_upgrade;
         proxy_set_header Connection "upgrade";
         proxy_set_header Host $host;
         proxy_cache_bypass $http_upgrade;
         proxy_ssl_verify off;
      }
   }
}
```

Listing 6.1: "Subsection of Nginx configuration file"

6.1.2 Engine Service

The Engine Service is the backbone of our backend architecture, managing key operations such as user management, event registration and core logic. It interacts directly with the database for data operations and facilitates system-wide communication through the message bus.

This was implemented using ASP.NET Core Web API, utilizing a three-layered architecture consisting of the API Layer, the Business Layer, and the Data Access Layer.

The API Layer exposes RESTful APIs and provides the interface for using the service. It handles all incoming HTTP requests, ensuring they are correctly routed to the appropriate endpoints. To secure API access, JSON Web Tokens (JWT) are used for authentication and authorization, ensuring that only authenticated and autho-

rized users can access certain endpoints. This layer also and manages CORS (Cross-Origin Resource Sharing) policies to control how resources are shared between different domains. By abstracting the underlying logic, the API Layer simplifies client interactions and ensures a standardized communication protocol.

The Business Layer contains the core business logic of the service. It processes data received from the API Layer, applying business rules, validations, and algorithms necessary for the operation of the system. This layer is responsible for executing the core functionalities. By separating business logic from data access and presentation concerns, it ensures that the system remains modular, maintainable, and scalable.

The Data Access Layer handles all interactions with the database, ensuring that data is efficiently retrieved, stored, and updated. Using an ORM (Object-Relational Mapping) framework like Entity Framework, this layer abstracts the complexity of database operations, providing a simplified interface for CRUD (Create, Read, Update, Delete) operations. It also implements transaction management and connection handling to optimize performance and ensure data integrity. By isolating data access logic, this layer enhances security and protects against SQL injection attacks.

The Engine Service interacts with the Communication Service when emergency events are triggered or modified. This interaction occurs through the message bus, allowing for decoupled and efficient communication between services.

```
public async Task PublishEmergencyReportedAsync(EmergencyEventDto emergencyEventDto, string? username)
   {
      try
      {
         var emergencyEventEntity = await _emergencyEventRepository.GetEmergencyEventByIdAsync(emergencyEventDto.Id);
         if (emergencyEventEntity == null) throw new Exception("Event not found");

         await _publishEndpoint.Publish(new EmergencyReportedEvent()
         {
            Id = emergencyEventEntity.Id,
            Description = emergencyEventEntity.Description,
            Location = emergencyEventEntity.Location,
            Latitude = emergencyEventEntity.Latitude,
            Longitude = emergencyEventEntity.Longitude,
            Type = emergencyEventEntity.Type,
            Severity = emergencyEventEntity.Severity,
            Status = emergencyEventEntity.Status,
            ReportedBy = emergencyEventEntity.ReportedBy ?? Guid.Empty,
            ReportedByUsername = username ?? "Anonymous",
            ReportedAt = emergencyEventEntity.ReportedAt,
            UpdatedAt = emergencyEventEntity.UpdatedAt
         });
      }
      catch (Exception ex)
      {
         _logger.Error(ex, "An error occurred while publishing user created event");
         throw;
      }
   }
```

Listing 6.2: ”Method to publish emergency events to the message bus”

6.1.3 Communication Service

The Communication Service is a critical component of our emergency management system, responsible for managing all communication channels within the system. This includes handling emails, real-time chat conversations, notifications, and alerts, ensuring effective and timely communication during emergency situations.

This service is also implemented using ASP.NET Core Web API, following the same structured approach as the Engine Service with distinct API, Business, and Data Access Layers. In addition to handling standard HTTP requests, this service incorporates SignalR to support real-time chat functionality through WebSockets. SignalR is a library that simplifies adding real-time web functionality to applications, allowing server-side code to push content to clients instantly. This integration ensures seamless and instant communication, which is critical for effective emergency management, enabling users to exchange messages and receive updates in real time.

```
[Authorize]
public class ChatHub : Hub
{
   private readonly IChatService _chatService;
   private readonly ILogger _logger;
   public ChatHub(IChatService chatService, ILogger logger)
   {
      _chatService = chatService;
      _logger = logger;
   }

   public async Task<Task> SendMessageToChatGroup(string chatId, string message)
   {
      var userId = Context.User?.FindFirst(CustomClaimTypes.UserId)?.Value;
      _logger.Information($"Sending message to chat with id {chatId} from user with id {userId}");
      var result = await _chatService.SaveMessageAsync(new Guid(chatId), userId, message);
      var messageDto = new EnhancedMessageDto()
      {
         Id = result.Data.Id,
         Text = result.Data.MessageText,
         Date = result.Data.Date,
         Username = result.Data.Username
      };

      return Clients.Group(chatId.ToUpper()).SendAsync("ReceiveMessage", messageDto);
   }
   public override async Task OnConnectedAsync()
   {
      var userId = Context.User?.FindFirst(CustomClaimTypes.UserId)?.Value;

      if (userId.IsNullOrEmpty())
      {
         await Clients.Caller.SendAsync("UnauthorizedAccess", "You are not authorized.");
         await Task.Delay(500);
         Context.Abort();
      }

      var result = await _chatService.GetUserChatIds(userId);
      if (!result.IsSuccess) return;

      var chats = result.Data;

      foreach (var chatId in chats)
      {
         _logger.Information($"user with id {userId} is in chat with id {chatId}");
         await Groups.AddToGroupAsync(Context.ConnectionId, chatId.ToString().ToUpper());
      }

      await base.OnConnectedAsync();
   }
```

```
    public override async Task OnDisconnectedAsync(Exception exception)
    {
        var userId = Context.User?.FindFirst(CustomClaimTypes.UserId)?.Value;
        var result = await _chatService.GetUserChatIds(userId);

        if (!result.IsSuccess) return;

        var chats = result.Data;

        foreach (var chatId in chats)
        {
            await Groups.RemoveFromGroupAsync(Context.ConnectionId, chatId.ToString().ToUpper());
        }

        await base.OnDisconnectedAsync(exception);
    }
}
```

Listing 6.3: "SignalR Chat Hub for Real-time Communication feature"

This service also handles the generation and distribution of email notifications. For instance, when a new emergency event is reported in the Engine Service, an event is wirtten to the message bus and the Communication Service constructs an email detailing the event's specifics such as location, severity, and type and sends it to relevant stakeholders. Below is an example of a consumer class that handles this process.

```
public class EmergencyReportedEventConsumer : IConsumer<EmergencyReportedEvent>
{
    private readonly IMailSendingService _mailSendingService;
    private readonly Serilog.ILogger _logger;
    private readonly IChatRepository _chatRepository;
    public EmergencyReportedEventConsumer(IMailSendingService mailSendingService, ILogger logger, IChatRepository
        chatRepository)
    {
        _mailSendingService = mailSendingService;
        _logger = logger;
        _chatRepository = chatRepository;
    }
    public async Task Consume(ConsumeContext<EmergencyReportedEvent> context)
    {
        try
        {
            _logger.Information($"Consuming EmergencyReportedEventConsumer: Id: {context.Message.Id}");
            await _chatRepository.AddChatEventAsync(context.Message.Id, $"Emergency at {context.Message.Location}");
            var administrators = await _chatRepository.GetAdministratorsAsync();
            foreach (var admin in administrators)
            {
                await _mailSendingService.SendEmailAsync(
                 MailRequest.EmergencyReported(admin.User.Email,context
                ));
            }
        }
        catch (Exception ex)
        {
            _logger.Error(ex,$"Error while consuming EmergencyReportedEventConsumer: {ex.Message}");
            throw;
        }
    }
}
```

Listing 6.4: "Consumer class for EmergencyRegisteredEvent"

6.1.4 Assistant Service

The Assistant Service abstracts away the complexities of communicating with the OpenAI Assistants API, providing an easy-to-use interface for seamless integra-

tion.Having a valid OpenAI API key, this service allows the creation of a specialized assistant tailored with specific prompts related to our system. This setup ensures that the assistant can effectively assist users by providing detailed and relevant information about the Sky Sentinel application.

This service is implemented using Node.js and Express, creating API endpoints for CRUD operations on assistants, threads and messages. To ensure the security and reliability of the interactions, the Assistant Service also uses JSON Web Tokens. These tokens are used across all backend services, maintaining a consistent and secure authentication mechanism.

