Danilo Silva
Guilherme Santos
João Pinto
Pedro Pinto
Tomás Santos

AI-Powered Platform for Smart City Issue Detection & Resolution

Danilo Silva
Guilherme Santos
João Pinto
Pedro Pinto
Tomás Santos

AI-Powered Platform for Smart City Issue Detection & Resolution

Document submitted to the University of Aveiro in fulfillment of the requirements for the approval of the curricular unit Project in Informatics. Carried out under the scientific supervision of Doctor Susana Sargento, Full Professor at the Department of Electronics, Telecommunications and Informatics of the University of Aveiro, and Doctor Pedro Rito, Assistant Professor at the Department of Electronics, Telecommunications and Informatics of the University of Aveiro.

acknowledgements

We would like to thank our supervisors, Professor Doctor Susana Sargento and Professor Doctor Pedro Rito, for all the knowledge they shared with us and for their availability throughout the development of this work. A special thanks also goes to Joaquim Ramos, Marcos Mendes and Gonçalo Perna for their invaluable support in the PIXKIT autonomous vehicle integration in our platform. Finally, we would like to thank the *Instituto de Telecomunicações* and all its staff and collaborators with whom we interacted, for providing all the necessary conditions for the successful execution of this work.

Keywords Artificial Intelligence, Smart City, Urban Incidents, Geospatial Indexing, Large Language Models, Kubernetes Orchestration, Distributed Databases, Asynchronous Processing, Internet of Things, Edge Computing

Abstract This report presents FixAI, an AI-driven platform engineered to optimize urban issue detection and resolution within smart city frameworks. The system features a dual-interface approach: a mobile application for citizen reporting, which leverages AI for faster incident description, categorization, and location acquisition via photo submissions; and a desktop application for municipal operators, facilitating efficient incident management through categorization, filtering, and heatmap visualizations. The foundational architecture of FixAI prioritizes high scalability and reliability, employing Apache Cassandra for structured data storage and MinIO for efficient management of multimedia content such as photos and videos. Kubernetes orchestrates seamless deployment and scaling of these components. A significant innovation lies in the integration of Google's Gemini Large Language Model (LLM), which, in conjunction with an Apache Kafka message broker, enables robust asynchronous processing of AI tasks, including incident detail generation and intelligent clustering of related reports. Furthermore, the platform demonstrates automated issue resolution capabilities through its integration with edge devices, notably exemplified by a proof of concept with the PIXKIT autonomous vehicle, which verifies resolved problems and updates system statuses for municipal operator validation. Future development trajectories include investigating self-hosted LLM solutions and more in-depth computer vision models for enhanced privacy and specialized performance, alongside advancements in H3 geospatial indexing for optimized distributed storage and accelerated spatial queries.

Contents

List of Figures

List of Tables

List of Acronyms

AI *Artificial Intelligence*

API *Application Programming Interface*

ASGI *Asynchronous Server Gateway Interface*

BLOBs *Binary Large Objects*

CD *Continuous Delivery*

CDNS *Content Delivery Networks*

CI *Continuous Integration*

CNCF *Cloud Native Computing Foundation*

CNI *Container Network Interface*

CQL *Cassandra Query Language*

DNS *Domain Name System*

DTOs *Data Transfer Objects*

GCS *Geographic Coordinate System*

GDPR *General Data Protection Regulation*

GHCR *GitHub Container Registry*

GNSS *Global Navigation Satellite System*

GPU *Graphics Processing Unit*

HCD *Human-Centered Design*

HPA *Horizontal Pod Autoscaler*

HTTP *Hypertext Transfer Protocol*

HTTPS *Hyper Text Transfer Protocol Secure*

IoT *Internet of Things*

IPC *Inter-Process Communication*

IPs *Internet Protocol Addresses*

IPVS *IP Virtual Server*

JWT *JSON Web Token*

LLM *Large Language Model*

ML *Machine Learning*

MMLU *Massive Multitask Language Understanding*

MQTT *Message Queuing Telemetry Transport*

MVC *Model-View-Controller*

NFS *Network File System*

NoSQL *Not Only SQL*

OCI *Open Container Initiative*

OLTP *Online Transaction Processing*

OS *Operating System*

PID *Process Identifier*

POC *Proof-of-concept*

PRs *Pull Requests*

PV *Persistent Volume*

PVCs *Persistent VolumeClaims*

RBAC *Role-Based Access Control*

RDBMS *Relational Database Management System*

REST *Representational State Transfer*

RTMP *Real-Time Messaging Protocol*

RTSP *Real Time Streaming Protocol*

S3 *Simple Storage Service*

SDK *Software Development Kit*

SQL *Structured Query Language*

SSL *Secure Sockets Layer*

TCP/IP *Transmission Control Protocol/Internet Protocol*

TLS *Transport Layer Security*

TTL *Time-To-Live*

UI *User Interface*

URIs *Uniform Resource Identifier*

URL *Uniform Resource Locator*

UUIDs *Universally Unique Identifiers*

V2X *Vehicle-to-Everything*

VM *Virtual Machine*

VPA *Vertical Pod Autoscaler*

XSS *Cross-Site Scripting*

Chapter 1

Introduction

1.1 Motivation

The continuous growth and intricate evolution of urban centers present a corresponding escalation in challenges pertaining to infrastructure maintenance, public safety, and the overall quality of urban life. Traditionally, the process by which citizens report urban issues, such as potholes, damaged public infrastructure, or instances of vandalism, has been characterized by its inherent inefficiency and fragmentation. Relying predominantly on conventional channels, including phone calls, emails, or rudimentary municipal applications, these methods frequently prove cumbersome, slow, and unresponsive, often leading to considerable frustration for the **citizens**. Moreover, the diverse origins of complaints and occurrences, spanning social media, emails, and traditional telecommunication, underscore the imperative for a centralized platform to streamline and optimize this entire reporting process. These existing applications, primarily designed **solely for reporting problems**, largely depend on extensive manual data input. This dependence renders the reporting process **bureaucratic** and **arduous** for citizens, despite their genuine interest in contributing to civic improvements, and it highlights significant usability challenges within these systems.

Furthermore, a critical aspect is that these applications were not architected for seamless integration within the growing **smart city** paradigm, where the leveraging of diverse **edge data** sources is paramount. Such integration is critical for bringing a wealth of real-time information into these systems. Access to a vast quantity of varied data sources would enable functionalities such as the real-time detection of nascent urban problems, facilitating the automatic verification of resolved issues, and fostering a truly data-driven approach to urban governance. This paradigm shift is paramount for significantly enhancing urban resilience and operational efficiency. Consequently, the development of a truly "data-driven" application necessitates a robust and scalable architecture capable of supporting vast volumes of data and requests. This inherently requires a system designed for high scalability, built upon a microservices architecture, and incorporating redundancy to ensure resilience against failures.

Additionally, the advent of recent advancements in Artificial Intelligence (AI) technologies presents a transformative opportunity to enhance these urban management systems. AI could significantly assist both citizens and municipal authorities in event reporting, for instance, by automatically populating occurrence details from submitted photographs. Furthermore, these intelligent technologies offer a potent solution to the pervasive problem of data redundancy for city councils. It is a major challenge currently, as multiple citizens often report the same issue, leading to substantial time expenditure for municipal staff. A system dimensioned to intelligently identify and group these redundant reports would considerably streamline the workload for city operators. Moreover, such intelligent systems could dynamically assess the escalating severity of a particular problem based on new citizen reports, automatically updating its status for the city operator. This capability would allow operators to swiftly identify and prioritize urgent issues,

thereby ensuring more timely and effective urban interventions.

1.2 Objectives

The overarching aim of our project is the creation of an intelligent system designed to optimize and enhance the detection and resolution of urban issues. To achieve this primary objective, a series of specific goals have been defined.

A fundamental objective involves the comprehensive study and adaptation of a robust solution for spatial geographic mapping and indexing. This solution will be **crucial** for efficiently representing urban areas, and **must be capable of supporting** complex scenarios such as overlapping jurisdictions or organizations – for instance, enabling both a municipal council and a university within the same city to operate as distinct entities on our platform. Furthermore, the chosen indexing approach should **prioritize efficiency** and seamless **integration** with smart city infrastructures that incorporate diverse and dynamic data sources, including real-time information from moving vehicles.

From a development standpoint, the project's objective is to develop a platform encompassing two main components: a mobile application for citizens and a desktop application for city operators. Firstly, a **mobile application for citizens should be developed** to facilitate effortless incident reporting; users **should be able to** simply capture a photograph, and the application **must automatically process** and populate the incident details in a matter of seconds. This **should include** leveraging Artificial Intelligence (AI) for precise incident description, categorization (e.g., damaged traffic lights, compromised pavement), and severity assessment, while automatically obtaining the location from the device. Users **must also be able to consult** their incident history and track the status of each report (i.e., pending, in progress, or resolved). Its interface **should support** dual representations, allowing users to view incidents and their associated details in both list and map formats. Secondly, a **desktop application for city operators must provide** a comprehensive digital representation of reported urban problems. Operators **must be able to view** incidents grouped by category within a customizable dashboard. The system **must support** various filters (e.g., by problem category or resolution status) and **should consolidate** all occurrences related to a specific incident onto a dedicated page to reduce redundancy and streamline operator workflow. This detailed incident page **must enable** operators to update the incident's status. Analogous to the mobile application, the desktop interface **must offer** an intuitive map view displaying all incidents, with capabilities to filter by category and present incident heatmaps for rapid visual identification of problematic regions.

From an architectural standpoint, a key objective **is to design and implement** a platform capable of handling substantial data volumes, ensuring **high scalability** and **fault tolerance**. This **requires** the selection of an appropriate database solution, particularly for storing user-submitted photographs, which **should leverage** object storage capabilities. Furthermore, this objective also **encompasses** acquiring the necessary expertise and implementing technologies such as Kubernetes to effectively scale and deploy the system on the infrastructure provided by the *Instituto de Telecomunicações*.

Regarding Artificial Intelligence integration, a crucial goal **is to meticulously research and evaluate** various Large Language Model (LLM) solutions, considering both self-hosted and cloud-based options. This evaluation **will involve** a thorough analysis of their respective advantages and disadvantages to select the optimal solution within the project's time constraints. The chosen LLM **must then be seamlessly integrated** with a message broker to facilitate robust asynchronous job processing for AI-driven tasks.

Finally, the project **aims to integrate** data originating from edge devices into the platform. This **should involve** enabling camera-equipped devices to autonomously detect their proximity to a registered urban problem and, through video analysis, ascertain if the problem has been resolved, providing the city operator with video evidence and a suggested resolution status for review and confirmation. Complementing these functional objectives, a strong emphasis **should be placed** on development quality, ensuring adherence to stringent code quality standards across all project

repositories to guarantee long-term maintainability. This **should include** the implementation of a Continuous Delivery (CD) pipeline and the execution of comprehensive system capacity tests to accurately assess the platform's performance.

1.3 Most Relevant Results

The development of the system has yielded a comprehensive set of functional and technical achievements, aligning with all defined objectives and incorporating additional features.

A robust geospatial mapping and indexing solution has been successfully implemented, utilizing the H3 framework, an open-source system developed by Uber. This solution effectively provides an efficient spatial representation, adeptly managing complex scenarios such as overlapping jurisdictions and enabling distinct organizational entities within the same geographic area to operate on the platform. Its design further supports seamless integration with various smart city data sources, including real-time information from moving vehicles.

A dual-interface platform has been successfully developed. The **mobile application for citizens** enables effortless incident reporting through photo capture, with AI capabilities automatically generating incident descriptions, categorizations and severity assessments, while automatically acquiring location data. The application further allows users to consult their incident history and track report statuses through an intuitive interface offering both list and map views. Concurrently, the **desktop application for city operators** provides a comprehensive digital representation of urban problems. Operators can view categorized incidents within a customisable dashboard, apply various filters (e.g., by category, status), and access dedicated incident pages that consolidate all related occurrences, thereby minimizing redundancy. This interface also facilitates status updates and offers an intuitive map view with incident filtering and heatmap visualizations for efficient problem identification.

From an architectural perspective, the platform has been designed and implemented to handle substantial data volumes, demonstrating high scalability and fault tolerance. This includes the successful selection and integration of a high-write database solution (Cassandra) for core data, complemented by an appropriate object storage system (MinIO) for user-submitted photographs. The entire system has been effectively deployed and scaled using Kubernetes on the infrastructure of the *Instituto de Telecomunicações*, showcasing its robust deployment capabilities.

Regarding Artificial Intelligence integration, we selected a Large Language Model (LLM) solution (Gemini, from Google), which has been seamlessly integrated with a message broker to facilitate robust asynchronous job processing for various AI-driven tasks such as description/categorization of the occurrence (Job 1), clustering of related reports (Job 2), and dynamic incident detail updates (Job 3).

A key achievement includes the integration of data from edge devices for automatic issue resolution. A successful proof of concept was realized using the PIXKIT autonomous vehicle from the *Instituto de Telecomunicações*. This demonstration showcased the system's ability to integrate with smart city infrastructure, where the vehicle, equipped with sensors and a camera, could detect when it passed by a previously reported and resolved urban problem. Upon detection, the system automatically provides the city operator a video evidence and a suggested "resolved" status for confirmation. This autonomous vehicle was chosen for the proof of concept due to the belief that future vehicles will widely incorporate cameras, making them pervasive sources of real-time data for such platforms.

In addition to fulfilling the primary objectives, several important features and practices were implemented. These include comprehensive **support for multiple languages**, implementation of robust **session management with refresh tokens** for enhanced security, establishment of a reliable **continuous delivery (CD) pipeline** for streamlined deployments, and comprehensive **secure account management**. Furthermore, the mobile application provides a **report anywhere** capability, allowing users the flexibility to report issues either directly on-site or from their homes. Overall, adherence to stringent code quality standards throughout all project repositories ensures long-term maintainability, further solidifying the system's foundation.

1.4 Document Organization

This document is structured into six distinct chapters, each designed to provide a comprehensive understanding of our project, from its foundational concepts to its implementation, validation, and future prospects.

As described before, this chapter provides an overarching view of the work, detailing the motivations and context that underpinned the project's inception. Furthermore, it explicitly defines the objectives that guided its execution and highlights the most relevant results achieved throughout the study.

Chapter 2, Preliminaries, serves to establish the foundational knowledge base. It commences with an analysis of existing *Related Work* in the domain of urban issue-reporting platforms, identifying current solutions and their limitations. Subsequently, it delves into essential *Background Concepts* crucial for understanding our project's architecture and functionalities, encompassing topics such as web frameworks, database solutions, image and object storage, container orchestration, geospatial indexing, and the application of Large Language Models (LLMs).

Chapter 3, Product and Vision Concept, articulates the core idea behind our project. It presents the project's *Vision Statement* and a comprehensive *Product Concept Overview*. A significant portion is dedicated to the *User-Centered Design (HCD)* methodology employed, detailing the identified user profiles, their experiences through scenarios, defining their needs via user stories, and outlining the specific use cases for both the mobile and desktop applications. The chapter concludes by enumerating the critical *Non-Functional Requirements* that govern the system's quality attributes.

Chapter 4, Architecture Notebook, provides a deep dive into the system's structural design. It offers an *Architecture Overview*, complete with various diagrams illustrating the system's architecture, deployment, and data access patterns. This chapter also elaborates on the chosen *Technology Stack* that underpins our platform.

Chapter 5, Implementation, constitutes the technical core of the report, detailing the practical realization of our project. It describes the development of the *Frontend Applications* (Mobile and Desktop) and thoroughly explains the *Backend Services*, covering API design, database integration (for both structured and object data), geospatial operations using H3, and a detailed exposition of the *Asynchronous Job Processing* mechanisms, including those leveraging LLMs for various tasks. Furthermore, it explains the *Issue Automatic Resolution* components, outlines the *Infrastructure and Deployment* strategies for our distributed system components, and summarizes the *Quality Assurance* practices.

Finally, **Chapter 6, Conclusion and Future Work**, summarizes the main *Conclusions* drawn from the entire body of work. It also outlines promising *Future Work Directions*, suggesting potential improvements for LLM integration, geospatial optimization, and enhanced security measures, paving the way for the continued evolution of our project.

Chapter 2

Preliminaries

2.1 Related Work

This section provides a critical analysis of existing solutions within the domain of urban issue-reporting platforms, identifying their current capabilities and inherent limitations. It further highlights areas where the integrated use of artificial intelligence can significantly enhance efficiency and expedite the reporting process for citizens.

Municipal Citizen Reporting Applications: Na Minha Rua Lx, @Coimbra, and ReportaPorto. Common among Portuguese municipalities, applications such as *Na Minha Rua Lx* (Lisbon), *@Coimbra* (Coimbra), and *ReportaPorto* (Porto), present in Figure 2.1, serve as primary digital channels for citizens to communicate urban issues to local authorities [Lis25], [Coi25], [Por25]. While these platforms successfully establish a direct line of communication, their functionalities often present significant operational friction. A pervasive limitation observed across these applications is the **absence of automatic geolocation detection**, requiring users to manually input addresses or precisely pinpoint locations on a map, which frequently leads to data inaccuracies and an increased reporting effort for the citizen. Furthermore, the **lack of intelligent categorization** necessitates manual selection of the problem type from a predefined list, a process that is significantly time-consuming. Critically, users on these platforms are required to manually compose textual accounts of the issues, a process that could be significantly improved if AI-generated descriptions were offered. This suite of manual requirements collectively renders the reporting process more time-consuming and cumbersome, diminishing user engagement and discouraging the submission of valuable reports.

Figure 2.1: Portuguese municipal citizen reporting applications: Na Minha Rua Lx, @Coimbra, and ReportaPorto.

Government Management Platforms: GovPilot. *GovPilot*, depicted in Figure 2.2, is a comprehensive government management software suite widely utilized in the United States, offering modules for both citizen issue reporting and municipal operations management [Gov25b]. While providing a robust framework for municipal operations, its approach to incident reporting reveals certain areas for enhancement concerning user efficiency. Notably, the submission process

for citizens often proves time-consuming due to the requirement for users to **complete lengthy forms**, meticulously **select problem categories and subcategories**, and **manually specify the exact location** of the incident. These manual steps contribute to a less streamlined user experience. Beyond these specific aspects of incident reporting, GovPilot functions as a widely adopted application by various states for broader governmental management.

Figure 2.2: GovPilot - Modern Local Government Management Software.

National Citizen Interaction Platforms: A Minha Rua (gov.pt). *A Minha Rua*, a service integrated within the official Portuguese government portal (gov.pt), present in Figure 2.3, aims to centralize citizen interactions regarding various public services [Gov25a]. Despite its national scope and foundational role in digital governance, its application for reporting urban incidents shares several functional and usability drawbacks. These include, but are not limited to, the **absence of automatic location detection** and the **reliance on manual selection for problem categorization**.

Figure 2.3: *A Minha Rua* (gov.pt) platform.

Crucially, due to its national scope, *A Minha Rua* faces the challenge of **non-adhering municipalities**. In such instances, citizens residing in these areas are compelled to utilize alternative problem reporting systems provided by their local authority or contact them directly, fragmenting the reporting ecosystem. This national centralisation presents both advantages and disadvantages. On one hand, it offers the benefit of unifying the reporting process, providing citizens with a single, known platform for submitting issues across the country. On the other hand, this broad scope inherently means the platform may not be optimally tailored to the specific operational nuances and unique requirements of individual municipalities. This lack of municipal-specific optimisation can lead to the omission of particular functionalities or refinements that could significantly enhance user satisfaction and operational efficiency at a local level.

In conclusion, while current urban issue-reporting platforms effectively establish channels for citizen participation, they largely remain reliant on manual human input. With the growing trend towards smart cities and the increasing availability of diverse data sources, particularly from edge devices, it becomes imperative for these systems to be designed from their inception to facilitate seamless integration. Such integration would enable the real-time detection of new urban problems, the automatic verification of resolved issues, and a more dynamic and data-driven approach to urban governance. This transition from purely human-driven reporting to an intelligent, sensor-informed framework represents the next crucial step in enhancing urban resilience and efficiency.

2.2 Background concepts

2.2.1 Web Model

The architecture of user-facing applications has evolved considerably over the years to address increasing demands for **scalability**, **maintainability** and **performance**. Among the foundational architectural patterns that have shaped modern software design is the **Model-View-**

Controller (MVC) paradigm. MVC promotes a clear separation of concerns by organizing applications into three distinct layers: the **Model**, the **View** and the **Controller** [Gam+94]. The **Model** layer encapsulates the application's data and business logic, managing state transitions and ensuring data integrity. It is responsible for interfacing with data storage systems, processing user actions and applying the business rules that govern application behavior. The **View** layer, in contrast, is concerned with rendering the user interface based on the current state of the Model. It provides the mechanisms by which information is visually communicated to the user, ensuring clarity and responsiveness. The **Controller** serves as the intermediary that handles user interactions, interpreting user inputs to manipulate the Model and triggering updates to the View. This dynamic interaction loop enables a responsive and intuitive user experience while maintaining a clean and modular architecture, as illustrated in Figure 2.4.

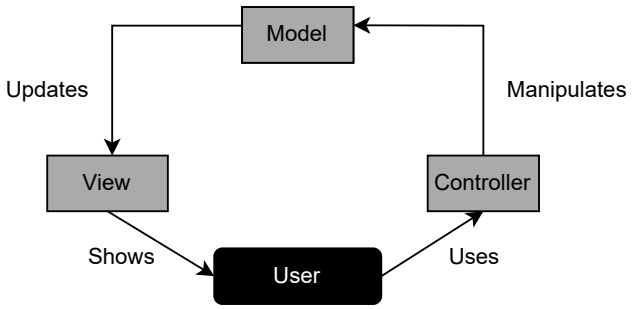


Figure 2.4: Schematic representation of the Model-View-Controller (MVC) architectural pattern.