```
const OpenAI = require("openai");

async function createAssistant() {
  const secretKey = process.env.OPENAI_API_KEY;
  if (!secretKey) {
    console.error("OpenAI API key is missing.");
    return;
  }

  const openai = new OpenAI({
    apiKey: secretKey,
  });

  try {
    const assistantResponse = await openai.beta.assistants.create({
      name: "Sky Sentinel Assistant",
      instructions: process.env.ASSISTANT_INSTRUCTIONS,
      tools: [],
      model: "gpt-3.5-turbo-16k",
    });

    return assistantResponse;
  } catch (error) {
    console.error("Error creating OpenAI assistant:", error);
  }
}

module.exports = { createAssistant };
```

6.1.5 Landcover Analysis Service

The last backend service of the system is the Landcover Analysis Service. This service handles the processing of satellite imagery, delivering essential land cover analysis for emergency management. It retrieves images through the Google Earth Engine API and utilizes semantic segmentation model based on DeepLabV3+ [22] to examine the terrain an extract the main roads.

The implementation was done using Python and FastAPI. The service interacts with the Google Earth Engine API to fetch the necessary satellite imagery. For the analysis, pre-trained AI segmentation model is used to find and extract main roads. This approach ensures accurate and timely analysis, helping in the development of effective emergency response plans.

6.2 Frontend implementation details

The frontend of the emergency management system is implemented using Angular, a powerful and versatile framework for building dynamic single-page applications. To enhance the user interface, we have integrated PrimeNG, a rich set of open-source UI components that provide a visually appealing and responsive design. This combination ensures that the application is both robust and user-friendly, essential qualities for an emergency management system. Reactive programming with RxJS (Reactive Extensions for JavaScript) is used to handle asynchronous data streams and event handling, making it easier to manage complex data flows and ensuring that the user interface remains responsive and performant.

The user interface is designed with simplicity and ease of use in mind, crucial for an emergency management system where users must navigate quickly and efficiently, even under stressful conditions. Key design principles include intuitive navigation with clear and accessible menus and buttons, a responsive design that works seamlessly across various devices, real-time updates via RESTful APIs and WebSockets, and adherence to accessibility standards.

Additionally, Leaflet maps integration is utilized to provide an interactive and user-friendly map feature. Leaflet is an open-source JavaScript library that makes it easy to create mobile-friendly, interactive maps [23]. This integration enhances the visual representation of emergency locations and statuses, allowing users to easily navigate and understand geographic information, which is critical in emergency response scenarios.

6.3 Integration with external APIs

This section details the integration of various external APIs within our emergency management system, which enhance its functionality and efficiency. The system leverages several external APIs to provide advanced features and seamless user experience.

For user authentication and authorization, the system integrates Google OAuth to allow users to login and register using their Google accounts [24]. This integration simplifies the authentication process, enhances security, and provides a seamless user experience by leveraging Google's robust authentication infrastructure.