To support **interoperability** between distributed components, modern web applications frequently rely on Application Programming Interfaces (APIs). An API defines a standardized set of operations that enable **communication between client and server** systems. One of the most widely adopted architectural styles for API design is Representational State Transfer (REST), which utilizes standard HTTP methods, such as `GET`, `POST`, `PUT` and `DELETE`, to manipulate resources identified by Uniform Resource Identifiers (URIs). RESTful APIs are characterized by their statelessness, resource orientation and uniform interface, making them particularly suitable for scalable and distributed web services.

While REST has become a **de facto standard** for web API development, alternative communication protocols offer distinct advantages in specific scenarios. Remote Procedure Call (gRPC), developed by Google, utilizes Protocol Buffers (protobuf) for efficient binary serialization, providing high-performance communication with low latency. It is particularly well-suited for microservices architectures, where inter-service communication efficiency is critical [Goo24b]. **WebSocket** technology enables full-duplex, real-time communication over a single, persistent connection, supporting applications that require immediate bidirectional data exchange, such as live chat systems, real-time notifications and interactive dashboards. Message Queuing Telemetry Transport (MQTT), on the other hand, is a lightweight publish/subscribe protocol specifically designed for Internet of Things (IoT) applications. Its minimal overhead and robust delivery mechanisms make it ideal for resource-constrained devices operating in unreliable network conditions [MQT24].

These architectural principles and communication protocols collectively provide the foundation for building robust, scalable and **user-centric web applications** capable of meeting the diverse requirements of modern digital ecosystems.

2.2.2 FastAPI Web Framework

Developing a backend service capable of supporting scalable, high-performance interactions over HTTP requires the implementation of a **REST API**. Among the various web frameworks

available for this purpose, **FastAPI** has gained considerable recognition for its effectiveness, particularly in scenarios that demand **asynchronous execution**, **high concurrency** and **low latency** [Fas25].

One of FastAPI's most significant **advantages** lies in its foundation on the Python programming language, recognized for its simplicity and readability. While Python is typically considered less performant than compiled languages, FastAPI mitigates these limitations through native support for asynchronous programming. By leveraging Python's **async** and **await syntax**, FastAPI enables non-blocking I/O operations, allowing the server to handle multiple concurrent client requests efficiently. This capability significantly reduces response times and runtime overhead, making FastAPI particularly suitable for high-throughput applications such as real-time data processing and distributed microservices, where scalability and responsiveness are essential. FastAPI applications are typically deployed using **Uvicorn**, a high-performance **Asynchronous Server Gateway Interface (ASGI)** server [Uvi25]. Uvicorn's lightweight architecture and compatibility with asynchronous frameworks enable FastAPI to deliver **low-latency** and **high-concurrency** performance. Independent performance benchmarks, such as those conducted by TechEmpower, have consistently ranked FastAPI, when deployed with Uvicorn, among the **highest-performing Python web frameworks** [Fas25],[Tec25]. Another notable strength of FastAPI is its native support for the **OpenAPI** standard (formerly known as Swagger) [Ope25]. FastAPI automatically generates comprehensive API documentation, which not only reduces development effort but also enhances maintainability by providing a clear and standardized description of the available API endpoints. This feature benefits both developers and stakeholders by simplifying system understanding and integration.

Despite these advantages, FastAPI presents certain **limitations** when compared to more mature enterprise-grade frameworks such as **Spring** [Spr25]. Its ecosystem remains relatively smaller and less mature, offering fewer enterprise-level plugins, libraries and extensions. Furthermore, FastAPI lacks out-of-the-box support for several critical features, including authentication, role-based access control and background job management. These functionalities must be implemented manually or integrated through external solutions, potentially increasing the complexity of development and maintenance.

2.2.3 Cassandra Database

Apache Cassandra is an open-source, distributed NoSQL database designed to handle large volumes of data with high availability, fault tolerance and no single point of failure. It is especially well-suited for applications requiring scalability, continuous uptime and efficient management of big data across multiple servers or data centers. This database employs a **peer-to-peer**, **ring-based architecture** without a master node, as described in Figure 2.5, ensuring equal responsibility among all nodes, eliminating single points of failure and allowing the system to remain operational even if entire data centers go offline [LM10]. Nodes in the ring communicate status and membership changes using the **gossip protocol**, a decentralized peer-to-peer communication method where nodes periodically exchange information about themselves and other nodes they know about. This ensures that all nodes in the cluster maintain an up-to-date view of the cluster's state. Data is automatically replicated across multiple nodes and data centers, providing robust fault tolerance and continuous availability [LM10]. Cassandra **scales horizontally** by adding more servers, maintaining performance and reliability as data volume grows. It is designed to handle massive amounts of structured data efficiently and for large datasets, Cassandra often outperforms traditional relational databases, especially as data size increases, offering faster query response times and higher throughput [ASS18].

2.2.3.1 Column-Oriented Storage Model

Unlike traditional relational databases that store data in rows, Cassandra employs a **column-oriented storage model**.

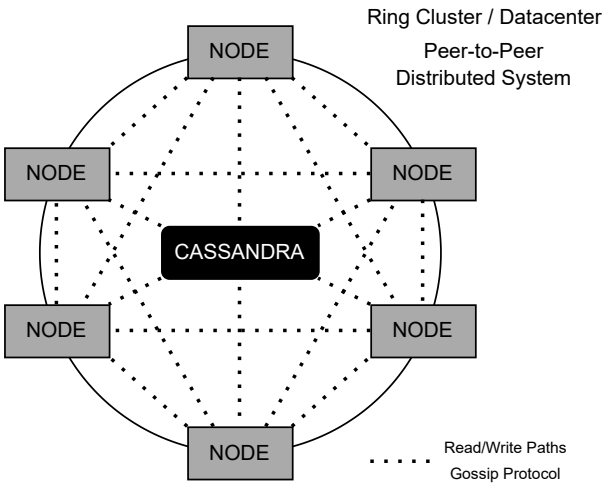


Figure 2.5: Cassandra Ring Topology and Data Distribution.

In a **row-oriented** database, data for each record is stored together sequentially. For the following example data, it would be organized as follows:

```
[UserId, Name, Age, Country]
[1, Alice, 25, Portugal]
[2, Bob, 30, France]
[3, Charlie, 28, Spain]
```

Figure 2.6: Row-oriented Storage.

Each entry represents a complete row containing the UserID, Name, Age and Country for an individual.

In contrast, a **column-oriented** database like Cassandra stores all the values for a single column together. The same example data would be conceptually stored as:

```
UserID:    [1, 2, 3]
Name:      [Alice, Bob, Charlie]
Age:       [25, 30, 28]
Country:   [Portugal, France, Spain]
```

Figure 2.7: Column-oriented Storage.

Here, all the UserIDs are stored contiguously, followed by all the Names, then all the Ages and finally all the Countries. This difference in storage layout offers significant advantages for certain types of queries and workloads. For example, when a query only needs a subset of columns (e.g., just the 'Name' and 'Country'), Cassandra only reads the relevant data from disk, leading to **improved read performance** and **reduced I/O** compared to row-oriented databases that would read entire rows for each matching record. Furthermore, the similarity of data within a column (e.g., all ages are integers, all countries are strings) often allows for more effective compression, resulting in storage savings. While Cassandra does have schemas, they are more dynamic than in relational databases, permitting the addition of new columns to a "column family" (similar to a table) without affecting existing rows, a flexibility well-suited for evolving data structures [LM10].

The fundamental unit of data in Cassandra is a cell, a tuple comprising a **column name**, a **value** and a **timestamp**. Rows are collections of these cells, uniquely identified by a **primary key**. Column families, or tables in the Cassandra Query Language (CQL), are then collections of these rows. Although CQL provides a tabular abstraction for user interaction, the underlying storage mechanism remains fundamentally column-oriented [LM10]. To achieve scalability and high availability, Cassandra distributes data across nodes based on the **partition key** (first part of the primary key) using **consistent hashing**. This process assigns data to specific token ranges on a logical ring of nodes. Replication, configured per **keyspace**, ensures fault tolerance by storing copies of the data on multiple nodes. Within a partition, rows are often ordered by the optional **clustering key** (the second part of the primary key). When querying, if the full partition key is provided, Cassandra efficiently routes the request to the responsible node(s). The strategic choice of the partition key is critical for balanced data distribution and optimal performance [LM10]. The Table 2.1 shows some comparisons with the traditional RDBMS.

Table 2.1: Comparison of Cassandra with Traditional RDBMS.

Feature	**Cassandra**	**Traditional RDBMS**
Data Model	NoSQL, wide-column, flexible	Relational, fixed
Scalability	Horizontal, easy to add nodes	Often vertical
Availability	High, no single point of failure	Varies
Query Language	CQL (limited joins/views)	SQL (rich joins)

2.2.3.2 Typical Applications

Building upon its core features of high scalability, availability and good write performance, Cassandra's architecture and data model render it an ideal choice for a diverse range of demanding applications. For instance, its efficient handling of write-intensive operations and its capacity to effectively query data based on temporal ranges make it particularly well-suited for managing and analyzing **time-series data** originating from sources such as sensors, financial markets, or application logs [CH10]. The growing field of the **Internet of Things (IoT)**, with its characteristically massive data volumes generated by numerous connected devices, also finds a robust solution in Cassandra's scalable and fault-tolerant nature, enabling real-time data ingestion and processing.

Furthermore, **social media platforms** and **web-scale and cloud applications**, such as Facebook, which grapple with high write throughput demands and need to manage vast quantities of concurrent user interactions and data points like feeds, activity streams and user profiles, requiring high availability, zero downtime and the ability to scale horizontally across global data centers, significantly benefit from Cassandra's capabilities [CH10]. Even **e-commerce platforms**, where managing extensive product catalogs, order histories and user sessions with high availability is paramount, can consider Cassandra as a viable option for specific aspects of their infrastructure due to its decentralized, peer-to-peer architecture ensuring no single point of failure. Cassandra also supports applications for **real-time analytics and online transaction processing (OLTP)** by providing the ability to handle high-velocity data streams [CKL15].

Ultimately, while Cassandra offers compelling advantages for these types of applications, it remains crucial to carefully evaluate the specific requirements of a given use case, particularly concerning the complexity of required queries and the stringency of consistency guarantees, to determine if it represents the most appropriate database solution.

2.2.4 Image and Video Storage Solutions

Modern applications often rely on the ability to store and retrieve multimedia content efficiently, especially images and videos. Choosing the appropriate storage strategy is essential for ensuring scalability, performance and maintainability. In this section, we explore the main models

available for media storage, with special attention to object storage due to its flexibility, scalability and strong alignment with cloud-native architectures.

2.2.4.1 Storage Models Overview

Several architectural approaches exist for handling media files, each offering distinct benefits and trade-offs depending on the application requirements and infrastructure constraints. The three most common storage models are **Database Storage**, where media files are stored as binary data (BLOBs) within relational databases; **File System Storage**, which organises files using hierarchical directory structures on physical or virtual disks; and **Object Storage**, which adopts a flat and highly scalable model based on buckets and uniquely identifiable objects.

We provide a detailed explanation of each model below, highlighting the advantages and disadvantages of each, with a particular emphasis on object storage due to its relevance to our system architecture.

Database Storage (BLOBs). In this model, binary files such as images and videos are stored directly inside a relational database using a special data type called `BLOB` (Binary Large Object). The media files are stored alongside metadata in a structured schema, which can simplify consistency and integrity between the data and the media. This approach offers advantages like atomic transactions and centralised access control through SQL. However, database storage presents performance and scalability limitations, especially for large media files or high-throughput access. Databases are optimized for structured data and storing large binary files can lead to increased database size, slower backups and complex maintenance operations.

File System Storage. This traditional approach uses the operating system's hierarchical directory structure to organize and store media files on disk. Each file is saved as a discrete entity and accessed via a file path. Metadata is typically stored separately, either in a database or in sidecar files. This model provides fast local access and is relatively simple to implement. It scales moderately well in controlled environments, such as within enterprise infrastructure or hosted servers. However, managing access control, redundancy and metadata can be cumbersome at scale, especially in distributed systems. Moreover, network file systems (e.g., NFS, SMB) can introduce bottlenecks under heavy loads.

Figure 2.8 illustrates the hierarchical structure typical of file system storage, where folders group files into directories and subdirectories, mimicking a tree-like layout. This visualization highlights the simplicity and organization of file-based systems, but also hints at their limitations in terms of flat scalability and distributed access.

Object Storage (Bucket Storage). Object storage is a modern, highly scalable solution designed specifically for unstructured data such as images, videos and backups. Unlike the file system model, object storage does not use a hierarchical structure. Instead, data is stored as objects in a flat namespace, often organized into logical containers called *buckets*.

Each object consists of:

- **Data**: the actual binary content of the file.
- **Metadata**: customizable key-value pairs associated with the object (e.g., content type, owner, creation timestamp).
- **Unique Identifier**: a globally unique ID or key used to retrieve the object.

Objects are accessed through HTTP-based REST APIs, typically using endpoints that follow the pattern `https://<endpoint>/<bucket>/<object>`. This makes **object storage ideal for cloud environments**, where it integrates seamlessly with web applications, content delivery networks (CDNs) and microservices. One of the key advantages of object storage is its near-infinite scalability. Systems like **Amazon S3**, Google Cloud Storage and MinIO are designed to scale

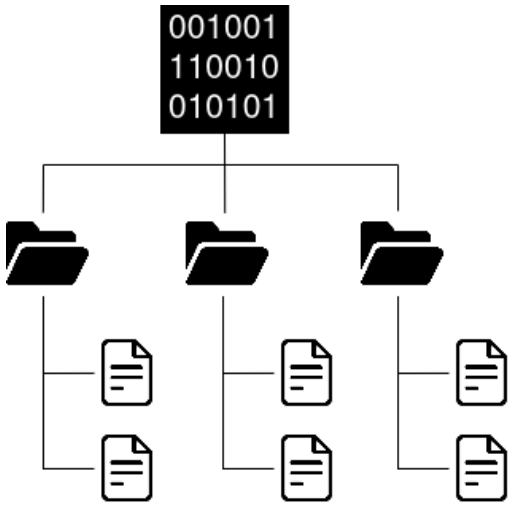


Figure 2.8: File Storage Hierarchical Structure.

horizontally by adding storage nodes without impacting availability. Features such as automatic replication, versioning and lifecycle policies enhance durability and reduce administrative overhead.

Furthermore, object storage is optimized for durability and cost efficiency. Most systems use distributed erasure coding and replicate data across multiple locations to ensure resilience. For example, Amazon S3 is architected to provide exceptional durability, with a design target of 99.999999999% (commonly referred to as "eleven nines") of data durability. By default, S3 stores data redundantly across at least three Availability Zones, offering built-in protection against the failure of an entire data center [Ama25]. Due to its flexibility, metadata support and seamless integration with modern development stacks, object storage is increasingly the preferred solution for storing large volumes of multimedia content in both enterprise and cloud-native applications.

2.2.4.2 Comparative Analysis: Object vs File vs Database Storage

To evaluate which storage model is best suited for image and video storage, it is crucial to analyze them along key operational and architectural dimensions. This comparison, summarized in Table 2.2, focuses on six essential metrics: scalability, performance, data structure, ease of access, cost efficiency and durability.

In summary, while database and file system storage models may be suitable for small-scale or on-premise use cases, they quickly become inadequate as data volume, access requirements, or availability expectations grow. Object storage provides a robust and scalable solution tailored to **modern cloud-native workloads**, making it the preferred model for storing media assets in distributed systems.

2.2.4.3 MinIO vs. AWS S3

MinIO. MinIO is a high-performance, open-source object storage solution designed to be compatible with the Amazon S3 API. It can be deployed on-premises or in private clouds and is particularly suitable for edge computing, hybrid cloud environments and developers seeking full control over their infrastructure. MinIO supports key S3-compatible operations and integrates seamlessly with tools and libraries designed for AWS S3, making it an attractive lightweight alternative for local or private deployments [Min25].

Amazon S3. AWS S3 (Simple Storage Service), on the other hand, is a fully managed object storage service provided by Amazon Web Services (AWS). It offers unmatched scalability, availability and durability. With multiple storage classes designed for varying access patterns and cost

Table 2.2: Comparison of Storage Models.

Metric	Database Storage	File System Storage	Object Storage
Scalability	Limited; databases are not optimized for large binary files	Moderate; scales with filesystem and disk management	High; designed for massive horizontal scaling in distributed environments
Performance	Slower for large media files; query overhead impacts speed	Fast for local access, but limited in distributed setups	High; optimized for large unstructured data and parallel access
Data Structure	Structured (BLOBs inside relational schemas)	Hierarchical (folders and paths)	Flat namespace with metadata and unique identifiers
Ease of Access	Requires SQL queries; not ideal for direct media serving	Simple for local access; harder for remote/distributed access	Accessible via REST APIs or SDKs; ideal for web/cloud environments
Cost Efficiency	Expensive for large volumes; storage not optimized	Moderate; depends on hardware and maintenance	Very high; supports tiered storage classes for access frequency
Durability	High with proper replication, but not built-in	Depends on backup/RAID configuration	Very high; built-in redundancy (e.g., 11 nines in AWS S3)

optimization, S3 serves as the backbone for cloud-native applications requiring long-term, resilient storage [Ama25].

Despite the difference in their deployment models and licensing (MinIO being open-source and AWS S3 being a commercial service), both systems share a high degree of compatibility through the S3 API. This allows developers to prototype or deploy solutions using MinIO and later migrate to AWS S3 with minimal changes to the application codebase. Such compatibility is particularly advantageous in agile development environments or when cloud migration is planned for the future.

2.2.5 Kubernetes

Kubernetes is an open-source system for automating deployment, scaling and management of containerized applications. It adopts a declarative, **control-plane/data-plane architecture** in which the control plane maintains the desired cluster state and the data plane executes workloads on the nodes [HBB17]. In the following subsections, we detail the core workload controllers, networking abstractions, persistent storage mechanisms, cluster architecture, scalability features and the lightweight K3s distribution.

2.2.5.1 Core Workload Controllers

Kubernetes orchestrates application workloads by defining and monitoring declarative resource objects. The primary workload controllers, **Pods**, **Deployments** and **StatefulSets**, form the foundation for running both stateless and stateful services, enabling consistent scaling, self-healing and ordered deployment behaviors. In the following paragraphs, we introduce each of these controllers and their roles within the cluster.

Pods. A **Pod** is the smallest deployable unit, encapsulating one or more co-located containers that share the same Linux namespaces (network, IPC, PID) and storage volumes, all scheduled

Table 2.3: Kubernetes Service Types.

Type	Description and Use Case
`ClusterIP`	Default; exposes the Service on an internal IP in the cluster. Used for inter-Pod communication.
`NodePort`	Exposes the Service on the same port of each Node's IP. Useful for simple external access without a load balancer.
`LoadBalancer`	Provisions an external load balancer (cloud providers) to forward to NodePorts. Ideal for production services requiring L4 routing.
`ExternalName`	Maps the Service to a DNS name (no proxy). Enables accessing external resources via a Service abstraction.

and managed as a single entity [Kub25p]. They are intrinsically ephemeral: when evicted or deleted, new Pods receive distinct IPs and are created from scratch.

Deployments. A **Deployment** manages stateless applications by maintaining a desired number of identical Pod replicas and providing declarative updates to the Pod template [Kub25c]. It handles **replication** (scaling up/down), **self-healing** (replacing failed Pods) and **ensures that all Pods run the specified container image version**, making it ideal for microservices and front-end workloads where no per-Pod persistent identity is required.

StatefulSets. **StatefulSets** manage stateful applications by assigning each Pod a unique, stable network identity (via ordinal indices) and persisting storage volumes across restarts. Pods are created and terminated in strict ordinal order, guaranteeing ordered initialization (e.g., clustered databases) and safe scaling operations [Kub25r].

2.2.5.2 Networking and Service Discovery

Services. In Kubernetes, a **Service** is a stable abstraction that defines a logical set of Pods and a policy by which to access them. Services decouple clients from Pod IPs, which can change over time, by providing a consistent virtual IP address and DNS name. Depending on the networking requirements and the desired exposure level, Kubernetes supports several Service types, each suited to different use cases. Table 2.3 summarizes the **four primary Service types** and their typical applications [Kub25q].

Headless Services. By setting `clusterIP: None`, Headless Services disable virtual IP allocation and instead return individual Pod IPs via DNS, enabling client-driven load balancing or peer discovery for StatefulSets [Kub25d].

Ingress. An **Ingress** is a Kubernetes **API object** that defines Layer 7 (HTTP/HTTPS) routing rules, mapping incoming requests (by host and path) to Services within the cluster, thereby centralizing traffic management at the edge of the platform [Kub25g]. An **Ingress Controller** (e.g., NGINX, Traefik) watches for these resources and configures a reverse proxy to implement the specified rules [Kub25h].

2.2.5.3 Persistent Storage

PersistentVolumes and PersistentVolumeClaims. **PersistentVolumes (PVs)** represent physical storage resources (block, file) with defined **capacity**, **access modes** (e.g., `ReadWriteOnce`, `ReadWriteMany`) and **reclaim policies**. **PersistentVolumeClaims (PVCs)** are user-defined requests that bind to matching PVs or trigger dynamic provisioning via the referenced **StorageClass** [Kub25o].