The sequence diagram bellow details the OAuth authentication process utilized in the system to facilitate user login via Google. Initially, the user clicks the 'login with Google' button, prompting the UI to redirect to Google's authentication service. After the user provides their credentials, Google returns an authorization code and a token.The token is sent to the Engine Service through the API Gateway. Upon

validating this token, the Engine Service stores the user's details in the database, generates a JSON Web Token (JWT), and sends it back through the API Gateway to the UI. This JWT is then used by the user for getting access within the system.

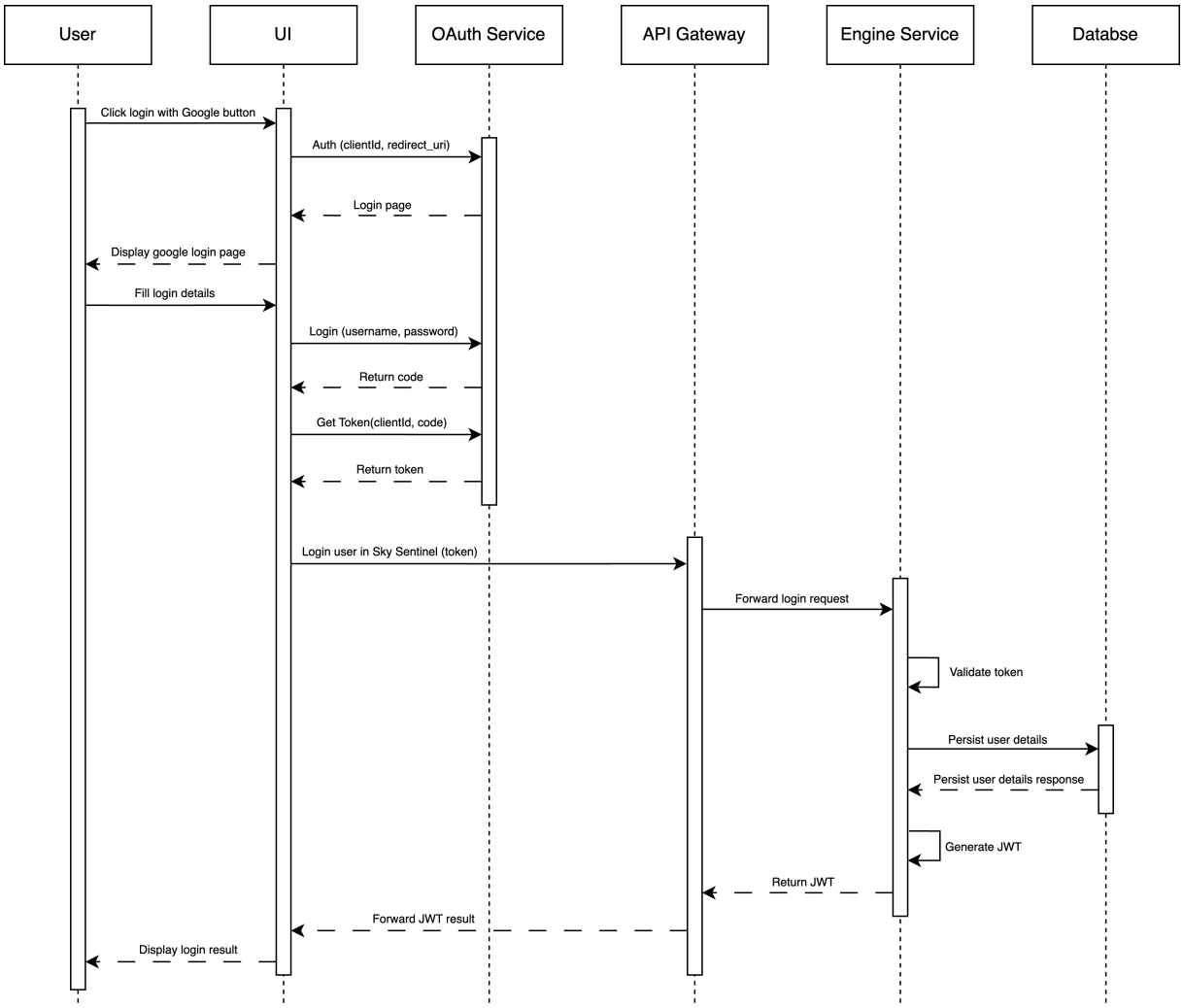


Figure 6.1: OAuth Authentication flow for Google Login

The OpenAI Assistants API is used to facilitate intelligent user interactions, offering automated assistance and guidance [25].

The Google Maps API and Autocomplete API are integrated to provide accurate location services and streamlined user input for addresses and places [26] .

For processing satellite imagery, the Google Earth Engine API is utilized, enabling sophisticated landcover analysis crucial for emergency response planning [27].

Additionally, the Cloudinary API is employed as a Content Delivery Network (CDN) to efficiently store and serve images [28].

To handle email communications, the system utilizes the Gmail SMTP server, ensuring reliable and secure delivery of emails [29].

Each of these external APIs plays a vital role in enhancing the overall capabilities of the system, ensuring it meets the high demands of emergency management.

6.4 Main features

6.4.1 Reporting an Emergency Event

To ensure rapid response capabilities, the user interface for reporting an emergency event has been designed with simplicity and efficiency. The feature is accessible directly via a prominent red button on the homepage, allowing any user who joins the platform to report an emergency without navigating through complex menus. This approach ensures that in critical situations, reporting an emergency is as quick and user-friendly as possible.

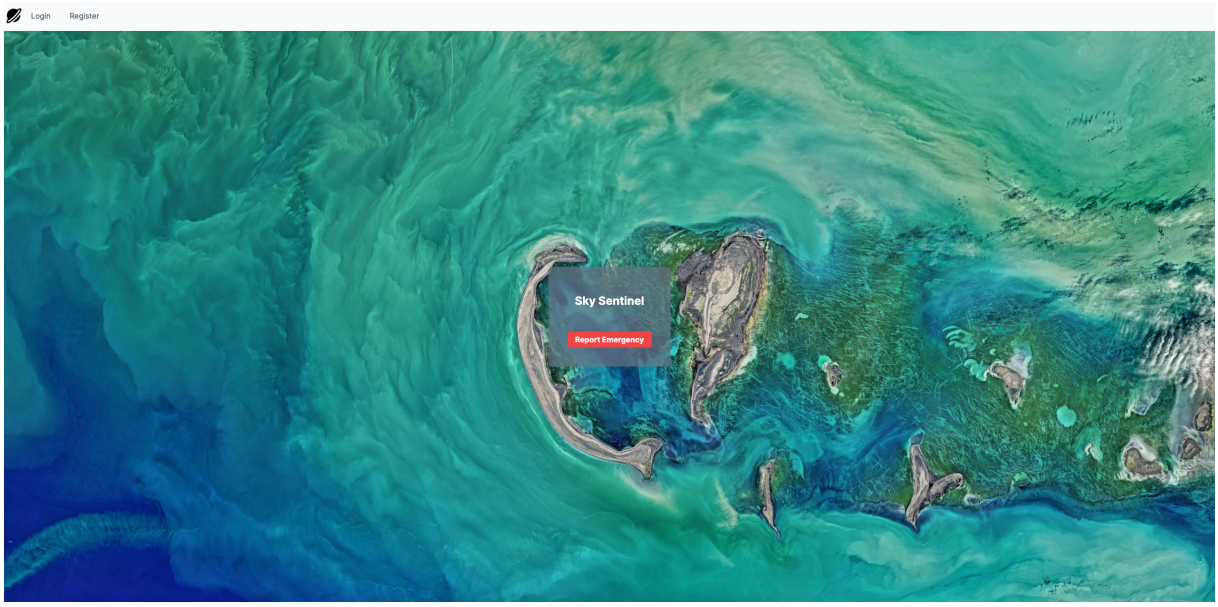


Figure 6.2: Homepage

The form itself is straightforward, requiring only essential information to initiate a response. Importantly, users are not required to create an account to report an emergency, minimizing barriers to crucial interaction. Account creation is only necessary for those who wish to access additional features of the app.

In the emergency event reporting form, users can specify the severity of the incident by choosing from predefined categories, which helps in prioritizing the response appropriately. For location details, the form offers the convenience of automatically capturing the user's current location, ensuring rapid reporting without the need to manually enter an address. If a different location needs to be specified, the form integrates Google's Autocomplete feature, allowing users to quickly search and select precise locations. This functionality not only speeds up the entry

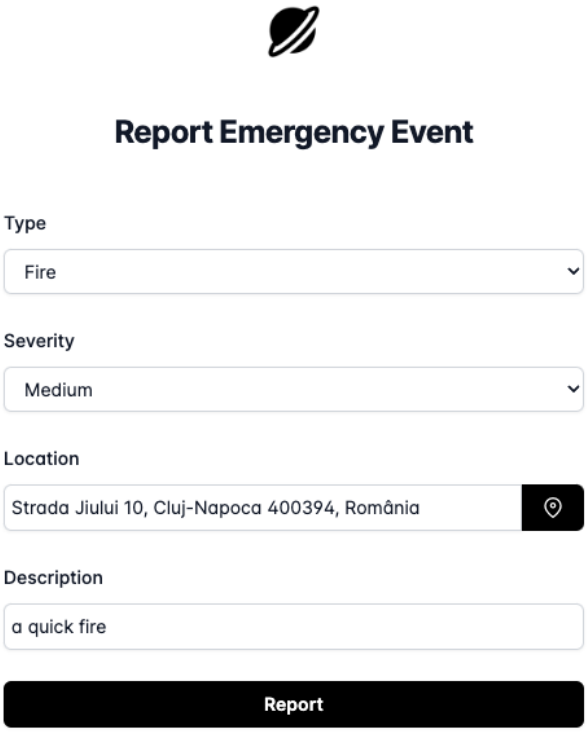


Figure 6.3: Report Emergency Event form

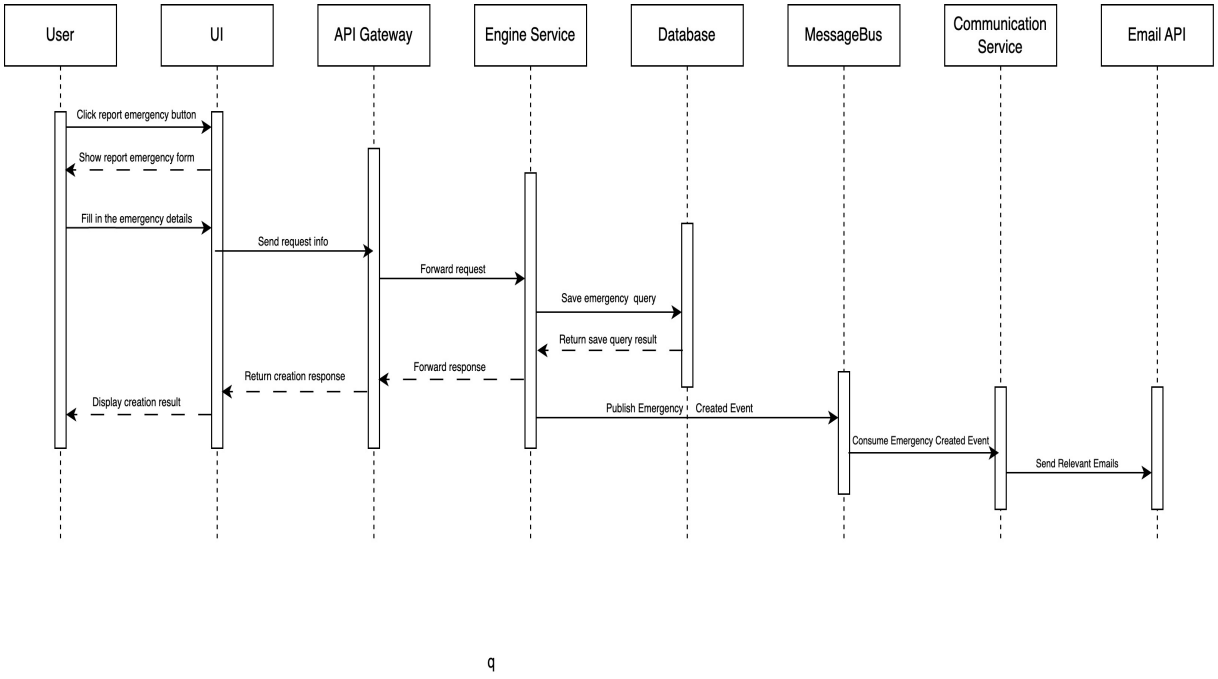
process but also enhances the accuracy of the location data provided to emergency responders.

Additionally, the form is designed with a user-friendly interface that features clear, easy-to-understand instructions and visually distinct buttons and fields to reduce user errors and streamline the submission process. The form also includes tooltips and help icons that provide users with information or clarifications as needed, ensuring they understand what information is required and why it is necessary. For enhanced security, the form incorporates data validation checks to prevent the submission of incorrect or malicious data, which is crucial for maintaining the integrity of the emergency response system. Moreover, the form is responsive, adjusting to different device screens, ensuring accessibility and ease of use whether accessed from a desktop, tablet, or smartphone. This adaptability is vital for situations where users might need to report emergencies on the go.

The sequence diagram illustrates the process of reporting an emergency event within the system. It begins when a user clicks the report emergency button on the user interface (UI), which then displays the report emergency form. After the user fills in the necessary emergency details and submits the form, the information is sent through the API Gateway. The API Gateway forwards the request to the Engine Service, which processes the request by saving the emergency details into

the Database and then returns the results back through the API Gateway to the UI, where the user sees the result of their report submission.

Simultaneously, once the emergency details are saved, the Engine Service publishes an ”Emergency Created Event” to the MessageBus. The Communication Service listens on the MessageBus for such events, and upon receiving this notification, it consumes the event and triggers the Email API to send relevant email notifications. This step ensures that appropriate personnel or systems are alerted about the new emergency event promptly.



q

Figure 6.4: Report Emergency flow

6.4.2 Interactive Dashboard

The interactive dashboard is a crucial component of the emergency management system's user interface, implemented using the Leaflet.js library. This map-based interface provides a real-time overview of various emergency events across the region, each represented by color-coded markers that denote the severity of the events. Users can see a wide array of events distributed geographically across the map. This spatial representation helps users quickly assess the areas with high activity and potential emergencies that need immediate attention. Each marker on the map has a different color corresponding to the severity of the emergency. For instance, red markers might represent critical emergencies, yellow markers for moderate severity, and green for minor issues.

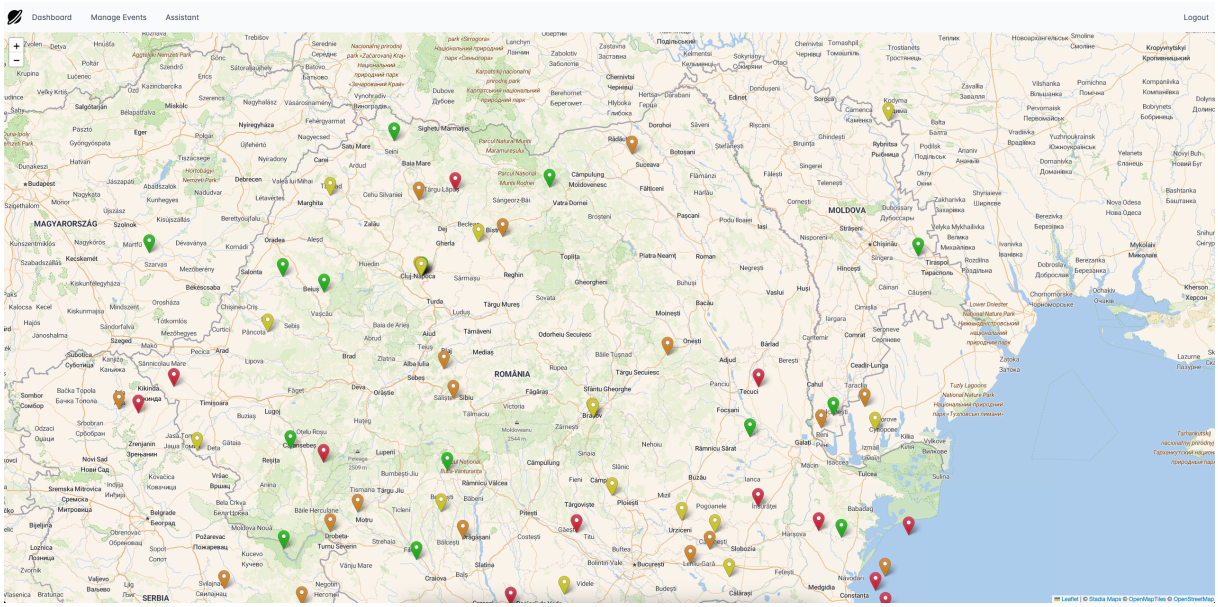


Figure 6.5: Interactive dashboard

By clicking on any marker, users can view a brief summary of the emergency, including details such as the type of event, the time reported, and other relevant data. This feature provides a quick snapshot of the event without overwhelming the user with information.

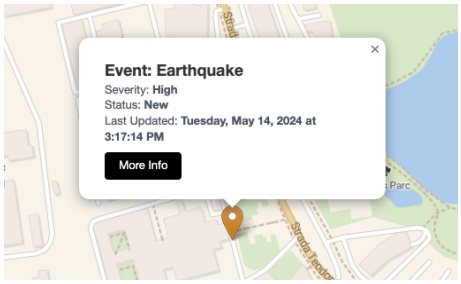


Figure 6.6: Emergency event preview

From the brief summary view, users have the option to navigate to a dedicated page for each event. This page contains more detailed information and possible actions. This allows users to engage with each event comprehensively and coordinate response efforts directly from the dashboard.

The sequence diagram below shows how, after a user clicks the dashboard button, the system retrieves and displays markers on the interactive map. It traces the data flow from the user interface, through the API Gateway and Engine Service, to fetching the geographic markers.

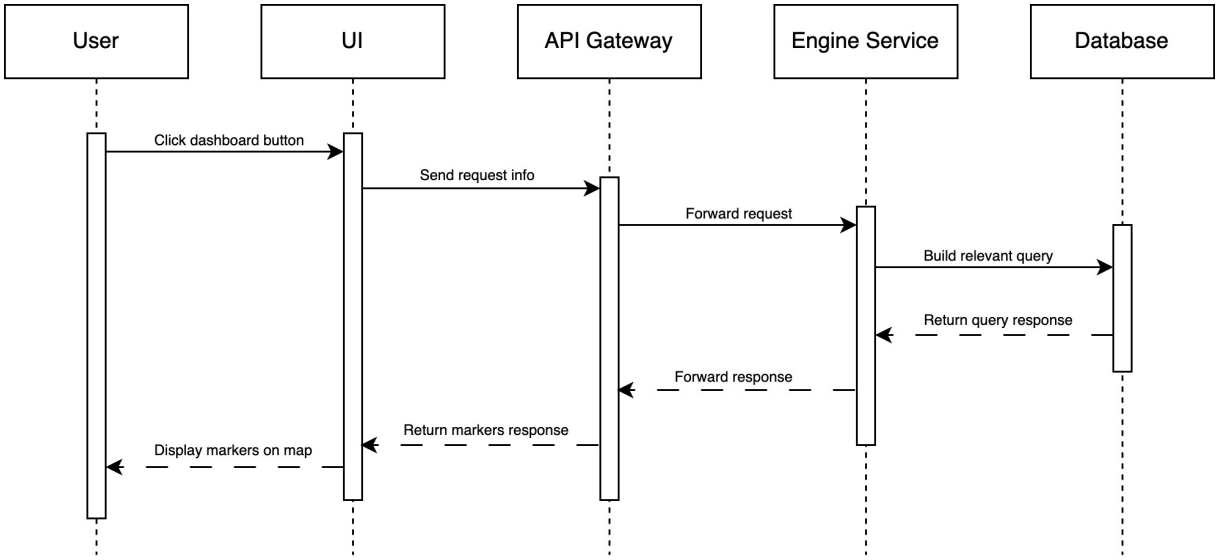


Figure 6.7: Interactive dashboard markers fetching

6.4.3 Landcover analysis

The landcover analysis feature has been designed to focus specifically on extracting main roads from satellite imagery in emergency regions. This tool is useful for administrators managing emergency responses. When a new emergency is reported, they can use the ”analyse landcover” function to quickly identify key roadways in the affected area. This feature helps administrators in finding effective response routes and logistics, ensuring rapid and informed decision-making in critical situations.