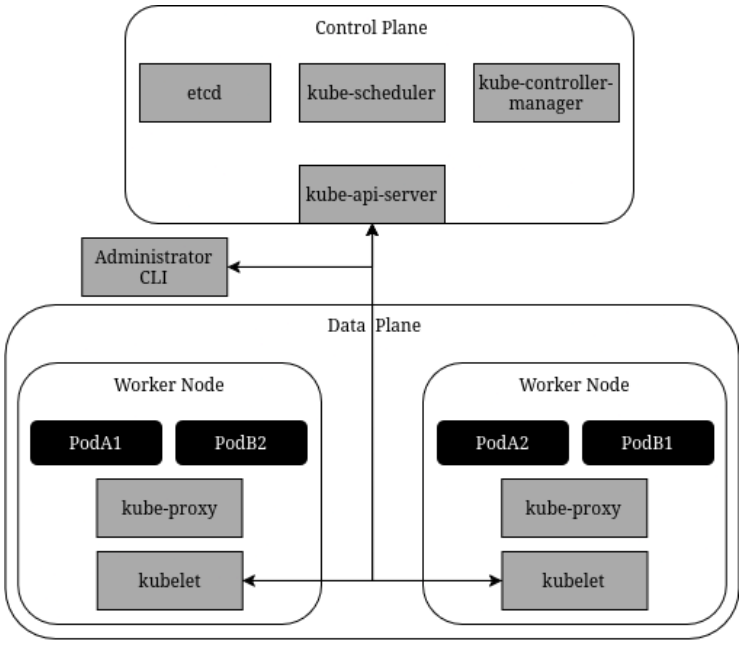


Figure 2.9: Kubernetes Cluster Architecture.

StorageClasses. **StorageClasses** abstract dynamic volume provisioning by specifying a provisioner plugin (e.g., AWS EBS, NFS CSI), parameters (e.g., `reclaimPolicy`, IOPS) and binding mode. Users reference a StorageClass in a PVC to request particular performance or durability characteristics without managing the underlying infrastructure [Kub25s].

NFS Server. An **NFS (Network File System) Server** exports directories over TCP/IP, enabling multiple clients to mount shared file systems. In Kubernetes contexts, an external NFS server can run on bare-metal or a VM, exporting one or more shares (e.g., `/mnt/nfs`) for Pods to mount.

NFS Subdir External Provisioner. The CSI `nfs-subdir-external-provisioner` automates PV creation on an existing NFS export by allocating per-PVC subdirectories. This removes the need for cluster administrators to pre-create NFS shares, enabling seamless network-file-system storage for stateful workloads [Kub25n; Kub25a].

2.2.5.4 Cluster Architecture

In Kubernetes, the cluster architecture is logically divided into two planes: the ***control plane***, which makes global scheduling and state-management decisions and the ***data plane***, which runs the actual workload on each node, as shown in Figure 2.9.

Control Plane. The **control plane** is responsible for maintaining the desired state of the cluster, **scheduling Pods**, **responding to events** (e.g. bringing up replacement Pods when replicas fail) and **exposing the API surface** that users and controllers interact with. It is mainly composed by the items summarized in Table 2.4 and, by default, the node hosting these components (often called the "master" node) is tainted to prevent regular Pods from being scheduled there, however, administrators may remove this taint if they wish to run application Pods on the control-plane node itself.

Table 2.4: Kubernetes Control Plane Components.

Component	Description
`kube-apiserver`	Exposes the Kubernetes API over HTTPS, handles authentication, authorization, admission control, and persists all resource definitions to `etcd` [Kub25i].
`etcd`	A distributed, strongly consistent key–value store that holds the entire cluster state, provides watch APIs for real-time synchronization, and enables failover via clustering [etc24].
`kube-scheduler`	Watches for unscheduled Pods and assigns them to nodes based on resource requests, constraints (affinity/anti-affinity, taints/-tolerations), and scheduling policies to optimize resource utilization [Kub25k].
`kube-controller-manager`	Runs various controllers (e.g., Node, Deployment, StatefulSet) in a single binary. Each controller reconciles the actual cluster state with the desired state as specified in the API [Kub25m].

Table 2.5: Kubernetes Data Plane Components.

Component	Description
`kubelet`	Node agent that registers the node with the API server, watches for Pod specification objects, and ensures containers described in Pods are launched, remain healthy, and conform to resource constraints [Kub25l].
`kube-proxy`	Manages network rules on each node (using iptables or IPVS) to implement Services, load-balance traffic, and maintain virtual IPs for internal clients [Kub25j].
Container runtime	Implements the OCI runtime interface to pull, unpack, start, and manage container images (e.g., `containerd` or Docker). `containerd` is preferred for its stability, performance isolation, and minimal footprint [Kub25b].

Data Plane. Each worker node in the data plane hosts the local agents and runtimes that execute Pods, enforce networking and report status back to the control plane. This can only be possible with the components listed in Table 2.5.

2.2.5.5 Scalability and High Availability

Horizontal Pod Autoscaling The **Horizontal Pod Autoscaler (HPA)** adjusts the replica count of Deployments or StatefulSets based on observed metrics (CPU, memory, custom metrics), enabling workloads to scale in response to demand [Kub25f].

Vertical Pod Autoscaling. Vertical Pod Autoscaler (VPA) recommends or enforces adjustments to Pod resource requests and limits at runtime, optimizing cluster utilization and workload performance [Kub25t].

Control Plane High Availability. Control-plane components can be deployed in **multi-master mode**, with `etcd` clustered across nodes and API servers behind a virtual IP or load-balancer for failover, ensuring continuous scheduler and controller operation during node failures [Kub25e].

2.2.5.6 K3s: Lightweight Kubernetes Distribution

K3s is a CNCF–certified, single-binary distribution that bundles the control plane and essential services (Flannel CNI, CoreDNS, sqlite3 by default) into a <100 MB package. It uses a simplified datastore (embedded sqlite3 or external etcd/MySQL) and reduced dependencies, making it **ideal for edge computing**, **IoT devices**, **CI/CD pipelines** and **resource-constrained environments** while retaining full API compatibility with upstream Kubernetes [Ran25].

2.2.6 Geographic Coordinate System

One of the most common ways to represent a location on the Earth's surface is through the *Geographic Coordinate System* (GCS). This system uses latitude and longitude to define positions on a spherical model of the Earth. Latitude (ϕ) represents the angular distance from the Equator, while longitude (λ) represents the angular distance from the Prime Meridian.

This model is practical and widely used in various applications such as navigation, mapping services and data collection, especially when the points of interest are independent and do not require relational analysis with other locations. It provides a simple way to store and share locations globally, making it the standard in many systems.

Several algorithms have been proposed for this representation. One widely adopted method is the *Ray Casting Algorithm*, described in the following section as a foundational approach for point-in-polygon analysis.

2.2.6.1 Ray Casting Algorithm

The *Ray Casting Algorithm* is a classical computational geometry technique employed to solve the *point-in-polygon* problem. Given a point and a polygon defined by an ordered sequence of vertices, the algorithm determines whether the point lies inside or outside the polygon by projecting an imaginary ray from the point and counting the number of intersections between the ray and the polygon's edges.

Formal Definition. Let $P(x, y)$ be a point in $\mathbb{R}^2$ and $\mathcal{P} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ a simple polygon defined by n sequentially connected vertices. The algorithm proceeds as follows:

1. Cast a semi-infinite ray from P in a fixed direction (typically along the positive x-axis).
2. Count the number of intersections between the ray and the polygon's edges.
3. If the number of intersections is **odd**, classify P as *inside* the polygon; otherwise, classify P as *outside*.

Edge Cases. Special care must be taken when the point P lies exactly on an edge or vertex of $\mathcal{P}$. Such conditions require numerical tolerance handling or explicitly defined boundary rules to ensure stable and deterministic behavior.

Computational Complexity. The computational cost for processing a single polygon is $\mathcal{O}(n)$, where n is the number of polygon edges. When extended to a collection of m polygons $\{\mathcal{P}_1, \ldots, \mathcal{P}_m\}$, the overall complexity becomes $\mathcal{O}(m \times n)$, assuming uniform polygon sizes. This linear scaling arises because the algorithm requires evaluating every edge of every polygon independently.

2.2.7 H3 Index

H3 is a geospatial indexing system developed by Uber Technologies to efficiently partition the Earth's surface into a **hierarchical grid of hexagonal cells** [Eng] for geospatial data visualization and analysis. Unlike traditional coordinate-based representations, H3 enables fast spatial

Figure 2.10: Hierarchical indexing structure of the H3 system (Source: Uber Engineering [Eng]).

queries, neighborhood traversal and scalable geometric operations by assigning a unique integer-based index to each hexagonal cell while maintaining a small distortion. Its hierarchical nature supports multiple resolutions, allowing applications to dynamically balance precision and performance based on operational requirements.

Hexagonal grids, shown in Figure 2.10, are particularly advantageous in geospatial systems due to their superior properties when compared to square or triangular grids. Specifically, hexagons provide uniform neighbor distances, minimize edge effects and allow seamless hierarchical subdivision without introducing singularities like those found in latitude-longitude systems.

2.2.7.1 Index Definition

The core of the H3 system is its indexing mechanism, which assigns a globally unique 64-bit integer identifier, referred to as an *H3 index*, to each hexagonal cell on the Earth's surface. This index encodes several properties, including the resolution level and the spatial position of the cell within the grid hierarchy.

Formally, the Earth is projected onto an icosahedron, which is then recursively subdivided into finer hexagonal cells across multiple resolution levels. Each resolution step increases the granularity by a factor of approximately seven, enabling both coarse and fine-grained spatial representations.

The H3 index is defined by:

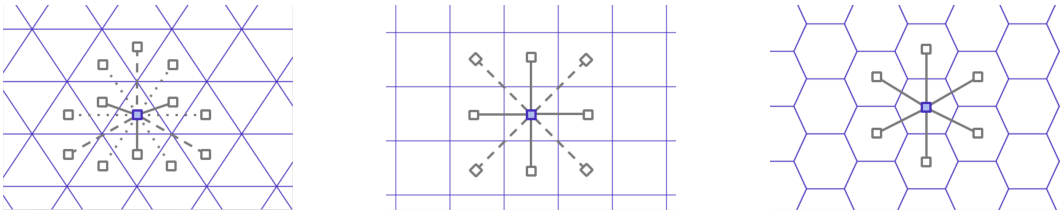
- **Resolution Level:** An integer value ranging from 0 (coarsest) to 15 (finest), representing the level of detail.
- **Base Cell:** One of the 122 base cells that tile the icosahedron.
- **Cell Position:** Encoded in a sequence of directional steps from the base cell to the target cell at the desired resolution.

This compact representation allows for efficient storage, comparison and transmission of geospatial data. Additionally, it supports fast computation of neighboring cells, parent-child relationships and spatial containment, making it highly effective for scalable spatial analysis.

The hierarchical design of H3 also enables spatial aggregation operations, where data can be generalized from finer to coarser resolutions without recomputing raw geographic coordinates. This makes H3 an effective foundation for geospatial data systems that require multi-resolution support.

2.2.7.2 Neighborhood Traversal

One key property of H3 indexes is the ease with which neighboring cells can be identified. This characteristic highlights why hexagons are superior to triangles and squares for spatial indexing. For instance, triangles, when used for tessellation, have 12 neighbors. However, these neighbors vary in their proximity: some share an edge, while others only share a vertex, leading to differing distances between them. Squares, while simpler in their grid structure, also suffer from non-uniform distances between a central cell and all of its direct neighbors. In contrast, hexagons are unique among regular polygons in that **all of their direct neighbors share an edge and are equidistant from the central hexagon**, as shown in Figure 2.11.



(a) Triangle-based grid with 12 neighbors.

(b) Squared-based grid with 8 neighbors.

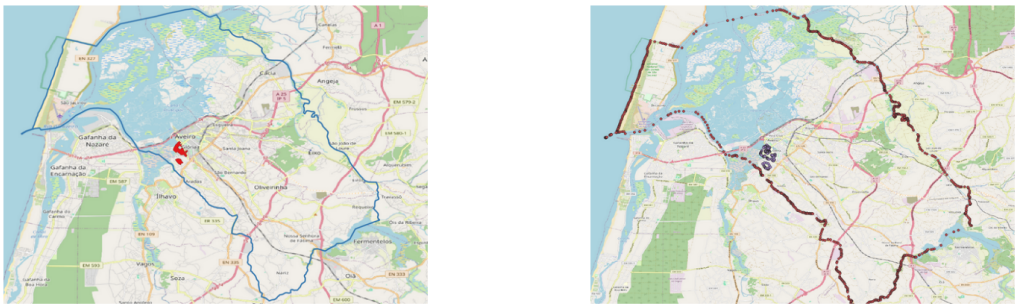
(c) Hexagonal-based grid with 6 neighbors.

Figure 2.11: Neighborhood comparison between polygons-based grid (Source: Uber Engineering [Eng]).

This uniformity in neighbor distance is crucial for efficient and simple spatial queries. This property greatly simplifies "neighborhood traversal", the process of finding and analyzing nearby locations, which is fundamental for many location-based services and analytical tasks.

2.2.7.3 Subdivision for Irregular Polygon Regions

Representing irregular polygons on a map is a complex task. Traditional latitude and longitude schemes, especially when using a large number of points to define regions, can lead to significant inefficiencies. For example, if we consider the city of Aveiro, Portugal, it might be represented by approximately 939 reference points, as shown in Figure 2.12b. Applying an algorithm like Ray-Casting to such a large number of points would be highly inefficient for point-in-polygon checks. Furthermore, representing nested regions, such as the University of Aveiro within the city (which alone could require 169 points), would add further complexity.



(a) Aveiro map with the University shown using lines.

(b) Aveiro map with the University shown using points.

Figure 2.12: Geographic Representation of Aveiro, including the University Area.

To optimize this representation using hexagons, one approach is to define a region composed of H3 cells corresponding to a specific region or area. This region can then be stored in a KeyValue

map store, where each key is an `h3_cell`, allowing us to determine the associated `region_id` for any given coordinates. This process can be conceptualized as:

$$\begin{aligned} &(\text{latitude}, \text{longitude}) \rightarrow \texttt{h3\_cell} \quad \text{(low resolution)} \\ &\text{KeyValue}(\texttt{h3\_cell}) \rightarrow \texttt{region\_id} \end{aligned} \tag{2.1}$$

While feasible, a potential issue arises with the quantity of cells that would need to be stored in the hash map to accommodate this flow, as each hexagonal area would require an associated region ID.

Approximately, for the city of Aveiro, with an area of 197.58 km2 and considering hexagons with resolution 13, which provide a precision of approximately 22.135 m2, around 8,926,400 hexagons would be needed for the projection of its area. However, a large part of an organization's area, particularly in its interior, does not require such fine detail. Hexagons, although not perfect like squares for fitting arbitrary shapes, allow for subdivision into seven smaller hexagons (or, more precisely, a parent hexagon can be "compacted" to represent its child hexagons at a finer resolution). This enables a compaction strategy where multiple fine-grained hexagons belonging to a region can be represented by a single larger hexagon, as illustrated in Figure 2.13.

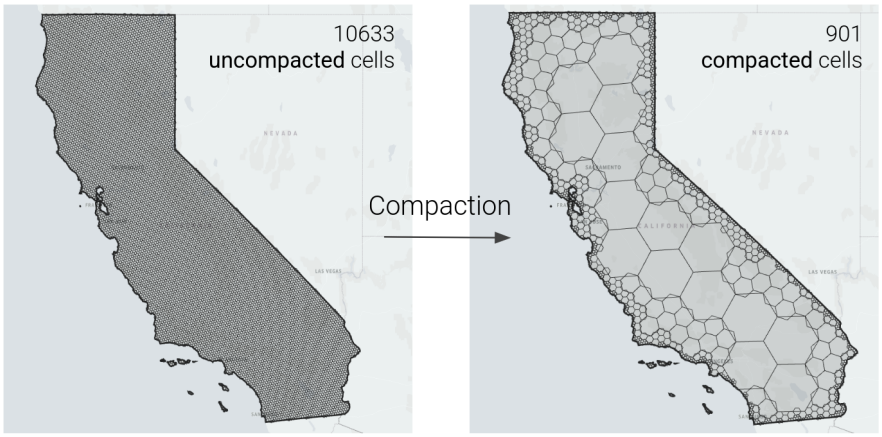


Figure 2.13: H3 Hexagonal Grid Compaction Process. Leaves Only (72 KiB) vs Tree (8 KiB) (Source: Uber Engineering [Eng]).

This approach offers numerous benefits in terms of memory consumption and data management complexity, while maintaining an $O(1)$ search complexity. This $O(1)$ efficiency is achieved because given a (latitude, longitude) pair, the H3 system can directly compute a unique `h3_cell` index at any desired resolution. This allows for a direct lookup in a key value map, which on average, is a constant-time operation. Furthermore, this search can operate in a recursive manner, starting from a higher resolution and progressively moving to lower (finer) resolutions as needed. For instance, if a point's region isn't found at a high resolution, the system can ascend the hierarchy to coarser parent cells, performing direct $O(1)$ lookups at each step, all while preserving the overall $O(1)$ lookup efficiency.

2.2.7.4 Distortion

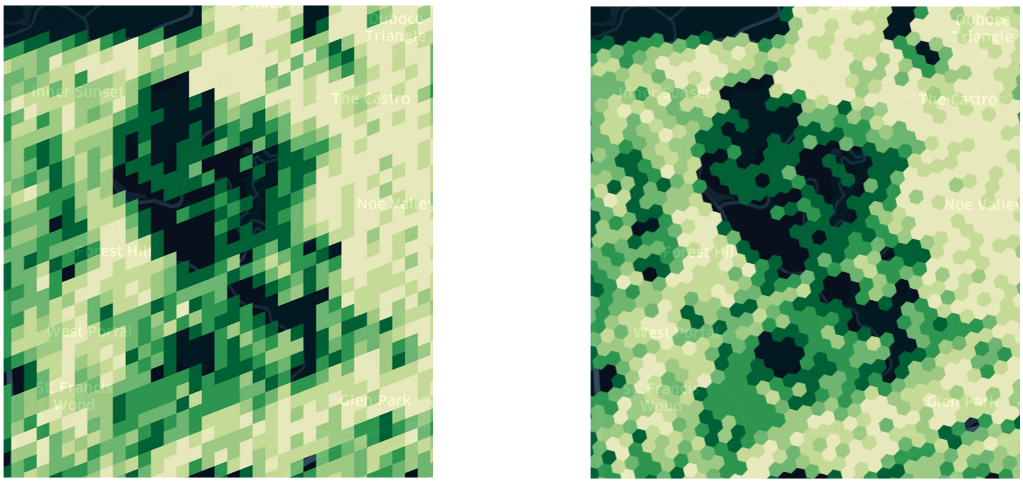
Distortion is a critical aspect when working with geospatial indexing systems, primarily because the Earth is a sphere, not a flat plane. To project a spherical surface onto a 2D grid, various projection methods are employed, all of which inevitably introduce some degree of distortion. The goal of such projections and a key strength of H3, is to minimize this error and allow for seamless transformations between the spherical and planar representations. H3 achieves this by choosing the **Icosahedron** as its base geometric shape for global tessellation. While any projection will

inherently have some distortion, the choice of the Icosahedron significantly reduces angular distortion compared to other shapes like a cube, which would exhibit very large distortions, especially at its vertices.

Another notable advantage of the Icosahedron, particularly when used with a Dymaxion projection, is that the areas with the highest distortion tend to fall over water bodies. This is a significant benefit for many land-based applications, as it ensures that the most crucial land areas experience minimal distortion, preserving the accuracy of spatial analyses.

This distortion is not as critical in terms of the size of individual cells, but rather in the consistency of distances between cells and their neighbors. In different geographic locations, the actual physical distances between cells might vary, impacting a wide range of applications. However, with the hexagonal grid used by H3, these distortions are minimized, ensuring a more uniform and reliable representation of distances across the globe.

Below, in Figure 2.14, you can see a visual comparison of how H3's hexagonal cells and Google's S2 quadrilateral cells represent regions on a map, illustrating the difference in their shapes and how distortion might manifest.



(a) S2-based spatial representation.

(b) H3-based spatial representation.

Figure 2.14: Comparison of distortion using S2 and H3 grids for sample data in San Francisco.

2.2.8 Large Language Models (LLMs)

Large Language Models (LLMs) are advanced artificial intelligence systems designed to process and generate human-like text. They achieve this by being trained on extensive corpora of diverse textual data, enabling them to perform a wide range of language-related tasks, including but not limited to summarization, translation, code generation and question answering [Bro+20].

2.2.8.1 Definition and Characteristics

The rise of LLMs began in 2018 with the release of BERT by Google [Dev+18], marking a significant advancement in natural language understanding. This was followed by OpenAI's GPT-3 in 2020, which demonstrated unprecedented capabilities in language generation with 175 billion parameters [Bro+20]. Recent developments, such as OpenAI's GPT-4 and Google's Gemini, have further expanded these capabilities, including the integration of multimodal processing that supports both textual and visual. One of the core operational principles of LLMs is the use of a **context window**, which defines the maximum number of tokens (words, punctuation and spaces) the model can process in a single interaction. Early models like GPT-3 supported context windows of up to 4,096 tokens, limiting their ability to handle extensive documents or maintain long conversations. In contrast, contemporary models such as GPT-4 and Gemini offer expanded

context windows ranging from 8,000 to over 1 million tokens, significantly improving their capacity to manage complex and lengthy interactions [Dee23].

Modern LLMs also exhibit **multimodal capabilities**, allowing them to process various types of data, including text, images and audio. This versatility enhances their applicability across different domains, from automated customer service to advanced medical diagnostics.

Benchmark evaluations have shown that state-of-the-art LLMs not only exceed average human performance in general language tasks but also demonstrate competitive results against domain experts in specialized fields. These evaluations are typically conducted using standardized datasets and metrics such as the Massive Multitask Language Understanding (MMLU) benchmark [Hen+20].