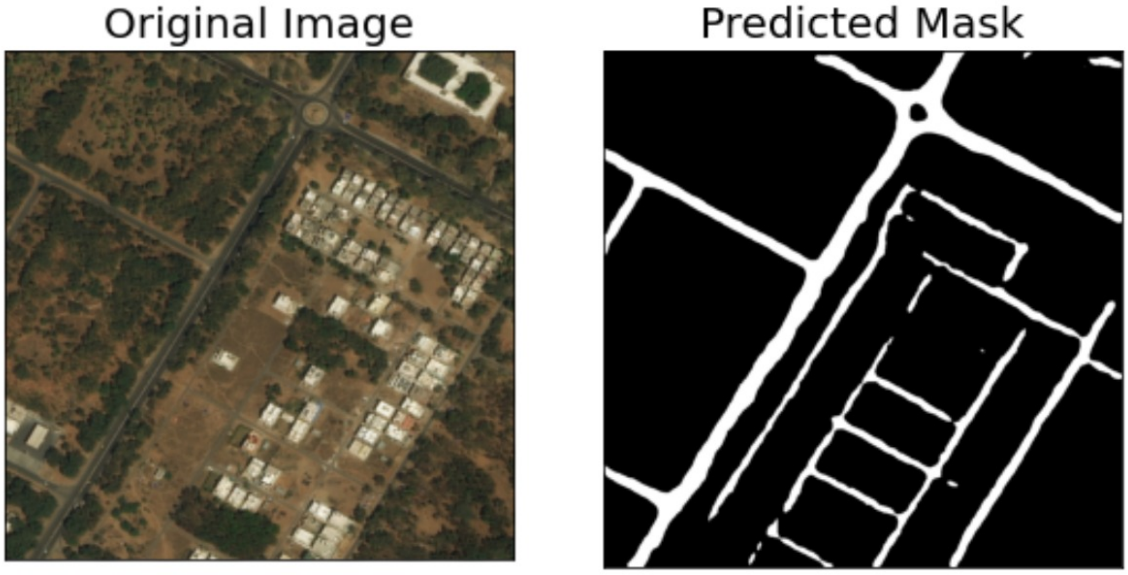


Figure 6.8: Road extraction [22]

The sequence diagram bellow illustrates the workflow for analyzing landcover in the emergency management system. When a user clicks the ”analyse landcover” button, the UI sends a request to the API Gateway, which forwards it to the Landcover Analysis Service. The service initiates a process to fetch a satellite image from the Google Earth Engine API, which is then returned for AI-driven image processing. After processing, the image is uploaded to Cloudinary CDN, and the URL of the processed image is received and stored in the database. Throughout this process, the user’s UI is updated in real-time via a WebSocket connection, which initially confirms receipt of the request with a 202 Accepted status and later updates the user with the final result. This WebSocket connection is established at the beginning and disconnected after the final notification. The UI queries for the result, the system fetches the stored URL from the database and displays the analyzed image, completing the workflow.

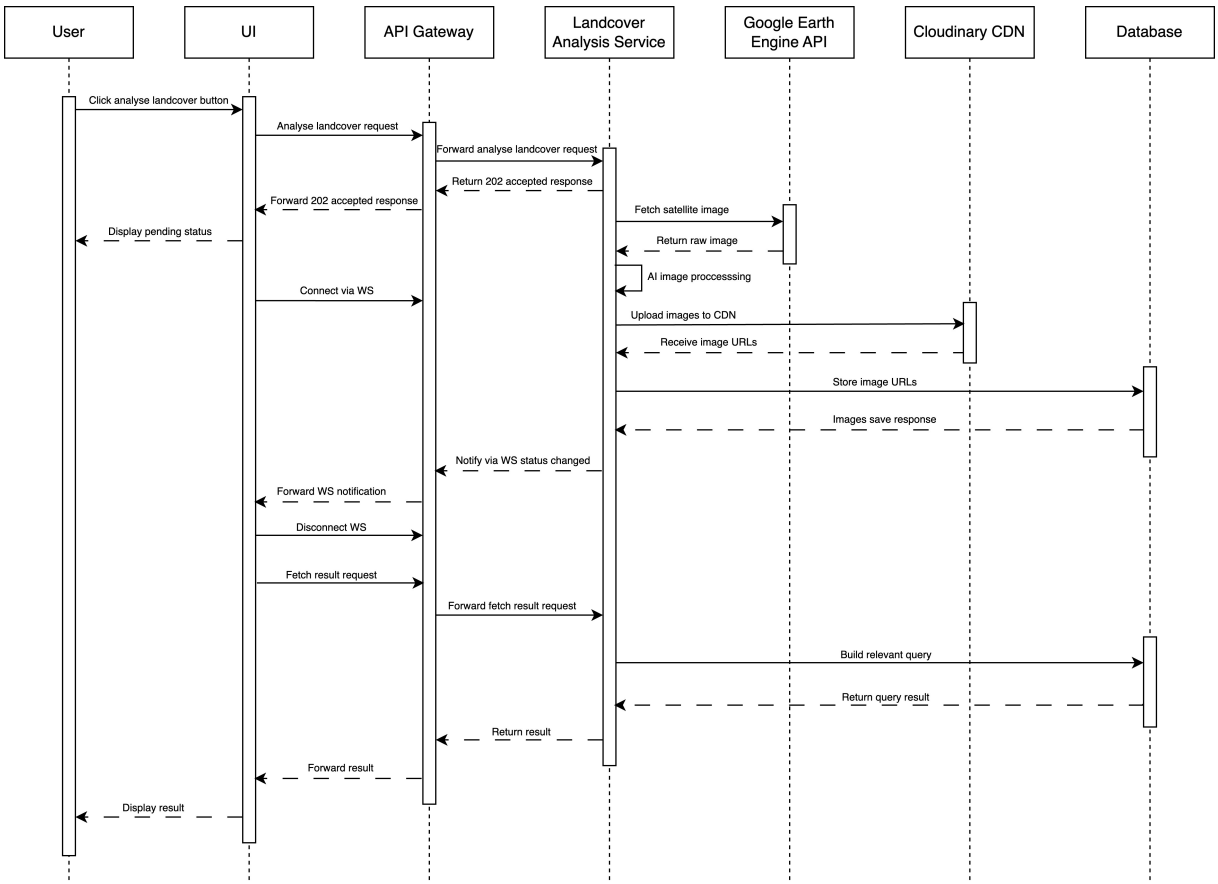


Figure 6.9: Landcover analysis process

6.4.4 Real-time communication and AI assistant

This feature is designed to facilitate easy coordination and communication during ongoing emergency events. Users who register as volunteers or participants for an active emergency event gain access to a dedicated real-time chat. This communication channel connects them directly with other volunteers and the administrative personnel managing the event. It allows for the instantaneous exchange of information, coordination of efforts, and quick updates, ensuring that all involved parties are well-informed and can respond efficiently to the evolving situation. This real-time communication capability is crucial for the dynamic environment of emergency management, where timely and clear communication can significantly impact the outcome.

From the list of events they are volunteering for, users can easily click on the chat icon to quickly access all recent updates and active conversations.

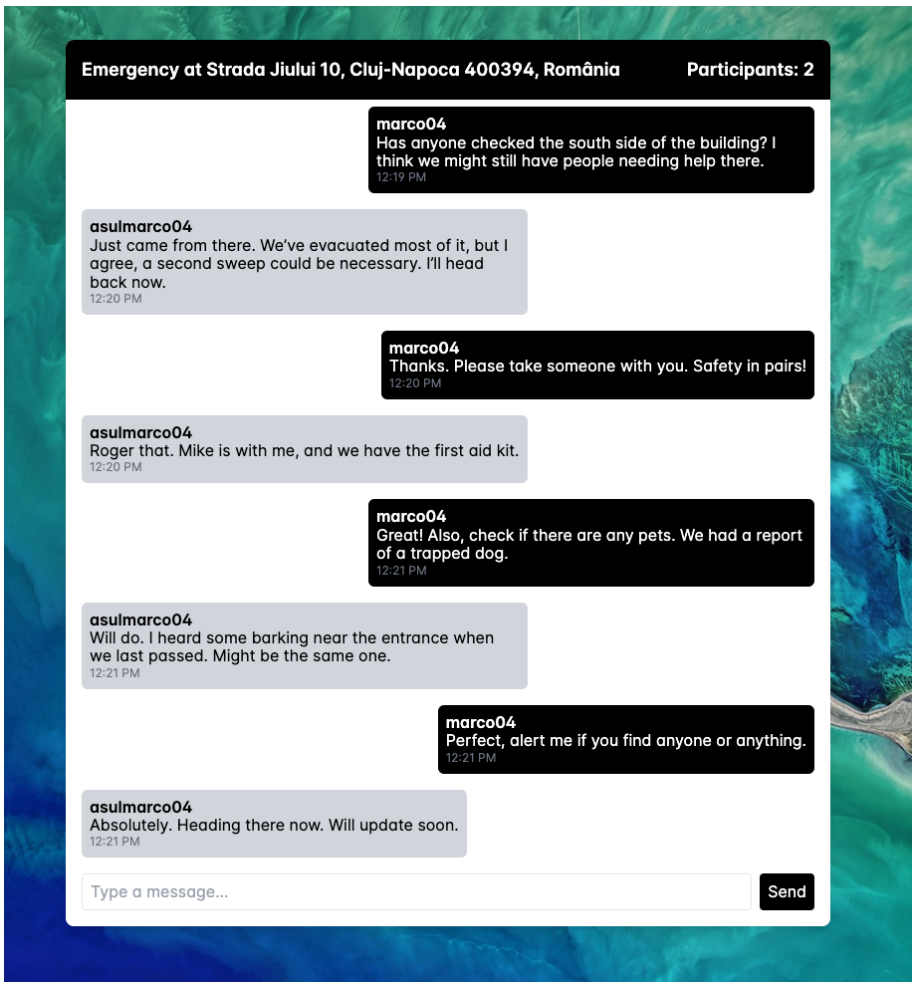


Figure 6.10: Real-time communication between emergency responders

The sequence diagram illustrates the workflow of the real-time chat feature within the system. It begins when a user accesses the conversation through the user in-

terface (UI). The UI sends a GET request via the API Gateway to retrieve existing messages, which forwards the request to the Communication Service. This service then queries the Database to fetch the requested messages and returns the results back through the API Gateway to the UI, where the messages are displayed to the user.

Concurrently, the user connects to the chat via WebSockets (WS), establishing a real-time communication channel. When the user sends a new message, it is transmitted through this WebSocket connection to the Communication Service, which forwards the request to save the message into the Database. Upon successful storage, the Database sends a confirmation back to the Communication Service, which then uses WebSockets to broadcast the new message to all connected users, ensuring that everyone in the conversation receives the update instantly. This setup provides a seamless and efficient communication experience, allowing users to engage in real-time discussions and share critical updates promptly.

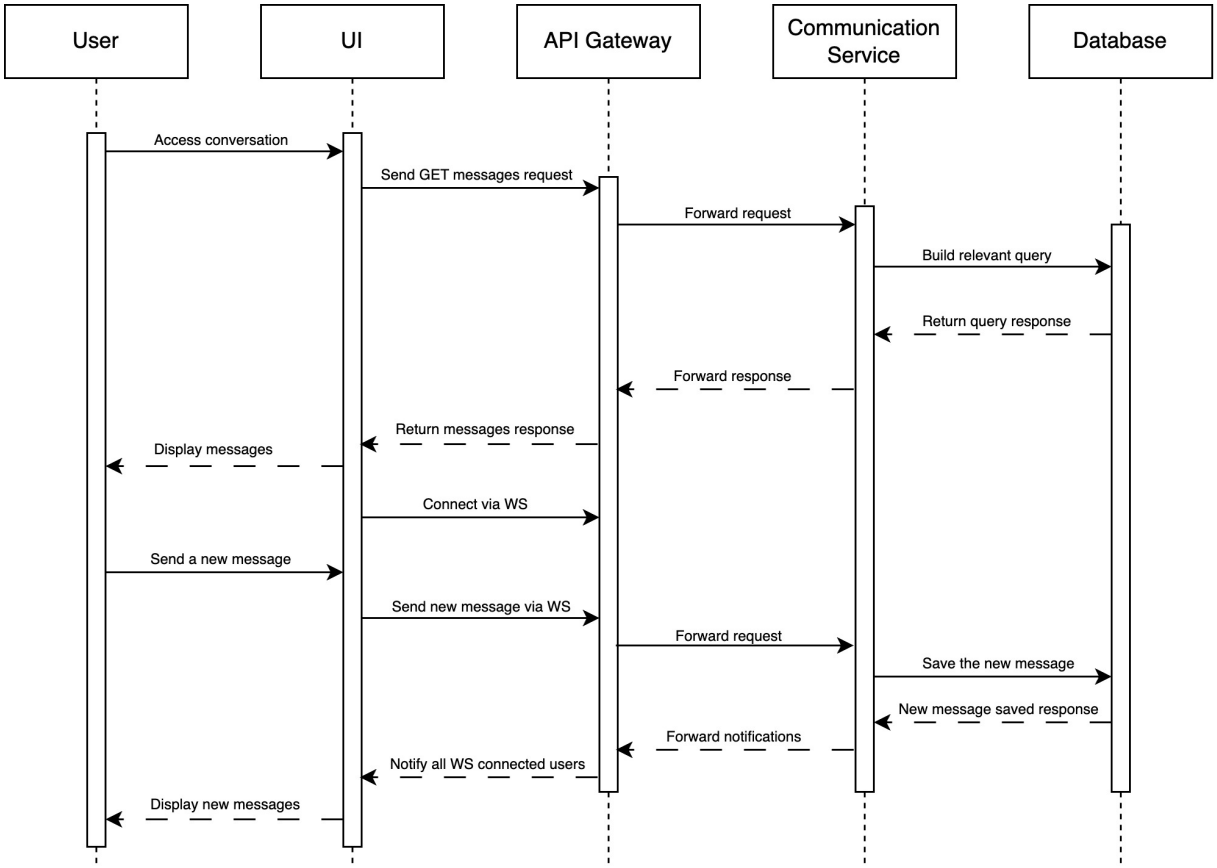


Figure 6.11: Sequence Diagram for Real-Time Chat Communication

Users can quickly access the AI Assistant by clicking on the assistant item within the Sky Sentinel navigation bar. This feature allows users to ask questions and receive instant responses about the application's functionalities and emergency procedures. Designed to be user-friendly, the assistant helps navigate the system's fea-

tures, making it easier for users to get the information they need without delay.

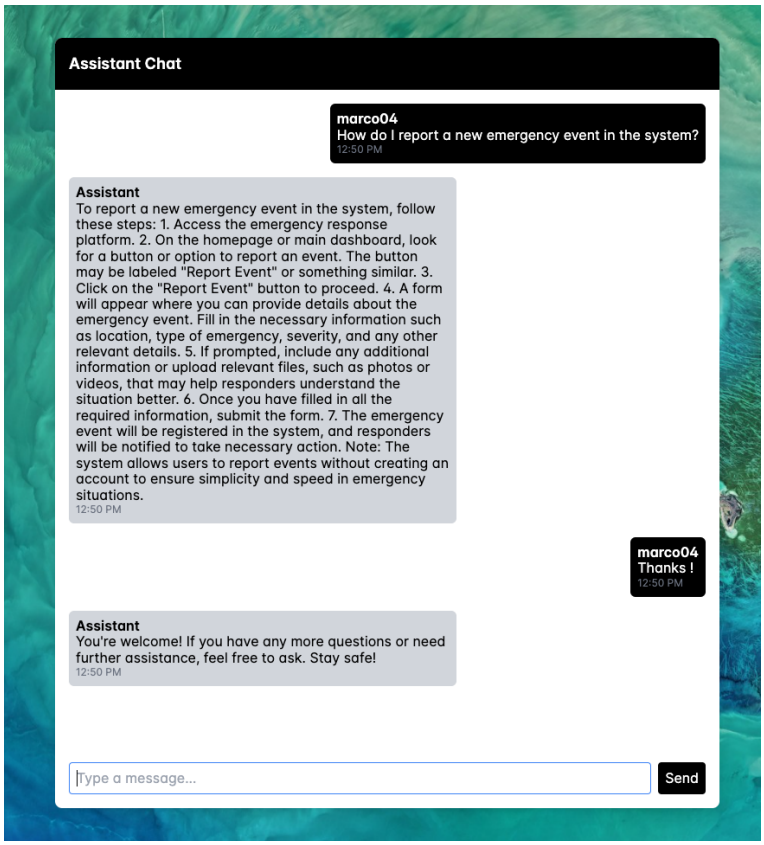


Figure 6.12: Sequence Diagram for Real-Time Chat Communication

The sequence diagram illustrates the detailed process of interacting with the AI Assistant within the system, beginning when a user clicks the assistant button in the user interface (UI). The UI initiates a request to the API Gateway to create a new assistant, which forwards this request to the Assistant Service. The Assistant Service then makes a call to the OpenAI Assistants API to create a custom assistant tailored for the user's needs. Upon successful creation, the Assistant Service requests the API to create a dedicated thread for ongoing interactions, allowing for structured and continuous dialogue.

Once the thread is established, the user can send messages through the UI, which are routed through the API Gateway and Assistant Service to the OpenAI API. Each message is processed and responded to in real-time, with responses returned via the same path back to the user's UI. This setup ensures a seamless and interactive communication flow between the user and the AI Assistant, allowing for immediate assistance and information exchange.

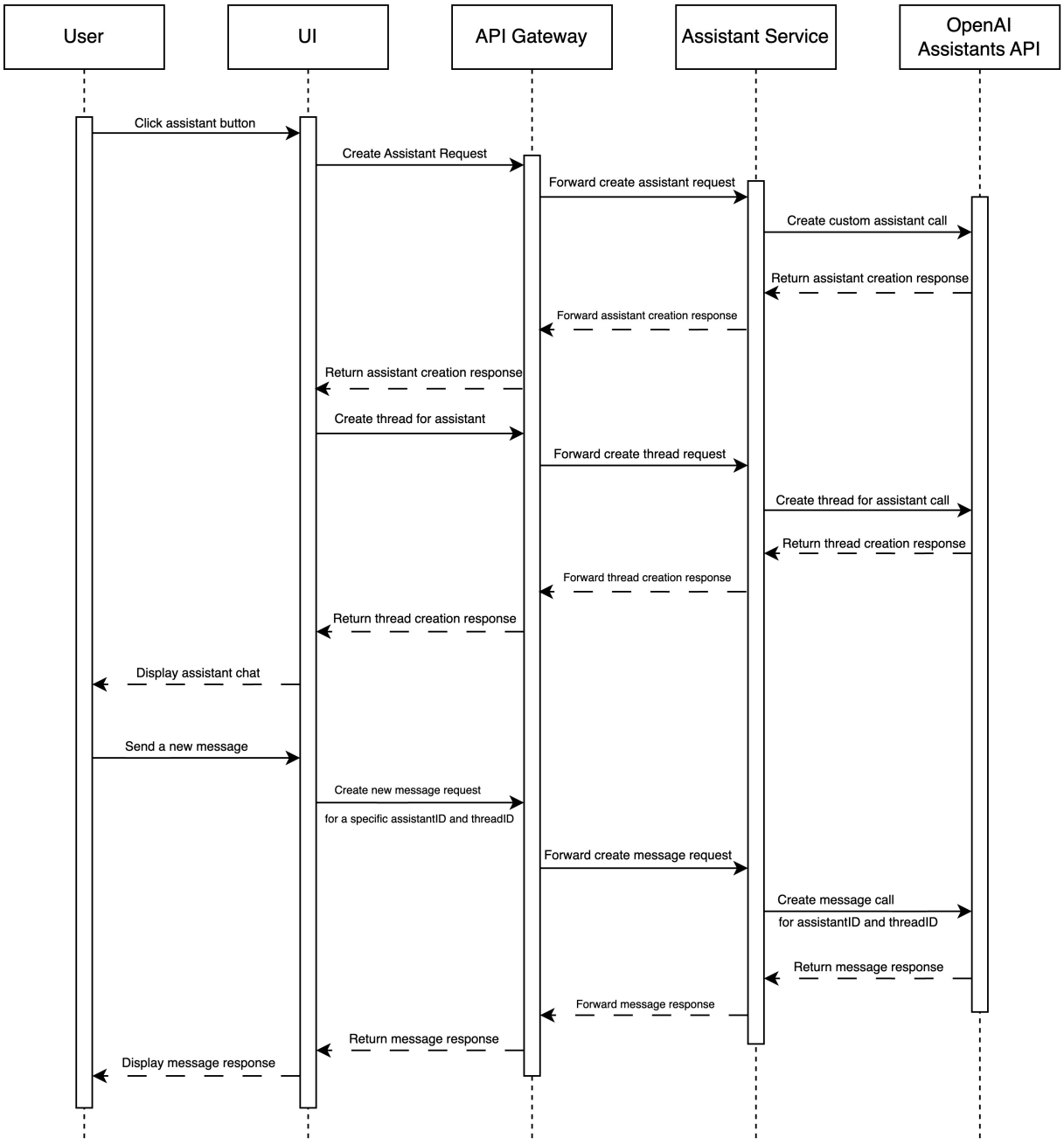


Figure 6.13: Sequence Diagram for AI Assistant interaction

6.5 Testing and Validation

Testing and validation are critical components of software engineering, ensuring that applications perform reliably and as intended. This is especially important in life-saving applications, where software failures can have severe consequences. Rigorous testing helps identify and resolve defects, ensuring the system is robust, secure, and capable of handling real-world scenarios.

6.5.1 Unit Testing

Unit testing is a fundamental testing strategy, focusing on verifying the functionality of individual components in isolation. The system includes unit tests across all backend microservices, including the Engine Service, Communication Service, and others. Unit testing is crucial as it helps detect bugs early in the development process, simplifies debugging, and ensures that each component behaves as expected. Reached the unit test coverage of 85%, providing a high level of confidence in the reliability of the backend services. This extensive coverage ensures that the core functionalities of each service are thoroughly validated.

6.5.2 System Testing

In addition to unit testing, comprehensive system testing has been conducted to validate the end-to-end functionality of the application. Using Playwright, end-to-end tests were performed on key functionalities, such as user login and reporting a new emergency event. System testing ensures that all integrated components work together and that the application meets its functional and performance requirements. By simulating real-world user interactions, it has been validated that the system performs reliably under various conditions, providing a robust and user-friendly experience for emergency responders.

6.5.3 Performance Testing

Performance testing is essential to ensure that the system can handle high loads and perform efficiently under stress. The map dashboard, a critical component of the application, underwent rigorous performance testing. The Faker library for Python was used to generate mock data, simulating a variety of emergency scenarios to test the system's performance under load.

The following table presents the tested load times for the map dashboard under different conditions:

Number of Events	Average Load Time (ms)	Maximum Load Time (ms)
100	150	200
500	300	400
1000	450	600
5000	800	1000
10000	1200	1500