2.2.8.2 Cloud-Based vs. Self-Hosted Solutions

Deploying LLMs can follow two primary strategies: cloud-based services and self-hosted solutions. Cloud-based LLM services, provided by vendors such as OpenAI and Google, offer straightforward integration via APIs. These services eliminate the need for infrastructure management, making them ideal for **fast prototyping and proof-of-concept (PoC)** development [Ope24]. They typically include free usage tiers, allowing limited access to their capabilities without financial commitment. For example, Google's Gemini API provides a free tier with a capped number of requests per minute, suitable for small-scale experiments and early-stage validation [Goo24a].

In contrast, self-hosted LLM solutions grant organizations full control over data privacy, security and model customization. By deploying models on private infrastructure, organizations can ensure that sensitive information remains confined within their controlled environment, addressing regulatory requirements such as the General Data Protection Regulation (GDPR) [Eur16]. Moreover, self-hosting enables offline deployments, removing dependency on external internet services.

A key advantage of self-hosted LLMs is the ability to **perform fine-tuning**. Fine-tuning involves further training a pre-existing model on domain-specific datasets to optimize its performance for specialized tasks. This process allows organizations to improve the model's accuracy on tasks such as legal document summarization or technical support automation. However, fine-tuning requires access to substantial computational resources, typically involving high-performance GPUs and must be carefully managed to avoid overfitting.

It is important to note that for PoC and early-stage projects, leveraging general-purpose cloud-based models without fine-tuning is often sufficient. These models provide a cost-effective and low-risk pathway to validate concepts before committing to the complexities of self-hosted solutions. The following Table presents a comparative overview of popular LLM offerings.

Table 2.6: Comparison of Popular LLM Solutions.

Model	Multimodal	Scalability	Free Tier	Accuracy	Self-Hosted	1M Tokens
GPT-4	Yes	High	Limited	High	No	$5.00
Gemini	Yes	High	Limited	High	No	$0.10
DeepSeek	Yes	Yes	Yes	Medium	Yes	Self-Hosted

2.2.8.3 Asynchronous LLM Processing

The integration of Large Language Models (LLMs) into real-time and high-demand applications necessitates architectural strategies that support scalability and responsiveness. Asynchronous processing is one such strategy, **enabling systems to decouple** computationally intensive language tasks from user-facing operations. This architecture ensures that long-running processes, such as image classification, summarization, or document analysis, **do not block the main execution flow**, thereby maintaining a fluid and responsive user experience [New15].

In asynchronous architectures, task delegation is commonly managed through message queues or distributed streaming platforms. Technologies such as Apache Kafka, RabbitMQ and Amazon

SQS exemplify widely adopted solutions in this domain [Apa24], [VMw24], [Ama24]. These platforms serve as intermediaries that facilitate the queuing, distribution and persistence of processing jobs between producers (e.g., API endpoints) and consumers (e.g., backend workers or LLM services). They support message durability, delivery guarantees and horizontal scalability, essential features for robust AI-powered applications.

This pattern is particularly well-suited to LLM workflows, where task durations can vary significantly based on input complexity, model characteristics and available compute resources. By leveraging asynchronous processing paradigms, modern architectures enhance throughput, improve system resilience and enable more efficient utilization of AI capabilities across distributed infrastructures.

Chapter 3

Product and Vision Concept

3.1 Vision Statement

The proposed system envisions a smarter urban platform where city operators and citizens collaboratively contribute to enhancing the overall quality of life in the city. This collaboration is essential for addressing urban challenges effectively and responsively.

Currently, incident reporting is often conducted manually through **inefficient channels** such as phone calls or emails. These fragmented methods lead to delayed responses and hinder the ability of municipal authorities to prioritize and resolve the most pressing and recurring urban issues. The absence of a direct and structured communication channel between citizens and public services further exacerbates these inefficiencies.

To address these limitations, the platform seeks to modernize the entire lifecycle of urban issue management, from initial reporting to resolution, by leveraging artificial intelligence, geospatial indexing and scalable infrastructure. This vision aligns with the principles of digital transformation in public administration and contributes to broader smart city initiatives.

3.2 Product Concept Overview

The proposed platform, **FixAI**, is an AI-driven system designed to optimize the detection, classification and resolution of urban infrastructure issues. It combines citizen input with automated backend processing to establish a continuous, data-informed feedback loop between users and municipal authorities.

The system comprises **two primary user-facing components**: a mobile application and a desktop interface. The mobile application enables citizens to report urban incidents with minimal effort. Users can capture a photo of the issue, after which the application automatically extracts relevant metadata, such as geolocation coordinates. An integrated AI module processes the image to generate a structured description, classify the incident by type (e.g., broken traffic light, pothole) and assess its severity. The application also offers a personal dashboard where users can monitor the status of their reports (e.g., pending, in progress, resolved) and view a history of reported issues. Incidents are displayed in both list and map formats, providing flexible and intuitive visualization options.

The desktop interface is designed for municipal organization operators and functions as a digital control center for managing citizen reports. It provides a customizable dashboard where incidents are grouped by category, status and other filters defined by the organization. Operators can access detailed records for each incident, including all associated occurrences, through a dedicated detail page. The system supports three visualization modes: the summary dashboard, the incident detail view and a dynamic map interface. The map allows filtering by category and status, and includes a heatmap visualization to identify areas with a high density of reports.

One of the major challenges faced by municipal authorities is the influx of **redundant reports** about the same issue, which consumes valuable time and resources. To address this, the platform incorporates functionality to automatically group related occurrences into a single incident. This is achieved through a combination of geospatial indexing, using the H3 framework (see Section 2.2.7) and AI-based similarity analysis to determine whether multiple reports refer to the same underlying problem or to distinct issues occurring in close proximity.

Together, these components form a cohesive and scalable solution for participatory urban governance. The platform enhances communication between citizens and public services, while supporting data-driven strategies for infrastructure maintenance and urban policy development.

3.3 User-Centered Design

The development of the proposed platform followed a **user-centered design** approach to ensure that both technical and usability requirements are aligned with the needs of its primary stakeholders. This section presents the key user **personas** identified during the design phase, outlines representative **scenarios** and **narratives** based on their expected interactions with the system and concludes with the specific **use cases** that guided functional development.

3.3.1 Personas

Two primary personas were defined to guide the design process: the citizen reporter and the municipal control operator. These personas reflect the system's core interaction model, which relies on the submission and management of urban incident reports.

The **first persona, João,** is a 35-year-old **local resident** who frequently commutes within the city, as shown in Figure 3.1. As a concerned citizen, he is motivated to contribute to urban safety by reporting issues such as potholes, accidents, or vandalism. João has previously found existing municipal services to be unresponsive, slow and impractical when it comes to reporting problems. Furthermore, he lacks any reliable means to track the resolution status of submitted incidents. His expectations from the platform are clear: a user-friendly mobile application that allows him to quickly submit reports, receive confirmation of their submission and monitor their resolution status. João also values access to a personal history of his reports, which enhances his sense of accountability and engagement.

Figure 3.1: Persona: João - Citizen Reporter.

The **second persona, Ana,** is a 45-year-old **employee in the city's urban management department**, as shown in Figure 3.2. Her primary responsibility is to monitor reported incidents and coordinate the appropriate response from municipal service teams. Ana faces several

operational challenges, including a high volume of duplicate and unstructured reports, difficulty prioritizing incidents based on urgency or category and limited visibility into real-time updates. Her primary motivation is to have a centralized dashboard that provides a comprehensive overview of incidents across the city. She values automated support for clustering similar reports and visualizations such as maps and statistics to enhance decision-making.

Figure 3.2: Persona: Ana - City Control Operator.

3.3.2 Scenarios

To validate the system's design and its alignment with real-world user needs, several representative **user scenarios** were developed based on the identified personas. These scenarios illustrate typical interactions that occur between end-users and the platform, highlighting both functional capabilities and user-centric outcomes.

Scenario 1: João Reports a Pothole. While commuting to work, João notices a large pothole on the road. He opens the municipal mobile application, takes a photo of the pothole and confirms the detected location. During the location confirmation, the integrated AI module automatically generates a brief and contextually accurate description (e.g., "Large pothole on Main Street") and categorizes the problem based on its type. João reviews the generated information and submits the report. The entire process takes less than 30 seconds, leaving João satisfied with the ease and efficiency of contributing to urban improvement.

Actions: Opens the municipal app; takes a photo; confirms location; reviews AI-generated description and category; submits the report.

Scenario 2: João Checks the Status of a Previously Reported Incident. On his way home, João recalls having reported a broken streetlight in his neighborhood two weeks earlier. To follow up, he opens the mobile application and navigates to the "My Reports" section. The interface provides two modes of visualization: a list of all previously submitted reports along with their current statuses, "Pending", "In Progress" or "Resolved", and a map view displaying all incidents with geospatial markers. João quickly locates the streetlight incident and sees that its status has been updated to "In Progress", indicating that the municipal services are actively addressing the issue.

Actions: Opens the app; navigates to "My Reports"; locates the incident; checks the current status.

Scenario 3: Ana Reviews Daily Incident Reports. Midway through her workday, Ana logs into the city control center's desktop-based platform to review all new incident reports submitted that day. The dashboard presents incidents grouped by type, resolution status and severity. Upon filtering the reports, she notices a series of unresolved pothole incidents across various city districts. Based on this overview, Ana allocates the appropriate municipal teams to the affected areas and updates the incident statuses to "In Progress".

Actions: Logs into the control center platform; filters incidents by category and status; identifies unresolved cases; delegates tasks to appropriate teams; updates incident statuses.

Scenario 4: Ana Manages a Surge of Reports Following a Storm. After a severe storm, Ana monitors the city's incident management platform to assess urban damage. The dashboard, which aggregates real-time citizen reports, indicates over 200 newly submitted cases, including flooding, fallen trees, damaged sidewalks and traffic signal malfunctions. She contacts municipal repair crews to address the most critical situations first and updates the status of the selected reports to "In Progress".

Actions: Accesses the incident management dashboard; reviews citizen reports; dispatches repair crews; updates incident statuses.

3.3.3 User Stories

Building upon the previously defined personas and interaction scenarios, a structured set of **user stories** was derived to inform the functional requirements and guide the development of the FixAI platform. These user stories reflect the core expectations and objectives of the primary stakeholders, citizens and municipal control operators, and are intended to ensure that the system supports both usability and operational efficiency.

From the citizen's perspective, the platform must facilitate **fast** and **intuitive reporting** of urban issues such as potholes, fallen trees, or vandalism. Users should be able to submit a report through the mobile application with minimal effort. The integrated **AI component automatically analyzes the uploaded photo** to generate a suggested description and categorize the issue, enabling users to complete their report submission in less than one minute. Furthermore, citizens expect access to a history of previously submitted reports and the ability to monitor the resolution status of each incident. Real-time updates indicating progress, such as transitions from "Pending" to "In Progress" or "Resolved", are essential for fostering user engagement.

From the perspective of municipal control operators, the system must provide a centralized dashboard that organizes all reported incidents by category and resolution status. Operators require filtering capabilities to manage incidents effectively, including sorting by status (e.g., pending, in progress, resolved) or category (e.g., infrastructure, urban drainage, traffic). The ability to access all related reports and images for a specific location is critical to assess problems comprehensively, reduce redundancies, and streamline resource allocation. Additionally, a live map visualization of the city should support rapid identification of affected areas and improve coordination of response teams. Operators must also be able to update the status of incidents in real time, ensuring that citizens are promptly informed of ongoing resolutions.

3.3.4 Identified Use Cases

The use case analysis of the FixAI system was conducted to formally capture the functional requirements associated with each user role. This analysis resulted in a structured model comprising **two main functional domains**: the mobile application and the desktop application. These domains reflect the responsibilities of the system's primary users, citizens and city control operators.

Figure 3.3 illustrates the complete use case model, depicting the relationships **between users and the functionalities they interact with** in each application package.

As illustrated, **citizens** interact exclusively with the **mobile application** to report and monitor incidents, while **city control operators** manage and respond to these reports through the

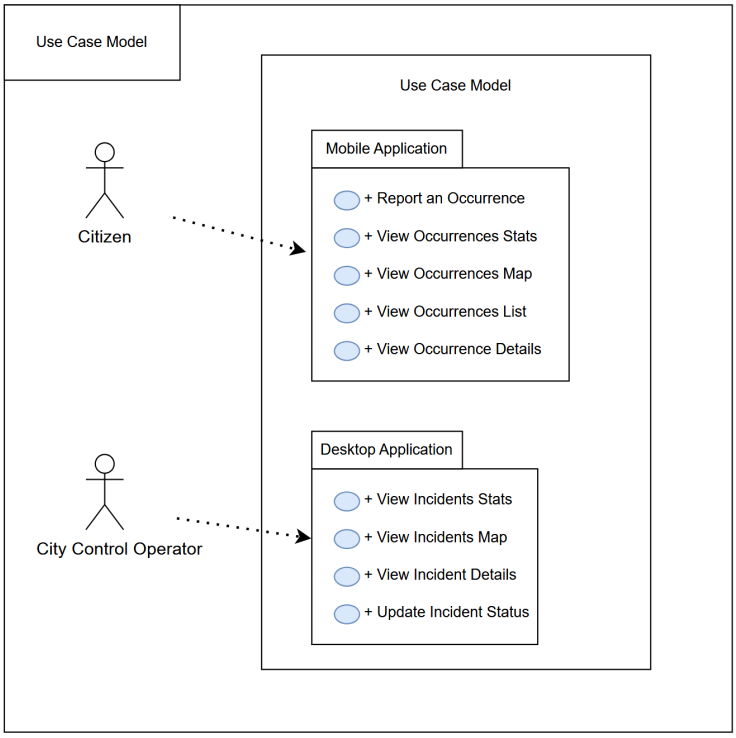


Figure 3.3: Use Case Model of the FixAI System.

desktop application. The following subsections describe the use cases associated with each platform.

3.3.4.1 Mobile Application

The mobile application use case diagram, shown in Figure 3.4, outlines the core functionalities available to the citizen user. Each use case reflects a key interaction within the application, designed to facilitate quick and effective participation in urban maintenance.

Report an Occurrence. Enables citizens to submit incident reports via the smartphone's camera and internet connection. Upon identifying an issue, the user captures an image, and the system generates a description and category using AI. The user may review or modify this information before final submission.

View Occurrences Statistics. On launching the application, users are presented with summary statistics showing the total number of reports, along with counts for each status: pending, in progress, and resolved. These statistics are refreshed only when the application is accessed with an active internet connection.

View Occurrences Map. Allows users to visualize previously reported issues on an interactive map. Each marker corresponds to a report, which users can select to access detailed information.

View Occurrences List. Provides a list-based alternative to the map view. Citizens can browse their past reports chronologically, with each entry linking to a detail page. Filtering by status (pending, in progress, resolved) is also supported to enhance usability.

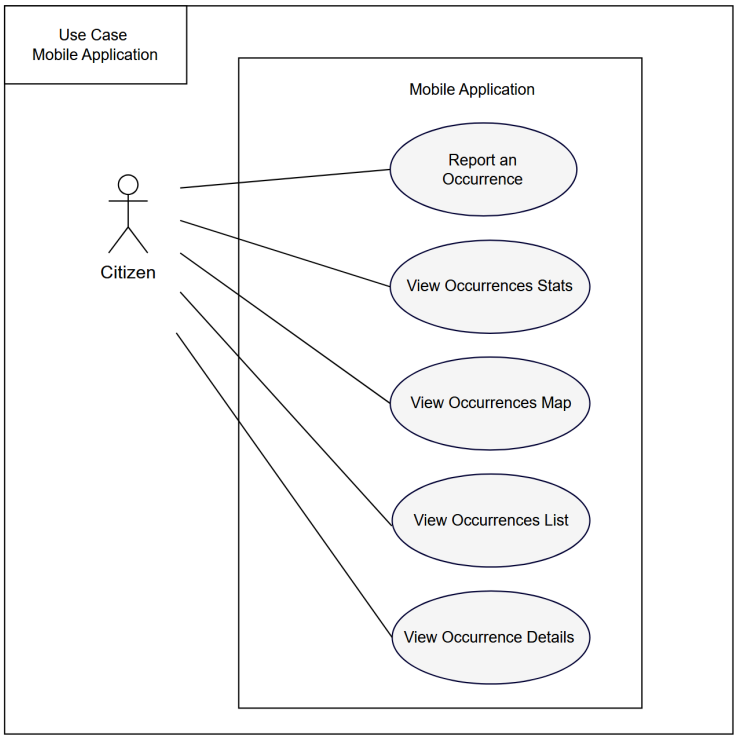


Figure 3.4: Use Case Diagram – Mobile Application.

View Occurrence Details. Displays comprehensive information about a specific report, including its category, submission date, current status, geolocation, and description.

3.3.4.2 Desktop Application

The desktop application use case diagram, shown in Figure 3.5, captures the activities available to city control operators. These use cases are focused on managing incoming reports and monitoring incident resolution.

View Incidents Statistics. Provides a comprehensive dashboard summarizing the total number of reported incidents, categorized by type and resolution status (e.g., pending, in progress, resolved). This overview enables operators to monitor the overall state of urban issues across the city. The statistics are updated each time the dashboard is accessed with an active internet connection.

View Incidents Map. Provides a geographic visualization of reported incidents. Operators can select incidents directly on the map and apply filters by category to refine the display.

View Incident Details. Grants access to the full history and context of a specific incident, including all associated occurrences. Each occurrence entry includes the submission date, description, and relevant metadata. Operators can also view the overall category and current status of the incident.

Update Incident Status. Allows operators to update the status of incidents based on their current resolution stage, transitioning reports from pending to in progress, and ultimately to resolved.

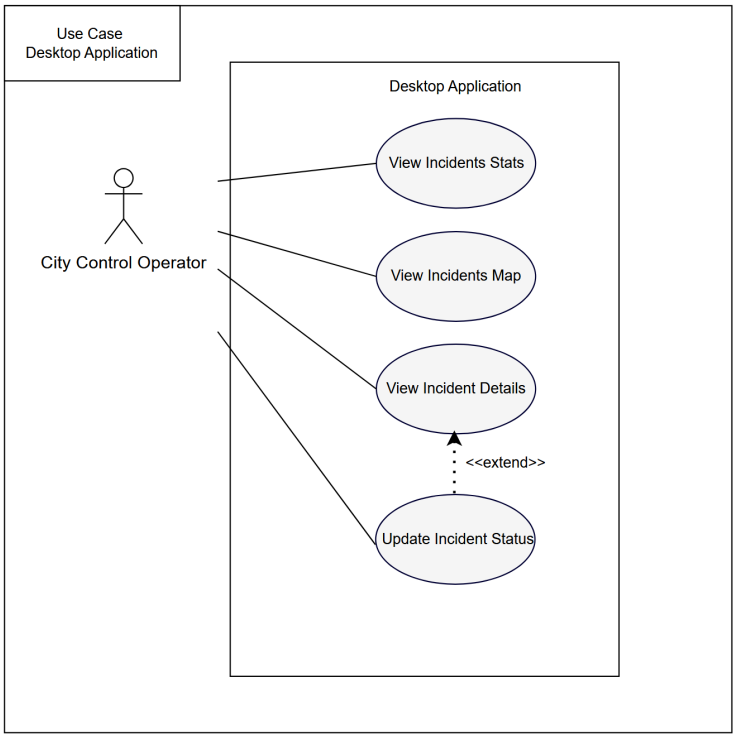


Figure 3.5: Use Case Diagram - Desktop Application.

3.4 Non-Functional Requirements

In addition to the functional requirements outlined in the user stories, the system must meet several non-functional requirements to ensure reliability, performance, security, and usability. These requirements define the quality attributes of the platform and provide guidelines for its implementation.

Performance. The system requires an **LLM Model** to generate an incident description and category within **10 seconds**. In cases of model failure, users must be able to manually input the description and select a category.

Scalability. The application must **scale horizontally** through the addition of instances, rather than relying on hardware upgrades. **Database queries** are to be optimized for efficient handling of increasing data indexation.

Maintainability and Extensibility. The codebase will adhere to **clean architecture** principles to facilitate independent updates and a maintainable environment. The system's design must allow for **easy integration** of new AI models and features with minimal rework. Components should be **loosely coupled** and **highly cohesive**, enabling easier modifications and integration with multiple systems, such as a **Digital Twin**.

Reliability and Availability. **Incident reports and status updates** must be synchronized in real-time across both the **mobile app and desktop platform**. A failure in an **AI model** should not compromise the entire system; users must still be able to submit an incident with manual description and category selection.

Security and Privacy. Communication between the **mobile app, desktop platform, and backend services** must be encrypted using **HTTPS with TLS** protocols.

Usability. Both the **mobile app** and **desktop platform** require a **user-friendly interface** that is intuitive for citizens and city control operators. The **mobile app** must be compatible with **Android and iOS**.

Accessibility. The platform should support **multiple languages** to accommodate diverse populations. **Font size** must be easily adjustable without disrupting the **UI layout or design**.

Chapter 4

Architecture Notebook

4.1 Architecture Overview

This section provides a comprehensive overview of the architecture of the system, describing its core structural components and the principles that guided their composition. The architecture has been designed following modern conventions, with a focus on **scalability**, **maintainability**, and **modular responsibility separation**. It reflects a **service-oriented paradigm** and adheres to Kubernetes deployment patterns, ensuring robustness under dynamic operational loads and extensibility across diverse use cases. The architecture is loosely coupled and **event-driven**, relying heavily on asynchronous communication and stateless services to promote horizontal scalability and fault isolation.

The following subsections elaborate on the architecture through a conceptual diagram, a deployment perspective, and the data access strategy, providing a high-level understanding of how our entire application works.

4.1.1 Architecture Diagram

The architecture of the system shown in Figure 4.1 adopts a **modular layered design** deployed on a **Kubernetes infrastructure**. The design is characterized by a clear separation of concerns between functional components, stateless service orchestration, and event-driven extensions, which together enable the system to operate efficiently under varying loads while remaining adaptable to future functional evolution. At a high level, the application is structured into three

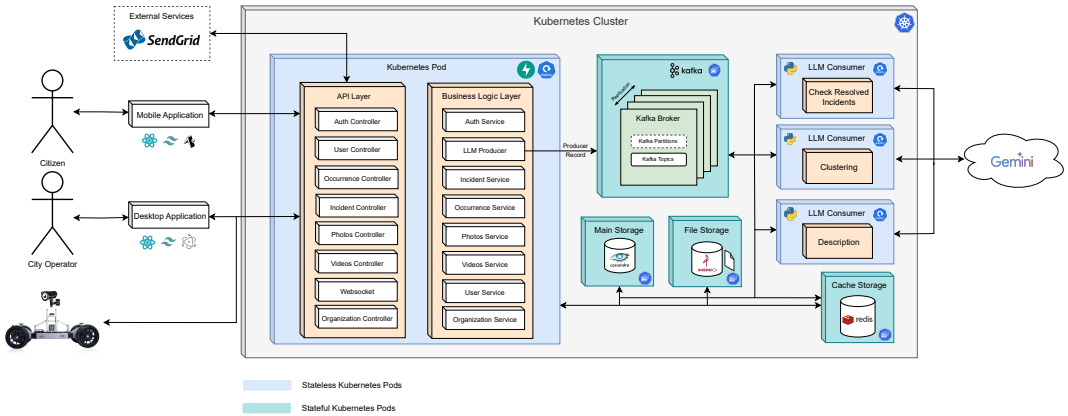


Figure 4.1: Architecture Diagram with Kubernetes Pods.

main domains: the external interaction layer composed of frontend clients and edge-data devices, the core backend service hosted within a stateless Kubernetes pod, and an asynchronous event-driven pipeline used for **LLM tasks**. These components are unified within a containerized deployment model that facilitates elastic scalability, isolation of failure domains, and ease of operational management.

The external interaction layer comprises multiple entry points to the system. End users, categorized as citizens or city operators, interact through dedicated interfaces, mobile and desktop applications, respectively and, additionally, the system interfaces with edge computing devices, such as autonomous vehicles equipped with vision capabilities, which are capable of identifying solved incidents directly to the backend. All of these clients and devices communicate via a centralized **RESTful API** exposed through the Kubernetes cluster into the backend pod.

The backend itself follows a strict layered architecture and is deployed as a **stateless Kubernetes pod**. Internally, the backend is divided into an **API Layer** and a **Business Logic Layer**. The API Layer is responsible for handling all external requests, including user authentication, incident and report management, and unidirectional communication with clients via web sockets. Each controller in this layer delegates execution to corresponding services in the Business Logic Layer, where domain-specific operations are executed and data is retrieved or persisted. This design adheres to the principle of **responsibility separation** and supports easy extension of functionality over time.

The Business Logic Layer maintains access to the system's persistent data sources and integrates with external service providers. Core data entities such as users, incidents, and organizational information are stored in a **high-availability distributed database** optimized for high write throughput. Media artifacts, including photos and videos, are offloaded to a dedicated object storage system and, additionally, the backend makes use of an in-memory data store for ephemeral caching, fast key-value access, and pub/sub coordination. Certain services also interact with third-party providers for functionality such as email notifications.

In order to support asynchronous processing and extend system responsiveness, the architecture includes an **event-driven pipeline** based on message brokering principles. Within the Business Logic Layer, specific operations trigger the publication of messages to a distributed message broker where these messages are organized by topic and correspond to distinct job types, such as checking if an incident is resolved, clustering reports or describing incidents using natural language. These topics are consumed by independently deployed stateless processing components, referred to as **LLM Consumers**, which are capable of scaling horizontally depending on computational demand. Each consumer retrieves messages from its designated topic, constructs semantic prompts, and delegates the final processing to a **cloud-based LLM service**. The results of these operations are subsequently returned to the user asynchronously, completing the event cycle without requiring client-side polling.

Persistent data in the system is managed by Kubernetes **StatefulSet**s, which guarantee stable network identities and volume persistence for the storage components. The architecture distinguishes between **main storage**, used for structured data; **file storage**, used for handling binary media; and **cache storage**, which supports high-speed data access patterns and transient coordination logic. All of these systems are hosted within the same Kubernetes cluster, ensuring efficient inter-service communication and centralized observability.

Overall, this architectural design prioritizes **resilience**, **modularity**, and **scalability**. The core backend services maintain a monolithic deployment structure internally, but their modular decomposition and reliance on external asynchronous processors allow the system to benefit from the scaling properties commonly associated with microservice-based architectures. By combining a modular monolith with asynchronous, independently scalable processing units, the system achieves a **hybrid architecture** capable of supporting a wide range of operational scenarios with minimal latency and robust fault isolation.

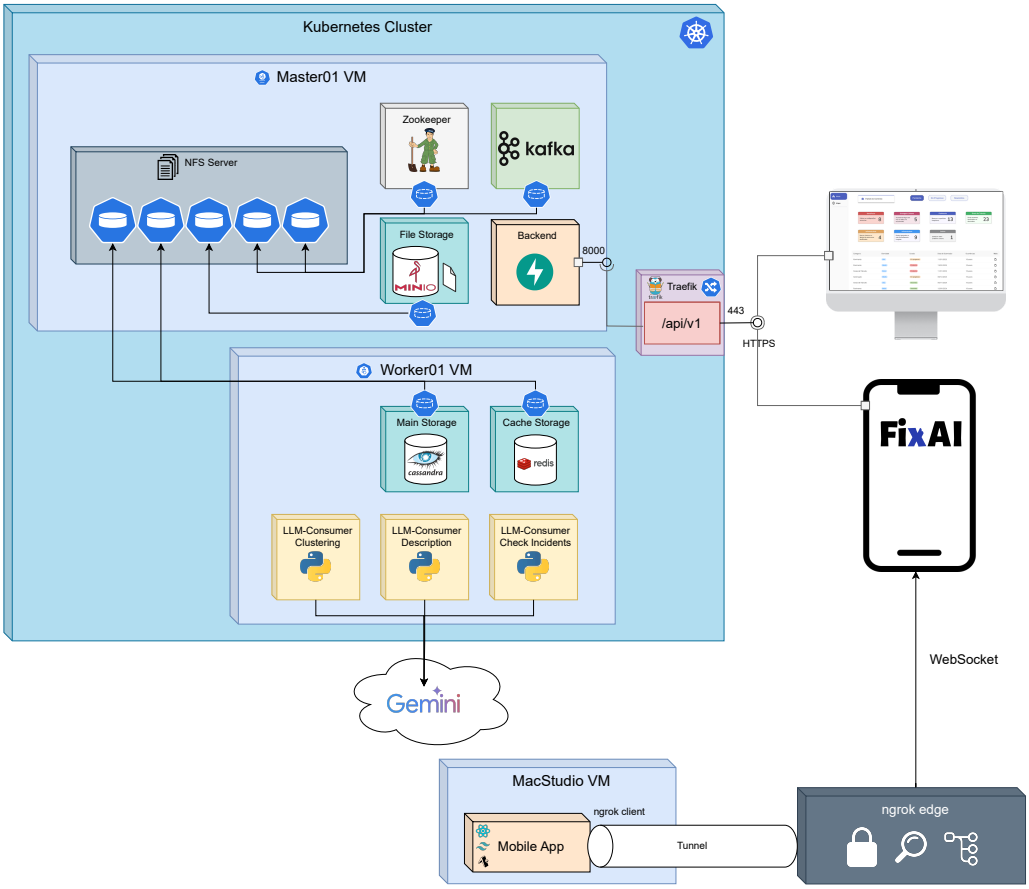


Figure 4.2: Deployment Diagram with Infrastructure.

4.1.2 Deployment Diagram

The deployment of the system, as illustrated in Figure 4.2, is realized through a **containerized infrastructure** orchestrated by a **Kubernetes cluster** composed of two virtual machines: one configured as the control plane (Master01) and the other as a worker node (Worker01). Both nodes operate within an Ubuntu environment and are responsible for hosting and managing all components required by the application, including backend services, message brokers, storage, and event-processing units.

The user-facing applications, both the mobile client and the desktop operator interface, interact with the system through a secure HTTPS connection where incoming traffic is managed by an ingress controller, specifically **Traefik**, which routes external requests to the internal API services. This controller forwards traffic to the backend pod over its exposed internal service port 8000. This configuration ensures a clean separation between external access and internal service orchestration while adhering to standard Kubernetes ingress routing principles as discussed in Section 2.2.5.

Each pod within the cluster is exposed internally through a **ClusterIP service**, which abstracts individual pod identities and performs internal load balancing. This approach facilitates **horizontal scaling**, as new replicas of stateless services can be deployed without requiring manual updates to pod-specific configurations. Components that require persistent identity and data storage, such as Cassandra, Kafka, and Zookeeper, are deployed as **StatefulSets** and are accompanied by **headless services** to support direct pod discovery and coordination so their internal distribution is possible. These distributed components form the backbone of the system's storage and event-driven subsystems, enabling both high availability and consistency under dynamic workloads.

For persistent storage, the system uses a centralized **NFS server** hosted in the Master01 node combined with dynamically provisioned Persistent Volumes via Kubernetes StorageClasses. This setup ensures that each pod requiring persistence is assigned a dedicated volume, supporting **fault-tolerant storage allocation** and replication. Data associated with structured storage (Cassandra), media objects (MinIO), and ephemeral cache (Redis) is thereby maintained in a resilient and scalable fashion within the same cluster infrastructure.

Special deployment considerations were necessary for the mobile application due to its uncompiled state, which precludes conventional distribution through app stores. Instead, the app remains in development mode and is accessed using Expo Go via QR code. To make this possible in a persistent and publicly accessible manner, the application is hosted on a MacStudio virtual machine and, since this VM resides within a private network and is not exposed directly to the internet, a secure tunnel is established using **ngrok**. This tunneling solution allows the Expo Go client to connect externally, ensuring reliable access to the app without requiring local infrastructure to remain online at all times.

This deployment strategy balances **performance**, **isolation**, and **operational simplicity**. Stateless components benefit from standard Kubernetes scaling and failover mechanisms, while stateful services are maintained with data integrity guarantees provided by persistent volumes and headless services. As explained in Section 2.2.5, all these components and patterns allow the system to remain robust under increasing load, while minimizing the operational complexity commonly associated with distributed architectures.

4.1.3 Data Access Diagram

The data access model of the system, depicted in Figure 4.3, outlines the **interaction patterns** between components across the internal and external domains, emphasizing access privileges, communication flows, and trust boundaries. The system is logically divided into two domains: the external domain, comprising end users and edge data sensors, and the internal domain, which hosts all backend components, persistent storage, and processing units.

All external communication is funneled through the **API layer**, which acts as the secure gateway to the internal domain. This layer is exposed over HTTPS and implements **role-based access control (RBAC)** enforced via JWT tokens. It receives requests from client applications

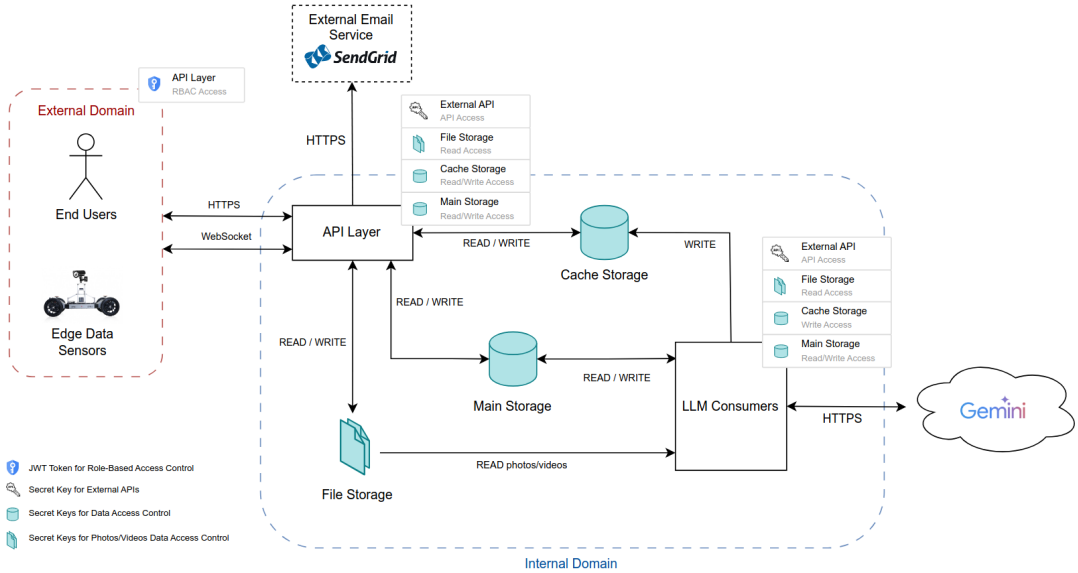


Figure 4.3: Data Access Diagram.

and devices, including mobile users, operators from the desktop application, and sensor-equipped platforms such as autonomous vehicles equipped with cameras. These clients interact exclusively with the API layer and do not have direct access to any internal services. The API layer is responsible for performing access validation and then mediating data operations across the internal storage systems. It can perform read and write operations on the main storage and file storage layers, and selectively interact with the cache storage. Additionally, it communicates with third-party services such as SendGrid for outbound messaging, using its secure API key explicitly scoped to this purpose.

LLM consumers operate asynchronously in the internal domain and have broader access to the internal data infrastructure. These **stateless services** interact with the main storage in both read and write modes, perform writes to the cache, and have read access to the object storage containing photos and videos. LLM consumers also interact directly with the Gemini cloud-based model through secure HTTPS communication with a dedicated secret API key, handling prompt generation and response evaluation autonomously.

The file storage system maintains access controls for multimedia data, allowing only authenticated read operations from the LLM consumers, while granting full access to the backend service. Similarly, the cache and structured databases are protected through **service-scoped secrets** that control access privileges based on component identity and operational role.

This diagram captures the **security-oriented design** of the system by showing clear domain isolation, minimal privilege assignments, and scoped secrets for inter-service communication. By limiting direct exposure and centralizing access mediation through the API layer, the architecture enforces strong data governance while supporting high-throughput data exchange between trusted services.

4.2 Technology Stack

This section outlines the primary technologies employed in the development and operation of the system, covering programming languages, architectural frameworks, infrastructure platforms, and service integrations. The choice of each technology was guided by factors such as scalability, compatibility with containerized environments, support for asynchronous processing, and ease of integration with modern DevOps pipelines. The overall system is composed of a hybrid deployment

Table 4.1: Technology Stack Overview.

Category	**Technology**
Backend Development	Python (FastAPI)
Frontend Clients	ReactNative (Mobile), Electron with React (Desktop)
Container Orchestration	Kubernetes (Ubuntu VM deployment)
Ingress Controller	Traefik
Structured Storage	Apache Cassandra
Object Storage	MinIO
Cache and Pub/Sub	Redis
Message Broker	Apache Kafka
LLM Integration	Google Gemini API
LLM Consumers	Python-based, deployed as stateless pods
Authentication	JSON Web Tokens (JWT)
Email Service	SendGrid
Persistent Volumes	NFS + Kubernetes StorageClasses

combining stateless service pods and stateful distributed components, as detailed in Sections 4.1 and 4.1.2. Below, the Table 4.1 summarizes the main technologies categorized by their functional roles in the architecture.

Chapter 5

Implementation

5.1 Frontend Applications

This section presents the two main frontend interfaces developed for the FixAI platform: the mobile application for citizens and the desktop application for municipal operators. We describe the design decisions, implementation strategies, and tools used in both applications, highlighting the technologies and libraries that enabled efficient development and user experience. Special attention is given to the different requirements and interaction models of each platform, as well as the rationale behind choosing a desktop-native approach over a purely web-based solution for the operator interface.

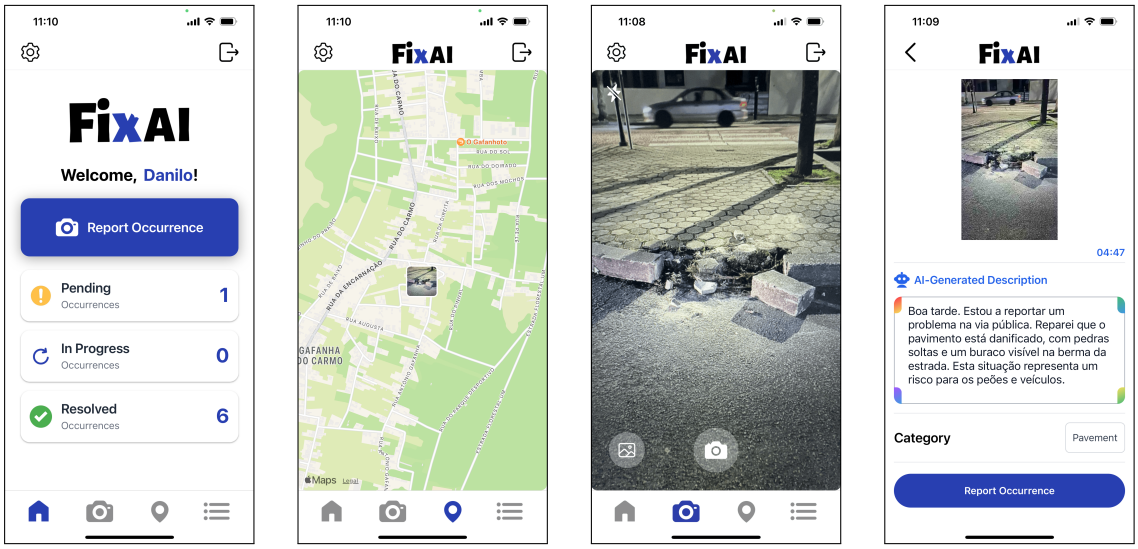
5.1.1 Mobile Application

The mobile application was developed using React Native in conjunction with Expo Go, providing a cross-platform development environment. Expo Go significantly streamlined the development workflow, offering tools for fast prototyping, easy device testing, and compatibility across both Android and iOS platforms. React Native was chosen over alternatives such as Flutter due to the team's familiarity with JavaScript and React, its large ecosystem of libraries, and the reduced learning curve, which allowed for more rapid development.

The application includes several key screens:

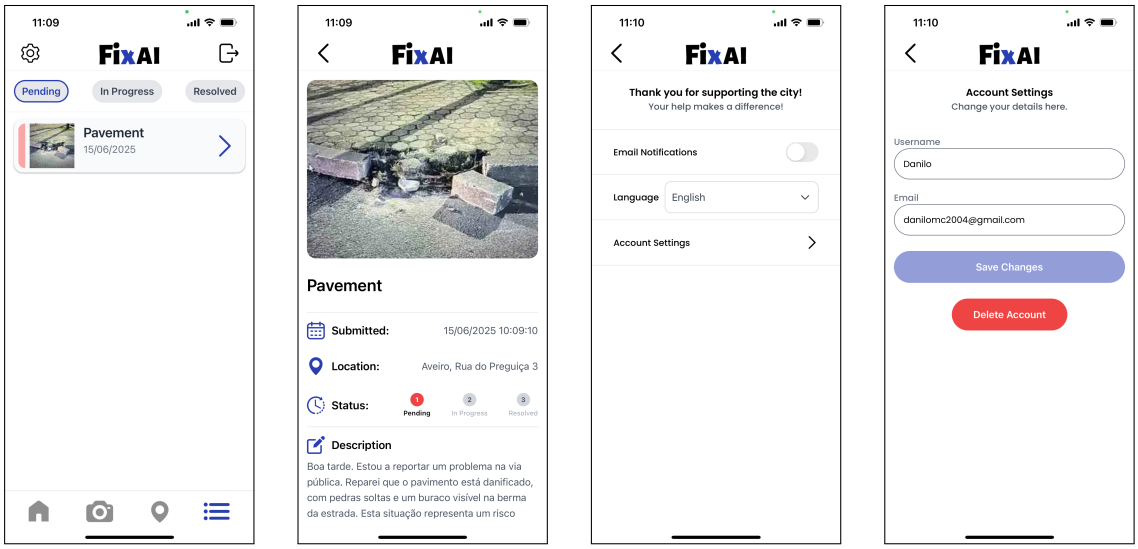
- **Home Page:** Displays user-specific statistics, such as the number of pending, in-progress, and resolved incident reports.
- **Report Page:** Allows users to capture a photo and submit a structured issue report.
- **Map Page:** Shows all current reports (pending and in-progress) using real-time geolocation.
- **List Page:** Displays a filtered list of all submitted issues by status.
- **Account Management:** Includes authentication, language preferences, and user profile options.

Libraries. Camera functionality was implemented using the `expo-camera` library, while user geolocation was handled with `expo-location`, which periodically updates the user's position through a React context. The incident map view was created using `react-native-maps`, offering a performant and customizable mapping component. For backend integration, the app uses `axios` to perform HTTPS requests to the FixAI server.



(a) Home Page. (b) Map Page. (c) Camera Page. (d) Report Page.

Figure 5.1: Mobile Application – Home, Map, Camera and Report pages.



(a) List Page. (b) Details Page. (c) Settings Page. (d) Account Page.

Figure 5.2: Mobile Application – List, Details, Settings and Account pages.

Modularity and Code Organization. The project follows a modular architecture to promote code reusability and maintainability. Common UI elements such as buttons, form inputs, and cards were developed as reusable components. Logical operations and cross-cutting utilities were encapsulated in hooks (e.g., custom hooks for geolocation or permissions) and utility functions. Constants such as icons, fonts, image resources, and supported languages are centralized in dedicated files, ensuring consistent usage throughout the app and simplifying updates. This structure allows for rapid feature extension and easier testing.