Table 6.1: Performance testing results for the map dashboard under different loads.

These results indicate that the map dashboard performs well under various loads, maintaining acceptable response times even under high stress. However, performance can be further improved by adding a caching layer, such as Redis, to store frequently accessed data. This would reduce the load on the database and enhance response times, particularly under high traffic conditions.

Chapter 7

Conclusions and future improvements

7.1 Results

An emergency management system that can effectively manage users, record occurrences, analyze landcover, and communicate in real time was constructed for this thesis. The system makes use of cutting-edge technologies like Angular and microservices to make sure it functions flawlessly, can support large numbers of users, and is durable. The system is even more effective at handling emergencies due to AI capabilities like land cover analysis and an interactive assistant. Extensive testing was conducted to ensure optimal functionality. All things considered, the project has produced a robust system that helps with the handling of emergencies.

7.2 Future Work

While the current implementation meets the essential requirements, there are several areas for future improvement and expansion:

7.2.1 Enhanced Performance with Caching

To further improve the performance of the map dashboard and other high-load components, implementing a caching layer, such as Redis, can significantly reduce database load and enhance response times.

7.2.2 Advanced AI and Machine Learning Integration

Integrating more advanced AI and machine learning models can improve the accuracy and efficiency of land cover analysis and other predictive functionalities. Custom-trained models tailored to specific emergency scenarios could provide more precise insights.

7.2.3 Expanded Notification and Alert System

Expanding the notification and alert system to include more communication channels, such as SMS and push notifications, can improve the reach and immediacy of critical alerts, ensuring that all relevant parties are promptly informed.

7.2.4 User Experience Enhancements

Further enhancing the user interface to make it more intuitive and accessible, especially under stressful conditions, can improve overall user experience. This includes refining the design, adding more interactive elements, and ensuring compatibility with a broader range of devices.

7.2.5 Geographic Information System (GIS) Integration

Integrating more advanced GIS capabilities can provide better visualization and analysis tools for emergency management. This includes layered maps, spatial data analysis, and integration with additional geographic data sources.

7.2.6 Scalability Improvements

As the user base and data volume grow, further scalability improvements will be necessary. This includes optimizing the microservices architecture, enhancing load balancing strategies, and ensuring the system can scale horizontally to handle increased demand.

7.2.7 Security Measures

Implementing more comprehensive security measures to protect sensitive data and ensure compliance with industry standards is crucial. This includes regular security audits, advanced encryption methods, and robust user authentication mechanisms.

Bibliography

[1] J. Bullock, G. Haddow, and D. Coppola, *Introduction to Emergency Management*. Butterworth-Heinemann, 2017.

[2] J. C. Pine, *Technology and Emergency Management*. John Wiley & Sons, 2017.

[3] D4H Technologies, "D4h: Readiness & response software," 2024, accessed: date-month-year. [Online]. Available: https://www.d4h.com/

[4] ——, "Crisis management software by d4h," 2024, accessed: date-month-year. [Online]. Available: https://www.d4h.com/crisis-management-software