Backend Integration. All backend communication is centralized in a dedicated application module (API Consumer), which encapsulates the logic for interacting with the FixAI REST API. This abstraction simplifies HTTPS request logic and promotes consistency in error handling. A secondary file exports the API consumer functions and associated DTOs, allowing for type-safe and organized imports across the codebase. This structure enforces separation of concerns between UI and data-fetching logic.

State Management. The application relies heavily on React's Context API for global state management. Several dedicated context providers were implemented to manage cross-cutting concerns such as authentication, language settings, and geolocation. This modular structure improved code maintainability and reusability across components.

Loading State Management. To provide feedback to users during asynchronous operations (e.g., API requests), a global loading mechanism was implemented using a custom `LoadingContext`. Initially, both the reading and writing logic were handled in a single context. However, this led to performance issues: every time the loading state changed (e.g., `setLoading(true)`), all components consuming the context were re-rendered, even those not directly related to the loading operation.

To solve this, the logic was split into two separate contexts: one for reading (`useLoading`) and one for writing (`useSetLoading`). This separation prevents unnecessary re-renders of components that only require one side of the loading state, improving performance and maintaining a responsive user interface. Internally, the loading state is managed using React's `useState`, and a spinner is conditionally rendered in the UI whenever the state is `true`.

```
const isLoading = useLoading();
const setLoading = useSetLoading();

setLoading(true); // before API call
// perform action
setLoading(false); // after completion
```

Figure 5.3: Frontend Loading Context for API calls.

This lightweight but effective approach avoids flickers and preserves component isolation, resulting in a better overall user experience.

Internationalization. FixAI provides multilingual support for Portuguese, English, and Chinese. To implement this feature, a custom `TranslationContext` was created using React's context API. This context handles language selection, translation lookup, and caching of user preferences.

All translatable strings are defined in language-specific JSON files (e.g., `en.json`, `pt.json`, `zh.json`), each containing key-value pairs where keys are consistent across all files and values are the translated strings. As we can see for Portuguese translations in Figure 5.4.

```
{
  "occurrences": "Occurrencias",
  "report_issue": Reportar um Incidente",
  (...)
}
```

Figure 5.4: Portuguese Translations Example (`pt.json`).

The `TranslationContext` provides a `translate(key)` function that can be used throughout the app to fetch the correct text dynamically. As representend in Figure 5.5.

The language is initially set based on the device's locale, using JavaScript's `navigator.language` or `Intl.DateTimeFormat().resolvedOptions().locale`. If the device's language is not supported, the system defaults to English. Users can also manually change the language, and their selection is stored using `@react-native-async-storage/async-storage`, so it persists across sessions.

Example flow:

1. On app startup, the system tries to load the previously selected language from AsyncStorage.
2. If none is found, it falls back to the device's locale, provided it matches a supported language.
3. The corresponding JSON file is loaded into memory, based on the selected language.
4. The `translate` function accesses the current language's object and returns the appropriate string for each key.

For example, if the current language is set to Portuguese, a call to:

```
translate('occurrences')
```

Figure 5.5: Translations Usage Example.

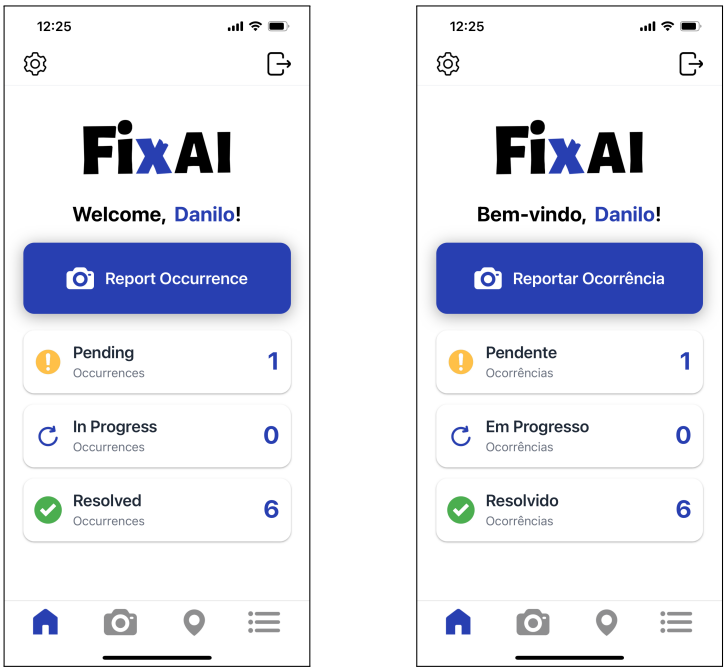
will return the value associated with the key in `pt.json`, such as "`Occurrencias`". This architecture enables fully dynamic and reactive UI translation without the need to reload components. It also makes the app easily extendable to new languages: adding support for another language only requires creating a new JSON file with the appropriate translations and adding it to the resource map. The consistent use of `translate(key)` across the codebase guarantees centralized control over language behavior.

5.1.2 Desktop Application

The desktop application was developed using ReactJS together with Electron, enabling the same codebase used for the web to run as a native desktop application. The integration of Electron required only a lightweight wrapper around the React project, making the transition to desktop seamless and efficient.

Dashboard and Incident Management. The main screen of the application presents a *dashboard* composed of several key components:

- A card-based overview of incident categories.
- A table displaying incidents filtered by category, status, and date.
- An interactive map for geolocating incidents, with optional *heatmap* or *marker* visualization.



(a) English Translation. (b) Portuguese Translation.

Figure 5.6: Mobile Application – Different Languages in Home Page.

- A detail page with a carousel containing all occurrences of an incident (i.e., multiple photos and descriptions referring to the same issue).
- A dedicated section for PIXKIT video submissions, which act as suggestions that a problem may have been resolved and are pending operator validation.

Map and Heatmap Integration. The map was implemented using the `react-leaflet` library, which provides an efficient and interactive mapping experience within React applications. Depending on the selected view mode, the map either displays individual markers for each incident or overlays a heatmap to represent the density of incidents.

The heatmap is rendered through a custom component that leverages the `leaflet.heat` plugin. This plugin enables the transformation of geographical coordinates into a colored gradient, allowing users to quickly identify areas with a high concentration of incidents. The heat intensity is determined by the number of incidents in a given area, and the color gradient ranges from blue (low density) to red (high density).

This feature greatly enhances the operator's ability to assess the distribution of incidents across regions, prioritize responses, and focus on emerging hotspots more effectively.

Language Support. Internationalization is handled through a custom `TranslationContext`, exactly as in the mobile application. The translation system loads static JSON resource files (e.g., `pt.json`, `en.json`) and makes the translation function available globally. Although only Portuguese and English are currently supported, adding new languages is as simple as creating a new key-value JSON file. This approach ensures consistency in multilingual support across all platforms.

API Integration. As with the mobile application, all server communication is handled through the `axios` HTTPS client. The desktop version benefits from the same abstraction and configuration logic, promoting consistency and reducing duplication across platforms.

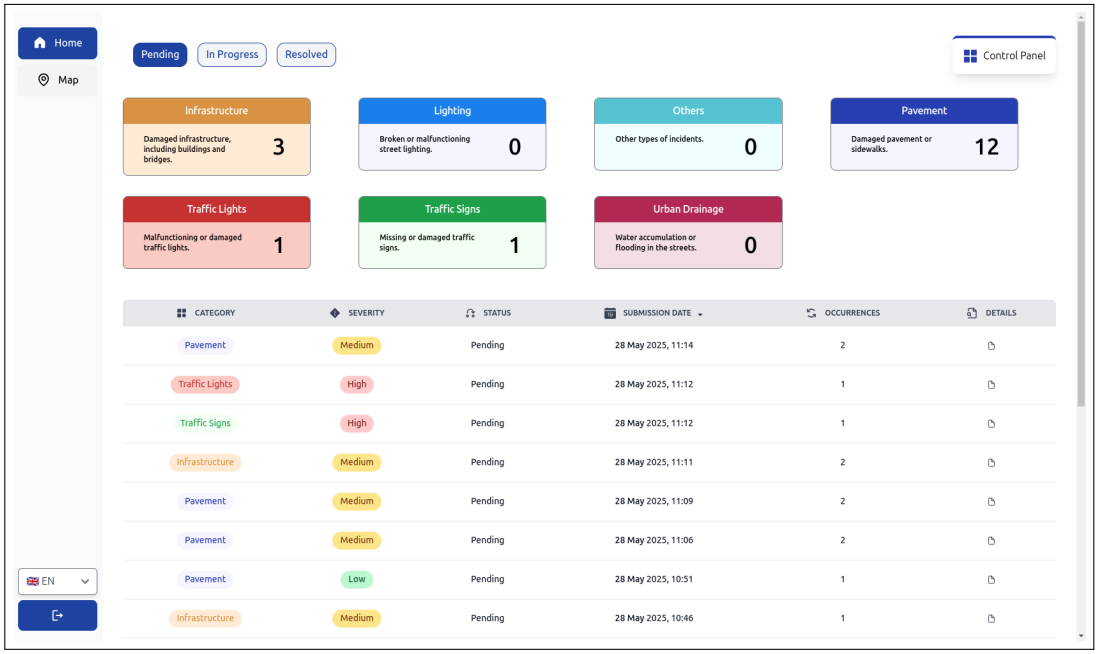
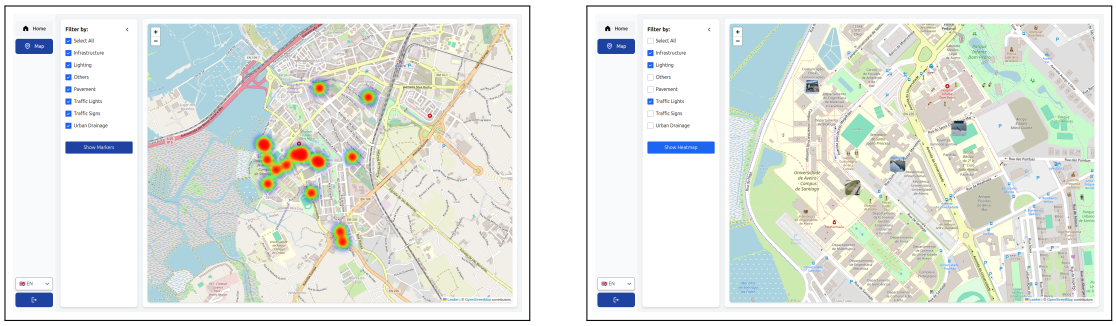


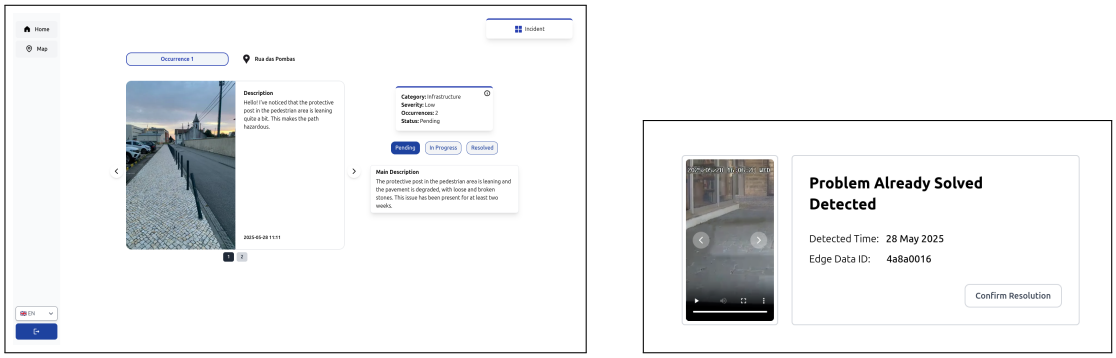
Figure 5.7: Desktop Application Dashboard Page.



(a) Heatmap Visualization.

(b) Map With Markers.

Figure 5.8: FixAI Desktop – Desktop Application Incidents Map Page.



(a) Incident Details Page.

(b) Incident Already Solved Detected.

Figure 5.9: FixAI Desktop – Desktop Application Incident Details Page.

Conclusion. The desktop application leverages the React ecosystem to provide a responsive, multi-functional interface for operators. Thanks to Electron, it runs natively across platforms with minimal setup. Features like multilingual support, advanced map visualization, and robust authentication make the application both powerful and user-friendly. Moreover, code reusability and modularity across mobile and desktop significantly improved development efficiency.

5.1.2.1 Desktop App vs. Web App

Web vs Desktop Application. The application was originally designed and implemented as a web-based platform, using technologies such as ReactJS and Axios for frontend logic and API communication. However, after further consideration and based on user feedback, we realized that the primary users of our system, the operators within organizations, would benefit more from a dedicated desktop application.

To address this, we decided to wrap our entire application using `Electron`, a framework that enables the packaging of web applications as cross-platform desktop applications (Windows, macOS, and Linux). One of the major advantages of this approach was that it allowed us to reuse our existing web codebase without the need to rewrite the application in native technologies. Several key factors motivated this migration:

- **Ease of Access and Use:** A desktop app removes the dependency on a web browser, enabling users to launch the application directly from their system. This is particularly practical for operators who use the system on a daily basis.
- **Better Operating System Integration:** Electron applications can take advantage of native OS features such as window management, native notifications, local storage, and custom menus, offering a more seamless user experience.
- **Unified User Experience:** By using Electron, we ensured a consistent look and behavior across platforms, avoiding browser-specific quirks and improving interface reliability.
- **Partial Offline Support:** Although the application still relies on a remote API, features like local caching and persistent state allow for smoother performance in environments with unstable connectivity.

This transition to a desktop environment did not imply abandoning the web version entirely. Instead, it was a strategic decision to better align the application with its most relevant use case. Thanks to our modular architecture, both the web and desktop versions can be maintained with minimal overhead, ensuring flexibility and adaptability for different usage contexts.

5.2 Backend Services

5.2.1 API Design and Endpoints

The backend is developed using **FastAPI**, a modern, fast (high-performance) web framework for building APIs with Python – see more about it in Section 2.2.2. Its efficiency and inherent support for asynchronous operations was a primary factor in its selection, ensuring the platform's scalability and responsiveness, critical for handling concurrent requests from both mobile and desktop clients.

The core API provides interactive documentation via Swagger UI at `/api/v1/docs`. This setup facilitates easy exploration and understanding of the API's capabilities for developers.

5.2.1.1 API Layer Architecture

These backend services are structured in a layered architecture to promote modularity, separation of concerns and maintainability. This architecture ensures that changes in one layer have

minimal impact on others, facilitating development and scaling. Figure 5.10 describes the flow of a request.

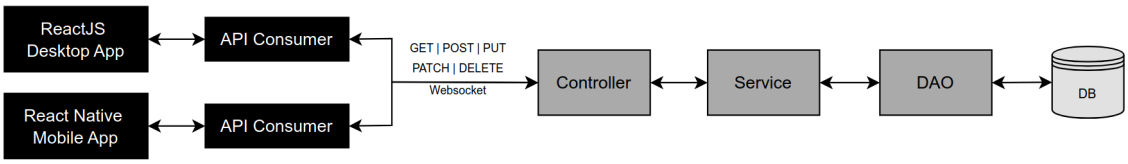


Figure 5.10: Full-Stack representation - Request Flow.

The primary layers involved in handling an API request are as follows:

- **ReactJS Desktop App**: This represents the client-side application specifically for desktop users, initiating requests to the backend.
- **React Native Mobile App**: This represents the client-side application specifically for mobile users, also initiating requests to the backend.
- **API Consumer**: This layer in both the desktop and mobile frontends is responsible for making HTTPS requests (`GET`, `POST`, `PUT`, `PATCH`, `DELETE`), using **Axios** and managing WebSocket connections to the backend API. It acts as an intermediary, abstracting the raw API calls from the main frontend logic.
- **Controller**: In the backend, the Controller (implemented using FastAPI Routers) is the entry point for incoming API requests. It receives requests from the API Consumer, performs initial validation and delegates the business logic to the Service layer. Controllers are responsible for mapping URLs to specific functions and handling HTTPS methods.
- **Service**: The Service layer encapsulates the core business logic of the application. It receives data from the Controller, orchestrates operations across multiple components (e.g., interacting with the database, calling external services) and applies application-specific rules. This separation ensures that business logic is centralized and reusable.
- **DAO (Data Access Object)**: The DAO layer is responsible for abstracting the data persistence details. It provides a clean API for the Service layer to interact with the database without needing to know the underlying database technology. Each DAO manages operations for a specific entity (e.g., UserDAO, IncidentDAO).

This layered approach ensures a clear flow of data and responsibility.

5.2.1.2 Middleware and Cross-Origin Resource Sharing (CORS)

To enhance API robustness and security, two middleware components are implemented:

- **Request Logging Middleware**: A custom HTTPS middleware is applied to log every incoming request's method and URL, along with the corresponding response's status code. This provides essential operational insights for monitoring and debugging.
- **CORS Middleware**: This middleware is crucial for enabling secure communication between the backend and its frontend applications, which might be hosted on different origins, ensuring that the mobile and desktop applications can interact with the API without encountering cross-origin security issues.

5.2.1.3 API Modularization and Routing

The API is organized into logical modules using FastAPI's `APIRouter` to maintain a clean, maintainable and scalable codebase. Each router handles a specific domain of the application's functionality. All API endpoints are prefixed with `/api/v1/`, for clear versioning. The routers described below are integrated into the main FastAPI application.

- `users_router`: Manages all user-related operations.
- `occurrences_router`: Handles the submission and retrieval of incident occurrences.
- `incidents_router`: Deals with incident clusters, their status updates and associated data.
- `auth_router`: Governs user authentication, registration and session management.
- `organizations_router`: Provides endpoints for organization-specific data, such as categories.
- `photos_router`: Facilitates the upload and retrieval of incident photos.
- `videos_router`: Manages the upload and retrieval of incident videos.
- `ws_router`: Establishes WebSocket connections for real-time data streaming.

5.2.1.4 Key API Endpoints and Functionalities

The following outlines the primary functionalities exposed through the API endpoints.

Auth Endpoints (`/api/v1/auth`)

- `POST /log-in`: Authenticates a user or operator and issues access and refresh tokens.
- `POST /sign-up`: Initiates the user registration process, sending a confirmation email.
- `POST /sign-out`: Logs out the current user and clears authentication tokens or cookies.
- `GET /users/me`: Retrieves the profile details of the authenticated user.
- `PUT /user-profile`: Updates the authenticated user's profile information.
- `POST /update-code-confirmation`: Confirms a code for critical account operations.
- `POST /forgotten-password`: Initiates a password reset process.
- `POST /new-password`: Sets a new password after a successful reset confirmation.
- `DELETE /user-profile`: Deletes the authenticated user's account.
- `POST /code-confirmation`: Confirms an email code for new user registration
- `POST /resend-code`: Resends a confirmation code to a user's email.
- `GET /refresh-token`: Refreshes access and refresh tokens.

Incident Endpoints (`/api/v1/incidents`)

- `GET /map`: Retrieves a list of unresolved incidents for display on a map, filterable by category.
- `PATCH /{incident_id}/status`: Updates the status of a specific incident.
- `GET /list-by-time`: Lists incidents based on status, optional category and chronological order.
- `GET /{incident_id}/occurrences`: Retrieves all occurrences associated with a given incident.
- `GET /{incident_id}/suggestions`: Fetches details suggestions for an incident.
- `GET /check-nearby`: Identifies incidents near given geographical coordinates and vehicle orientation.
- `POST /process-video`: Processes a video from the autonomous vehicle to assess incident resolution.
- `GET /check-nearby/stats`: Retrieves statistics related to nearby incident checks.
- `GET /{incident_id}`: Provides detailed information about a specific incident.
- `GET /{incident_id}/videos`: Retrieves videos associated with a specific incident.

Occurrence Endpoints (`/api/v1/occurrences`)

- `GET /{occurrence_id}`: Fetches detailed information about a specific incident occurrence.
- `POST /pre-submission`: Allows for pre-processing of an occurrence (e.g., AI analysis of a photo) before final submission.
- `POST /`: Submits a new incident occurrence.

Organization Endpoints (`/api/v1/organizations`)

- `GET /categories`: Retrieves the list of incident categories configured for the authenticated operator's organization.

Photo Endpoints (`/api/v1/photos`)

- `GET /{photo_id}`: Downloads a specific photo associated with an incident occurrence.

User Endpoints (`/api/v1/users`)

- `GET /stats`: Provides statistics related to the current user's reported occurrences.
- `GET /occurrences-by-time`: Lists a user's occurrences based on status and time.
- `GET /occurrences-not-resolved`: Retrieves a user's unresolved occurrences for map display.
- `GET /email-notifications`: Retrieves the user's preference for email notifications.
- `PATCH /email-notifications`: Updates the user's preference for email notifications.

Video Endpoints (`/api/v1/videos`)

- `GET /{video_id}`: Retrieves an incident resolution assessment video from the autonomous vehicle.

WebSocket Endpoints (`/ws`)

- `GET /llm/{incident_id}`: Establishes a WebSocket connection to stream real-time updates related to LLM processing for a specific incident, leveraging Redis Pub/Sub for asynchronous messaging. This connection is established with the frontend mobile app, at the time the user is submitting an ocurrence.

5.2.1.5 Error Handling

The API utilizes FastAPI's `HTTPException` to signal various client-side and server-side errors, providing clear status codes and detailed messages (e.g., `400 Bad Request` for "Email already registered" or "Incorrect password", `403 Forbidden` for unauthorized access to resources). This ensures that the frontend applications can gracefully handle API responses and provide informative feedback to users.

This comprehensive set of API endpoints, combined with robust authentication and modular design, forms the backbone of the platform, enabling communication between the client applications and the backend services.

5.2.2 Database Integration

5.2.2.1 Column-Based (Cassandra)

For primary persistent storage of structured data, we employ **Apache Cassandra**, a highly scalable, high-performance, distributed NoSQL database. A more detailed description of this tool can be found in Section 2.2.3. Its column-based architecture and masterless design were chosen to ensure high availability and linear scalability, vital for managing FixAI's anticipated growth in incident reporting and monitoring. A core design principle was to guarantee service scalability from inception, supporting both individual users reporting numerous issues and the concurrent activity of a large user base upon deployment. Cassandra's selection was driven by its **High Write Throughput**, crucial for rapidly ingesting occurrence submissions from mobile users and autonomous systems. Its architecture is optimized for write-heavy workloads, ensuring efficient data ingestion without performance degradation. Furthermore, **Scalability and Availability** are paramount for an urban infrastructure platform, necessitating support for increasing users and incidents without downtime. Cassandra's distributed nature enables horizontal scaling and continuous availability even with node failures. Its **Always-On Architecture** and peer-to-peer distribution model provide fault tolerance, essential for a system operational 24/7.

Cassandra is utilized to store various critical data entities within the platform, forming the backbone of the system's operational data. This data is organized across three main keyspaces. The **Auth keyspace** manages all authentication and user management data, including user and operator credentials (e.g., email, hashed passwords, user/operator IDs, creation timestamps), along with refresh tokens and confirmation codes. The **H3 Index keyspace** is dedicated to geospatial data, storing information for both organizational regions and incident regions, crucial for location-based functionalities. The central **App Data keyspace** holds the primary application data. This includes user profiles, which store user-specific information such as name and email notification preferences; organization data, encompassing details like organization name, language settings and specific incident categories configured for each organization, along with counts for pending, in-progress and resolved issues within those categories.

Core to this keyspace are the incident and occurrence records. Incident clusters store comprehensive information such as main category, description, centroid coordinates and aggregate counts of occurrences. Individual incident reports, or occurrences, are detailed with geolocation, description, category and associated user and incident identifiers. These records are optimized for efficient retrieval based on status, category and chronological order and occurrences are also indexed for user-specific status tracking. Additionally, this keyspace manages the linking of incidents to their constituent occurrences. Finally, it also stores metadata for media, specifically unique identifiers referencing photos and videos that are stored as objects in MinIO.

The data models within Cassandra are primarily designed to optimize query performance for specific access patterns identified through the API endpoints. This often necessitates **data duplication** across multiple tables to achieve efficient retrieval without complex join operations, a common trade-off in NoSQL databases for read optimization. For instance, data is structured to efficiently support retrieval of incidents by status and category for the desktop dashboard's listing functionalities and to quickly fetch all occurrences related to a particular incident ID. Cassandra's strong support for composite primary keys allows for precise and efficient data retrieval based on various query parameters, crucial for the system's real-time monitoring and reporting capabilities.

5.2.2.2 Object Storage (MinIO)

For the storage of unstructured data objects such as photos and videos associated with incidents, we utilize **MinIO** (2.2.4). The primary motivation for choosing MinIO was the clear requirement for a robust object storage solution that offered a cost-effective and operationally simpler alternative to cloud services like Amazon S3, while retaining a compatible interface. Given that our platform did not require the complexity of a hierarchical structured filesystem, a flat object storage model was highly suitable. MinIO provided the optimal solution, offering a high-performance, S3-compatible API that significantly eases the potential future migration to Amazon S3, should scalability demands exceed on-premises capabilities. This design ensures that large binary data, such as incident photos and videos submitted by citizens and autonomous vehicles, are efficiently stored and retrieved, complementing Cassandra's role in managing structured metadata.

5.2.3 H3 Integration for Spatial Operations

H3 integration, explained in Section 2.2.7, is the core of our solution for managing and analyzing spatial data. Its efficiency in geographic indexing and finding neighboring areas is vital for how well our application performs and grows. This section explains how the H3 system fits into our overall design, from preparing organizational areas to handling incidents as they happen, making sure we have a strong base for all our spatial tasks.

5.2.3.1 Organization Indexing

Organization indexing is key to connecting complex geographic areas with H3 identifiers. Our system pre-calculates and saves all the hexagons that belong to a specific organization. We use resolution 13, which provides a lot of detail. To save storage space and reduce data complexity, we use a compaction method. This method lets us use smaller H3 resolutions in larger geographic areas where less detail is acceptable. This cuts down on data size and makes queries faster. The steps for organization indexing are shown in the Figure 5.11:

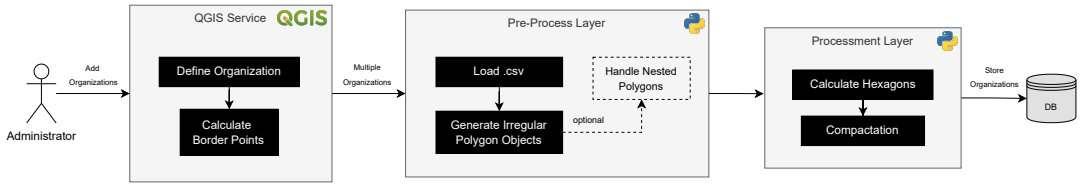


Figure 5.11: Add Organization Workflow.

As depicted in Figure 5.11, the process begins with an administrator adding organizations. This input then flows into a QGIS Service where the organization's geographic boundaries are defined and border points are calculated. The data subsequently moves to a Pre-Process Layer, which handles several operations: loading data (e.g., from CSV files), optionally managing nested polygons and generating irregular polygon objects suitable for H3 processing. Finally, the Processment Layer calculates the H3 hexagons for these polygons and compacts them to optimize storage and query efficiency before the organizational data is stored in the database (DB).

This process ensures that organizations are accurately and efficiently represented in the H3 structure, allowing for quick and scalable spatial operations.

5.2.3.2 Incident Indexing

Incident indexing is essential for effectively managing and analyzing events in real-time. Each incident is linked to an exact geographic spot (latitude, longitude), which is then mapped to an H3 hexagon at resolution 13. This approach means several incidents can occur within the same hexagon, making it easier to group and analyze them spatially. H3's ability to easily find nearby cells, thanks to its neighborhood traversal feature, as explained in Section 2.2.7.2, is fundamental to this process. Also, because H3 uses projections based on an Icosahedron, Section 2.2.7.4, there's very little distortion in land areas. There are only minor issues in some sea zones, similar to any Dymaxion-based projection. This gives us confidence that we can consistently group problems in equally sized areas worldwide, for example, by location and other factors, which we'll explore in the next section.

5.2.4 Asynchronous Job Processing

This section delves into the intricate flow of urban occurrence reporting within the FixAI mobile application. It specifically outlines the asynchronous job processing mechanisms that underpin the platform, from leveraging multimodal Large Language Models (LLMs) for generating incident descriptions, categories and severity, to intelligently clustering related reports and dynamically updating incident information over time. An overarching view of this comprehensive process is provided in Figure 5.12.

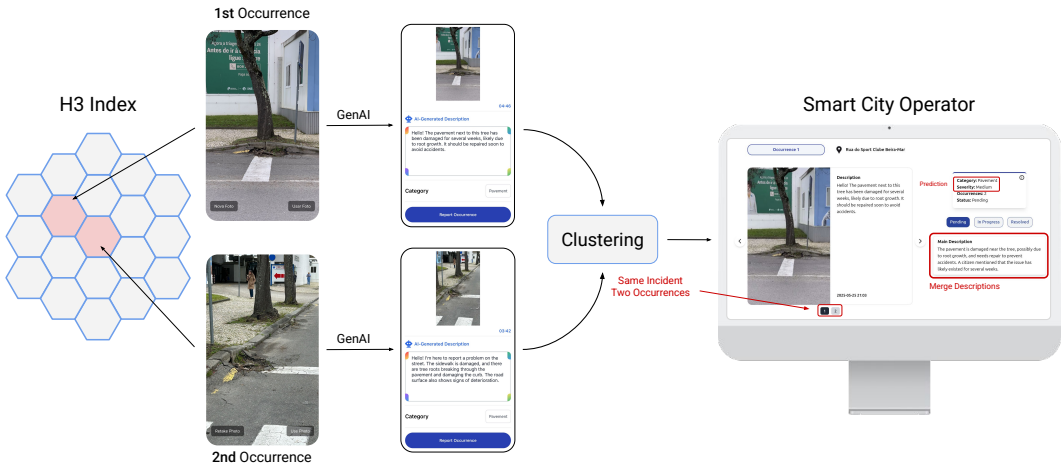


Figure 5.12: Overall Incident Processing Flow in FixAI, showcasing H3 Indexing, AI-Driven Classification, Clustering Related Reports and Incident Updated Details Based on New Occurences.

Following this comprehensive overview of the asynchronous processing pipeline, the subsequent sections will detail each major component, beginning with the initial incident reporting process by the citizen.

Report Flow The process, illustrated in Figure 5.13, outlines how a citizen initiates and completes an incident report via the mobile application, encompassing frontend interaction, backend data processing and AI-driven classification.

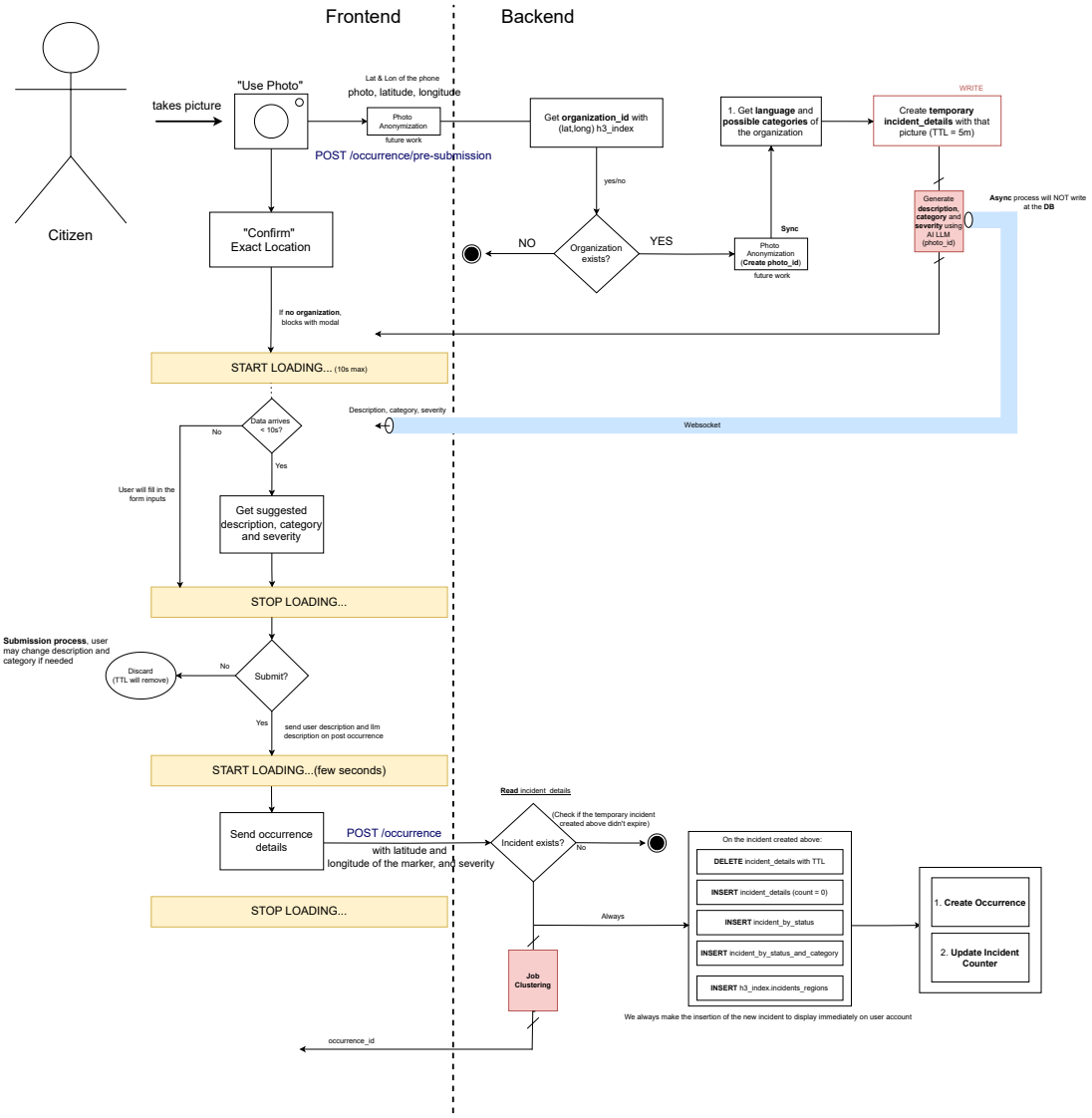


Figure 5.13: User Reports an Occurrence Flow Description.

Phase 1: User Initiates Report and Pre-Submission The reporting process commences with the citizen taking a photograph of the urban issue. Concurrently, the system automatically captures relevant metadata, including the precise geographical coordinates (latitude and longitude). This preliminary data, comprising the photograph and its geolocation, is then transmitted to the backend via a `POST /occurrence/pre-submission` API endpoint.

Upon receiving this request, the backend undertakes an initial processing step. It attempts to identify an associated `organization_id` based on the provided geolocation. A critical check ascertains if a corresponding organization is registered within the platform's database. Should an organization exist, the process seamlessly proceeds. Following this, the system retrieves essential organizational data, specifically the language settings and the pre-defined categories relevant to that organization (e.g., "urban drainage," "traffic signs," "potholes").

Crucially, a temporary incident is then instantiated on the backend, linked to the captured picture and assigned a short **Time-To-Live (TTL) of 5 minutes**. Immediately thereafter, an asynchronous process, designated as **"Job 1 - Generate description, category and severity,"**

is triggered. This job leverages the Gemini multimodal LLM to analyze the submitted image and generate a contextual description, categorize the incident and assess its severity. The detailed mechanics of Job 1 will be elaborated subsequently. Once Job 1 successfully processes the data (i.e., generates the description, category and severity), the backend proactively establishes a WebSocket connection to the frontend to transmit this AI-generated information.

Phase 2: Frontend User Interaction and AI Response The frontend also prompts the user to confirm the exact location of the incident. After this, the application displays a loading state. If the AI-generated "Description, category and severity" are not received within a strict **10-second window**, the system intelligently allows the user to manually input the description and select a category, ensuring continuity of the reporting process even in the event of an AI processing delay or failure.

Once the user reviews the displayed (either AI-generated or manually entered) information, they are prompted to confirm the submission. Should the user decide to discard the report, the temporary incident and any associated data are promptly removed from the backend, preventing unnecessary data retention.

Phase 3: Final Submission and Backend Processing Upon the user's confirmation, the frontend transmits the finalized occurrence details—including the confirmed latitude, longitude of the marker and severity—to the backend via a `POST /occurrence` API endpoint. This constitutes the definitive submission of the incident report.

The backend then rigorously processes this final submission. A vital preliminary check verifies the validity of the temporary incident created in Phase 1; if its TTL has expired, the user is required to re-initiate the reporting process. Assuming the temporary incident is still valid, the system proceeds with the incident management logic. For every occurrence reported, a new occurrence record is created and the overall incident counter is updated (as a single incident can encompass multiple related occurrences). Simultaneously, the temporary incident is removed and the permanent data is committed to the appropriate tables within the database.

Immediately following this, **"Job 2 - Clustering"** commences as a second asynchronous process. This job is pivotal for identifying and grouping related occurrences, ensuring that city operators receive a consolidated view of similar reports. This clustering mechanism is instrumental in avoiding redundancy and significantly enhancing operational efficiency, a process that will be explained in greater detail in a later section.

5.2.4.1 Job 1 - Generate Description, Category and Severity

This asynchronous job is initiated as part of Phase 1 of the incident reporting flow, immediately following the creation of a temporary incident. Its primary function is to leverage a Large Language Model (LLM) to analyze the submitted photograph and automatically generate a contextual description, classify the incident into a predefined category and assess its severity. This job exemplifies the core AI integration within the FixAI platform.

The process, illustrated in Figure 5.14, begins with the **LLM Producer** (part of the Business Logic Layer) creating a message that contains essential data: the `Photo ID` (identifying the image stored in MinIO), the `Incident ID` (referencing the temporary incident), the `Org Categories` (pre-defined categories specific to the organization), the `Org Language` (the language set for the organization) and the `TTL` (Time-To-Live for the temporary incident). This message is then sent to the Kafka broker.

Subsequently, an **LLM Consumer** instance (running on the Worker01 VM) retrieves this message from the Kafka broker. The LLM Consumer's first action is to use the provided `Photo ID` to download the actual image from MinIO, which serves as the high-performance file storage.

Once the image is retrieved, the LLM Consumer constructs a tailored prompt for the **LLM** (specifically, Gemini, as detailed in the architecture). This prompt intelligently incorporates the downloaded `Photo`, the relevant `Categories` from the organization and the specified `Language`. The LLM then processes this comprehensive prompt to perform its analytical task.

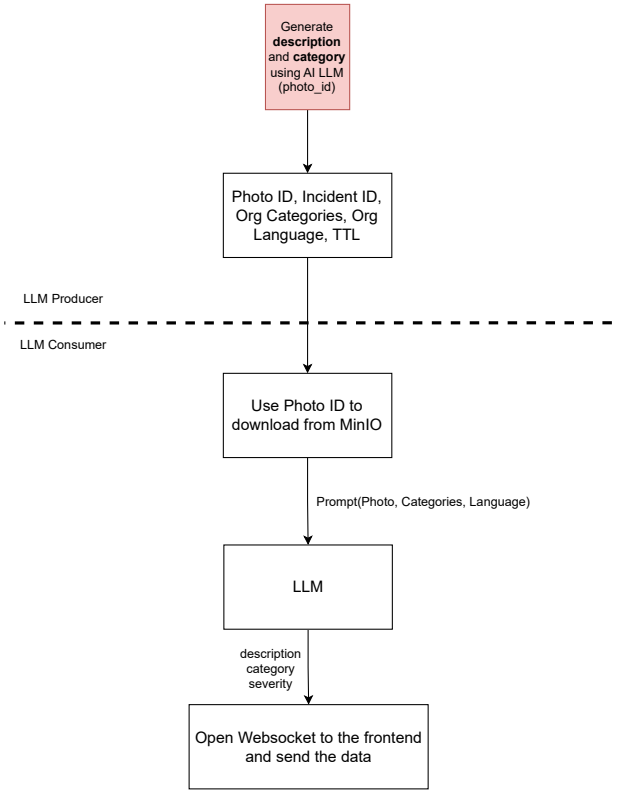


Figure 5.14: Flow Diagram for Job 1 - Generating Incident Description, Category, and Severity using LLM.

The output from the LLM consists of the generated **description**, the assigned **category** and the assessed **severity** of the incident. Upon receiving this structured response from the LLM, the LLM Consumer's final step is to establish a WebSocket connection directly to the frontend. This connection is used to transmit the newly generated description, category and severity data back to the user interface in real-time, enabling the citizen to review the AI's output before finalizing their report.

5.2.4.2 Job 2 - Cluster Related Reports

This asynchronous job is initiated immediately following the final submission of an occurrence (as detailed in Phase 3 of the report flow). Its primary objective is to group newly reported occurrences with existing incidents that are spatially and semantically similar, thereby reducing redundancy for city operators and enhancing management efficiency.

The clustering process, depicted in Figure 5.15, begins by compiling two distinct lists of incident IDs. The first list comprises **incident IDs from neighboring hexadecimal cells** (leveraging the H3 geospatial indexing framework), while the second lists **incident IDs located within the same hexadecimal cell** as the new occurrence. These two lists are then concatenated to form a comprehensive list of potential related incident IDs.

A critical check ascertains if this combined list of incident IDs has a length greater than zero. If the list is empty (indicating no proximate existing incidents), the process bypasses further clustering. However, if potential incidents exist, the system proceeds to loop through all incidents to stay only with the incidents with the same category. This filtering step generates an

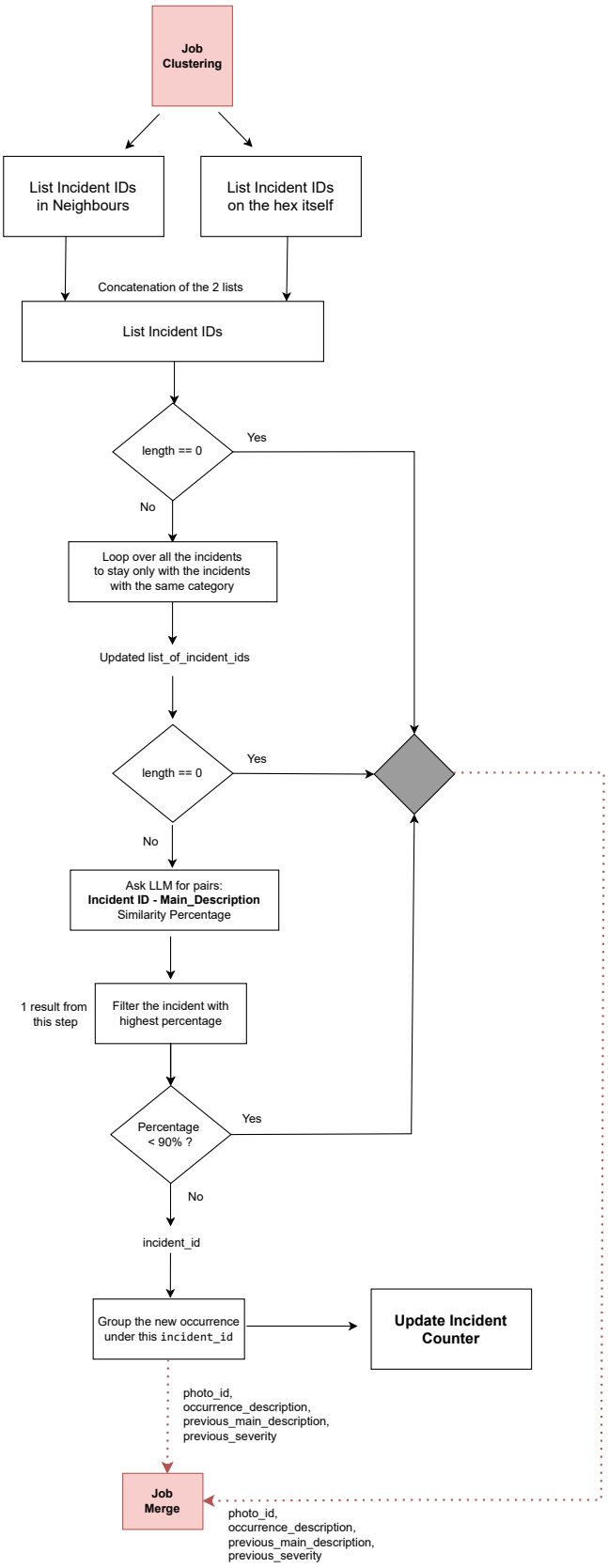


Figure 5.15: Flow Diagram for Job 2 - Cluster Related Reports.