[5] Noggin, "Noggin: Integrated resilience workspace," 2024, accessed: date-month-year. [Online]. Available: https://www.noggin.io/platform

[6] N. A. Ernst and G. C. Murphy, "Case studies in just-in-time requirements analysis," in *2012 Second IEEE International Workshop on Empirical Requirements Engineering (EmpiRE)*, 2012, pp. 25–32.

[7] P. Becker *et al.*, "Applying an improving strategy that embeds functional and non-functional requirements concepts," *Journal of Computer Science & Technology*, vol. 19, 2019.

[8] R. N. Charette, "This car runs on code," *IEEE Spectrum*, vol. 46, no. 3, p. 3, 2009.

[9] M. Abraham, "A guide to standalone applications and why enterprises need them," *Medium*, 2020, accessed: 2024-05-20. [Online]. Available: https://medium.com/swlh/a-guide-to-standalone-applications-and-why-enterprises-need-them-1764fd1f8a0c

[10] H. S. Oluwatosin, "Client-server model," *IOSR Journal of Computer Engineering*, vol. 16, no. 1, pp. 67–71, 2014.

[11] J. Thönes, "Microservices," *IEEE Software*, vol. 32, no. 1, pp. 116–116, 2015.

[12] I. Nadareishvili *et al.*, *Microservice Architecture: Aligning Principles, Practices, and Culture*. O'Reilly Media, Inc., 2016.

[13] T. S. L. Blog, "The history of angular," *Medium*, 2021, accessed: 2024-04-16. [Online]. Available: https://medium.com/the-startup-lab-blog/the-history-of-angular-3e36f7e828c7

[14] "Introduction to asp.net core," https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-8.0, 2024, accessed: 2024-04-16.

[15] "Techempower framework benchmarks," https://www.techempower.com/benchmarks/#hw=ph&test=plaintext&section=data-r22, 2022, accessed: 2024-04-16.

[16] A. Farooq and V. Zaytsev, "There is more than one way to zen your python," in *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering*, 2021.

[17] Node.js, "Overview of blocking vs non-blocking," https://nodejs.org/en/learn/asynchronous-work/overview-of-blocking-vs-non-blocking, 2023, accessed: 2023-06-01.

[18] "Introduction to masstransit," https://masstransit.io/introduction, accessed: 2024-06-05.

[19] Docker, Inc., "Docker documentation: Getting started overview," 2024, accessed: 2024-04-16. [Online]. Available: https://docs.docker.com/get-started/overview/

[20] Docker, "Docker 11-year anniversary," 2024, accessed: 2024-04-16. [Online]. Available: https://www.docker.com/blog/docker-11-year-anniversary/

[21] PostgreSQL Global Development Group, "Postgresql: History," 2024, accessed: 2024-04-16. [Online]. Available: https://www.postgresql.org/docs/current/history.html

[22] Balraj, "Road extraction from satellite images using deeplabv3," https://www.kaggle.com/code/balraj98/road-extraction-from-satellite-images-deeplabv3/notebook, 2020, accessed: June 7, 2024.

[23] V. Agafonkin, "Leaflet - a javascript library for interactive maps," 2024, accessed: 2024-06-01. [Online]. Available: https://leafletjs.com/

[24] Google, "Google oauth 2.0 documentation," 2024, accessed: 2024-06-06. [Online]. Available: https://developers.google.com/identity/protocols/oauth2

[25] OpenAI, "Openai assistants documentation overview," 2024, accessed: 2024-06-06. [Online]. Available: https://platform.openai.com/docs/assistants/overview

[26] Google, "Google maps api documentation," 2024, accessed: 2024-06-06. [Online]. Available: https://developers.google.com/maps/documentation

[27] ——, "Google earth engine api documentation," 2024, accessed: 2024-06-06. [Online]. Available: https://developers.google.com/earth-engine/apidocs

[28] Cloudinary, "Cloudinary documentation," 2024, accessed: 2024-06-06. [Online]. Available: https://cloudinary.com/documentation

[29] Google, "Google gmail imap and smtp documentation," 2024, accessed: 2024-06-06. [Online]. Available: https://developers.google.com/gmail/imap/imap-smtp