`updated_list_of_incident_ids`.

Following this, another check verifies if this refined list is empty. If it is, the clustering process again bypasses the similarity assessment. If the list contains incident IDs, the system invokes the LLM (Large Language Model) to **assess the semantic similarity** between the new occurrence's description and the descriptions of the incidents in the refined list. The LLM returns a "Similarity Percentage" for each pair. From these similarity results, the system filters for the **single incident with the highest similarity percentage**. A crucial threshold is then applied: if this highest percentage is **less than 80%**, it is determined that the new occurrence is not sufficiently similar to any existing incident and it proceeds as a distinct incident. Conversely, if the similarity percentage is **80% or greater**, the new occurrence is considered related. In this scenario, the new occurrence is **grouped under the identified existing incident**. This grouping also triggers an update to the overall incident counter for that existing incident.

Upon completion of the clustering logic, regardless of whether the occurrence was grouped or became a new incident, the process proceeds to trigger **"Job Merge"**. This job will be explained in greater detail in the next subsection.

5.2.4.3 Job 3 - Update Incident Description and Severity

This asynchronous job is a critical final step in the incident processing pipeline. It is triggered after an occurrence has been successfully clustered with an existing one. The primary purpose of Job Merge is to intelligently update the main description and severity of an incident based on the latest incoming occurrence and the incident's historical data, ensuring the most accurate representation of the ongoing issue.

The process, as depicted in Figure 5.16, begins with the job receiving a set of crucial inputs. These inputs include the `photo_id` of the newly processed occurrence, its `occurrence_description` (initially generated by Job 1), the `previous_main_description` and `previous_severity` of the incident to which this occurrence is now linked.

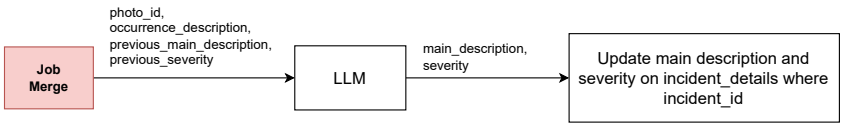


Figure 5.16: Flow Diagram for Job 3 - Update Incident Description and Severity.

These combined inputs are then fed into the **LLM** (Large Language Model). The LLM's task in this context is to synthesize this information. By considering both the new occurrence's details and the incident's prior state, the LLM generates a refined, updated `main_description` and `severity` for the incident. This allows the system to evolve the incident's summary as more related occurrences are reported, providing a dynamic and current overview.

Finally, the output from the LLM is used to **update** the `incident_details` record in the main database, ensuring that the centralized incident record accurately reflects the most current understanding of the urban issue, which is then visible to city operators.

5.3 Security Mechanisms

5.3.1 Tokens, Cookies, and Session Management

For the FixAI platform, we implemented a robust authentication and session management system leveraging **JSON Web Tokens** (JWTs) to secure user interactions. This application-level security is further enhanced by network-layer defenses such as **HTTPS encryption** and **reverse proxying**, the comprehensive details of which are elaborated in Section 5.5.1 on Kubernetes-based orchestration. This approach provides flexible authentication for both mobile and desktop applications.

5.3.1.1 Login Process

The authentication process begins when a user submits their credentials (e.g., username and password) to the Auth Server's `login` endpoint, as illustrated in Figure 5.17. Upon successful validation, the Auth Server issues a pair of JWTs: an Access Token, which is a short-lived token used to authenticate requests to protected resources, and a Refresh Token, a long-lived token used to obtain new access tokens once the current one expires. For the React Native Mobile App, these tokens are stored in secure local storage within the client device. Conversely, for the ReactJS Desktop App, the tokens are securely set as HttpOnly cookies, providing an additional layer of security by preventing client-side JavaScript access to these sensitive credentials. This strategic differentiation in token storage accommodates the distinct security postures and operational requirements of mobile and desktop environments.

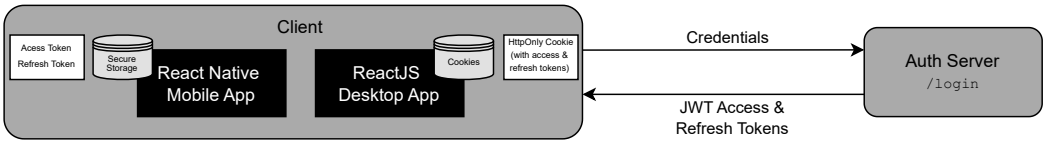


Figure 5.17: Security Login Process.

5.3.1.2 Successful Request Flow

Once authenticated, the client can make requests to the Resource Server by including the Access Token in the request headers (using a Bearer token in the `Authorization` header, in the case of the mobile app, and a cookie, in the case of the desktop app). As depicted in Figure 5.18, the Resource Server does not directly validate the token's authenticity or expiration. Instead, it forwards the Access Token to the Auth Server's internal `verify` endpoint. This separation of concerns ensures that token validation logic is centralized and managed exclusively by the Auth Server. If the Access Token is valid and unexpired, the Auth Server responds with a token verification success, allowing the Resource Server to process the request and return the requested data with a `200 OK` status. This design minimizes the Resource Server's knowledge of authentication mechanisms, enhancing security and maintainability.

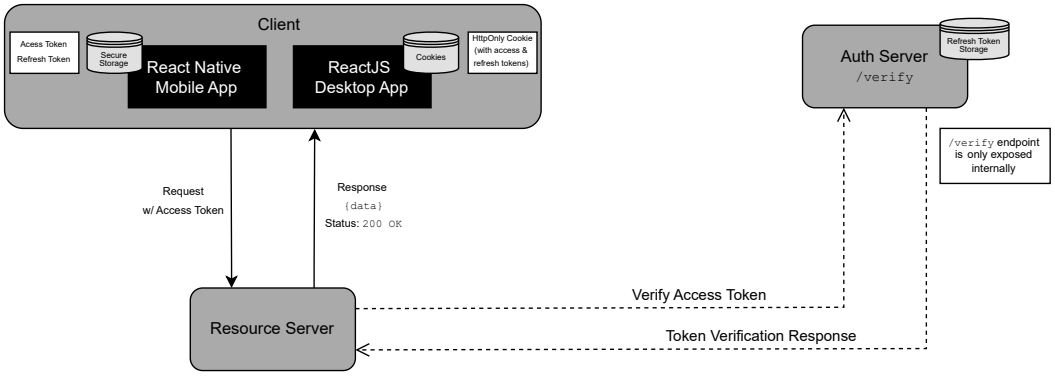


Figure 5.18: Security Successful Request Flow.

5.3.1.3 Failed Request Handling

In scenarios where a client's Access Token is invalid or has expired, the request handling deviates as shown in Figure 5.19. When the Resource Server forwards the invalid Access Token to the Auth Server's internal `verify` endpoint, the Auth Server detects the token's expiry or invalidity.

It then responds to the Resource Server with a "Token Expired Response." Consequently, the Resource Server returns a `401 Unauthorized` status to the client. This error handling mechanism clearly signals to the client that the current Access Token is no longer valid, prompting a token refresh action without exposing sensitive authentication details.

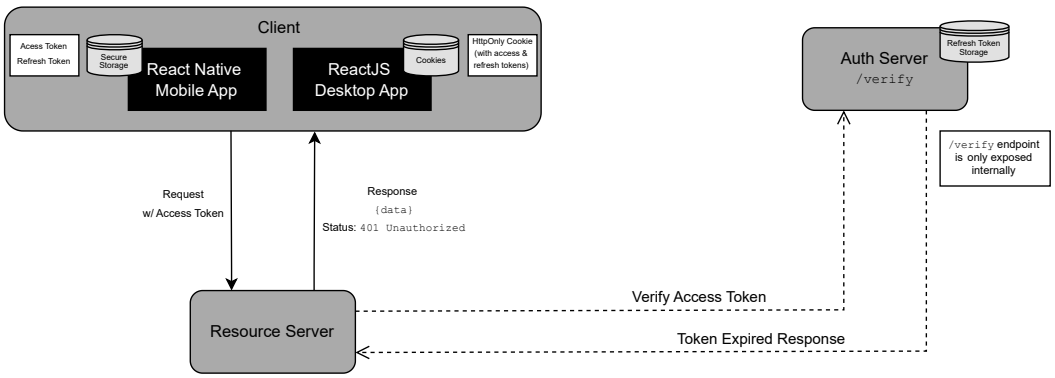


Figure 5.19: Security Failed Request Handling.

5.3.1.4 Access Token Refresh Mechanism

To maintain continuous user sessions and mitigate the risks associated with long-lived Access Tokens, we implemented an automated token refresh mechanism, as detailed in Figure 5.20. When an Access Token expires, the client utilizes the Refresh Token to request a new Access Token from the Auth Server's `refresh` endpoint. The Refresh Token, securely stored alongside the Access Token (in secure storage for mobile, or as an HttpOnly cookie for desktop), is sent to the Auth Server. The Auth Server validates the Refresh Token against its Refresh Token Storage. If the Refresh Token is valid and unexpired, the Auth Server issues a new Access Token, which is then returned to the client. By implementing this process, we allow users to remain logged in without repeatedly re-entering credentials, significantly improving user experience while upholding security best practices by rotating Access Tokens periodically.

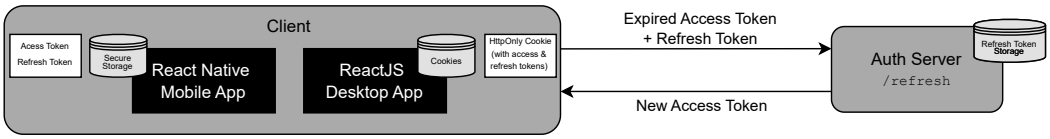


Figure 5.20: Security Access Token Refresh Mechanism.

5.3.1.5 Frontend Session Management

Authentication on the frontend is uniformly managed through a dedicated `AuthContext`, which centralizes logic for login, logout, and session management.

For the React Native Mobile App, JWT tokens issued by the backend upon login are securely stored using the `expo-secure-store` library. Initially, `async-storage` was employed for token persistence, but it was subsequently replaced by `expo-secure-store` due to its capability to encrypt data at rest, offering enhanced protection for sensitive user credentials. On application launch, the authentication context performs a check for a stored access token to determine whether an existing user session should be restored. A preconfigured `axios` instance, integrated within the context, automatically attaches the Access Token to every outgoing HTTP request. Furthermore, it leverages response `interceptors` to transparently manage session expiration. When a request elicits a `401 Unauthorized` response, the interceptor ascertains if the error is attributable to token

expiration. If so, it automatically attempts to refresh the access token by dispatching a request to the `refresh` endpoint, with the stored Refresh Token temporarily included in the request header. Should the token refresh prove successful, the newly issued tokens are securely updated in storage, and the original failed request is automatically retried. Conversely, if the refresh operation fails, indicating an invalid or expired Refresh Token, the user is redirected to the login page to re-authenticate. The encapsulation of this intricate logic within the authentication context provides a clean and centralized interface for all authentication-related operations, including login, logout, and token management.

Similarly, in the Desktop App, authentication is also handled via a dedicated `AuthContext`.

This consistent approach across both mobile and desktop applications delivers a secure authentication experience throughout the entire FixAI platform.

5.3.1.6 Benefits of this Approach

This JWT-based authentication strategy offers substantial benefits across both user experience and security, pivotal for the effective operation of FixAI.

From a **user experience** perspective, the implementation of Refresh Tokens enhances session continuity. Users are not subjected to frequent re-authentication prompts, even after extended periods of inactivity, as the system renews Access Tokens in the background. This smooth, uninterrupted access fosters a more intuitive and less frustrating user journey.

The security advantages of this architecture are multifaceted. The core benefit stems from the **stateless nature of JWTs** on the Resource Server. By not requiring the server to store session information, the system gains inherent scalability, as any Resource Server instance can process any request without maintaining individual user states. This also reduces the server's memory footprint and minimizes the attack surface for session-related vulnerabilities. The use of **short-lived Access Tokens** significantly limits the exposure window if a token is intercepted, even if compromised, its utility is brief. Conversely, **long-lived Refresh Tokens**, when stored securely (e.g., in HttpOnly cookies for desktop applications, preventing client-side JavaScript access via Cross-Site Scripting (XSS) attacks), allow for extended user sessions without the security risks associated with long-lived Access Tokens. The **separation of concerns**, where the Resource Server delegates token validation to the Auth Server, further strengthens security. This centralization of authentication logic reduces complexity on the Resource Servers, limiting their responsibilities and potential vulnerabilities. It also facilitates easier maintenance and updates to authentication mechanisms without impacting the core application logic.

In general, this approach provides a scalable, secure and user-friendly authentication foundation for FixAI.

5.3.2 Role Based Access Controll (RBAC)

A Role-Based Access Control (RBAC) system is implemented to ensure that users can only access resources and functionalities relevant to their assigned roles and permissions. This granular control is critical for maintaining data integrity and system security within a collaborative platform serving diverse user types. The system defines two primary roles: **Citizen** and **Operator**.

The **Citizen** role is assigned to general users interacting with the mobile application. Citizens are granted permissions strictly limited to their own reported incidents and occurrences. For instance, a citizen is authorized to view the details and track the status of occurrences they have personally reported, but they are explicitly excluded from accessing reports by other users, a restriction directly informed by discussions with our main stakeholders, the city council workers, and cannot manage any organizational data.

The **Operator** role is designated for employees who interact with the desktop dashboard. Operators possess broader access rights pertinent to urban infrastructure management. Crucially, their access is scoped to the specific organization (e.g., a municipality, a private organization, etc.) they belong to. An Operator can view, update, and manage incidents and occurrences

that fall under their organization's purview. They are, however, restricted from accessing data or performing operations related to other organizations.

The enforcement of RBAC is deeply integrated into the API's endpoint logic. JWT Access Tokens in FixAI are structured to carry essential claims about the authenticated user within their payload. Specifically, Access Tokens include a user identifier, an email address, and optionally an organizational identifier for Operator users, along with a token type ("access"). This data is embedded into the token during its creation, where for Citizen users, the organizational identifier field is explicitly set to a null value, differentiating their role directly within the token's claims. When a request for specific resource details arrives at a data retrieval endpoint, the system determines access rights based on these claims within the authenticated user's token. If the token contains an organizational identifier, indicating an Operator, the system verifies that the requested resource's organizational affiliation matches the organizational identifier embedded in the authenticated Operator's token. Conversely, if the organizational identifier is absent in the token (i.e., it holds a null value), indicating a Citizen, the system verifies that the resource's user identifier matches the user identifier embedded within the Citizen's token. Should these conditions not be met, an `HTTPException` with a `403 Forbidden` status is raised, effectively preventing unauthorized access. This is one example of the verifications that we have implemented. This dynamic validation ensures that every data access request is authorized not just by a valid token, but also by the user's role and their specific scope within the system, adhering to the principle of least privilege.

5.4 Issue Automatic Resolution

In order to streamline the management of reported incidents, this module aims to provide an automated way to verify whether an issue has already been resolved. The proposed solution involves the use of an autonomous IoT device capable of capturing video footage of the incident area and sending it to the backend for analysis. For validation purposes, we used the PIXKIT device from ATCLL.

Once the video is received by the backend, it is processed using a Large Language Model (LLM) that compares the current visual evidence with the original report and determines whether the problem has been effectively resolved. If so, a resolution suggestion is sent to the organization's operator via the desktop application, accompanied by the recorded video. The operator can then confirm the result and officially mark the incident as resolved.

This section describes the backend process and explores different solutions for verifying incident resolution, including a mobile client implementation and integration with the PIXKIT device.

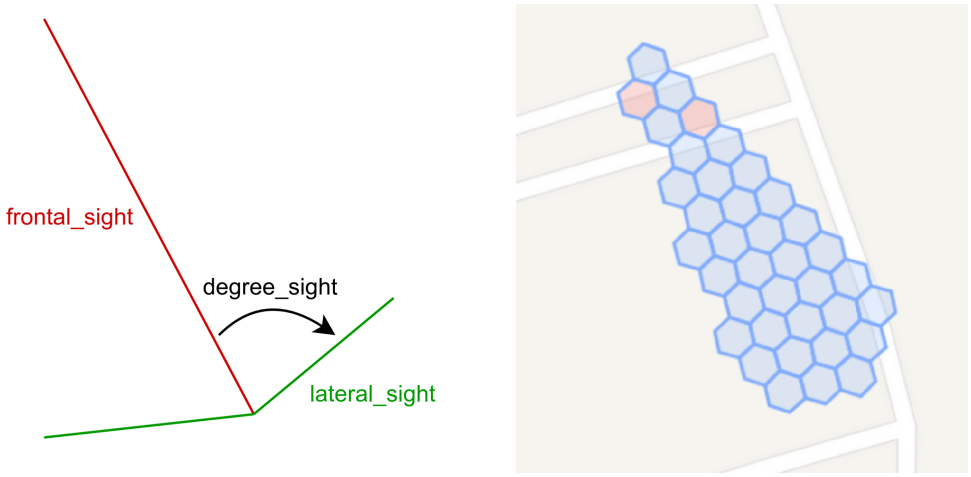
5.4.1 Backend Process Overview

To support both the mobile client and the PIXKIT device, the backend exposes a set of REST endpoints that enable the detection and resolution verification of previously reported incidents. This functionality is central to the system's goal of automating incident closure suggestions based on real-world visual input.

The backend includes two key endpoints:

- `GET /incidents/check-nearby` – Accepts geolocation data and vision parameters such as `latitude`, `longitude`, `heading`, `frontal_sight`, `lateral_sight`, and `degree_sight`. It returns a list of UUIDs corresponding to incidents located within the current field of view of the device.

- `POST /incidents/process-video` - Receives an `incident_id` and a recorded video file. It creates an asynchronous job that sends the incident's image, category, and description, along with the captured video, to a Large Language Model (LLM) for analysis. The LLM compares the current visual evidence against the original report to determine if the issue appears resolved.



(a) Sight Representation. (b) Real-Time H3 Visualization Website.

Figure 5.21: Camera Field of View and Incident Mapping in the System.

To determine whether an incident is in view, the system relies on geometric calculations that simulate the device's field of vision. Based on the device's location and heading, three key points are computed to define a triangular area:

- p_0 is the current geolocation (`latitude`, `longitude`),
- p_1 and p_3 represent the lateral limits of the camera's field of view,
- p_2 represents the frontal limit, based on the forward sight.

These points are computed using geodetic projections that take into account the heading and sight distances. The system then queries all incidents located within this defined area. The sight parameters are dynamic and adjust according to the speed of the device, for instance, a higher speed narrows the lateral sight while extending the frontal sight, simulating a natural tunnel-vision effect to avoid capturing unrelated areas.

Figure 5.21 visually demonstrates this concept. On the left, it illustrates the geometrical projection of the device's frontal, lateral, and angular field of view. On the right, it shows the same area translated into H3 hexagonal indexes, with red hexagons highlighting the zones containing active incidents.

5.4.2 Existing Solutions for Video Transmission

Several approaches were explored for transmitting video from the client to the backend to enable automatic incident resolution analysis. The goal was to provide real-time or near-real-time visual data to the server, which could then be processed using a Large Language Model (LLM) to verify whether an incident had been resolved.

The initial implementation used the `Gorilla WebSocket` library in Go, but only for receiving real-time geolocation data from the client device. To complement this, we attempted to implement video streaming using the RTMP (Real-Time Messaging Protocol) protocol. A media server was configured to receive and process RTMP streams sent from the client. However, this approach presented several limitations. Processing live RTMP streams required significant runtime resources on the server, including real-time decoding and handling of continuous data streams. This increased the complexity of our infrastructure and added unnecessary latency and fragility to the system, especially considering the short duration and purpose of the video clips.

We also considered WebRTC, a peer-to-peer video communication protocol widely used in real-time applications. However, WebRTC is primarily designed for browser-based or device-to-device

streaming. In our system architecture, which is inherently client-server and requires server-side processing and storage, the peer-to-peer nature of WebRTC offered limited advantages.

After evaluating these alternatives, we concluded that the most efficient and robust solution for our current needs was to record a short video clip (e.g., 5 seconds) and send it directly to the backend via a standard REST API endpoint. This allowed for asynchronous processing, simple error handling, and compatibility with both the mobile app and future IoT integrations. The endpoint `/incidents/process-video` accepts the video file and initiates a background task that forwards it to the LLM for resolution analysis.

In the case of the PIXKIT integration, the video is processed directly on the device. This offloads computational work from the backend and allows the PIXKIT to act as a smart edge device, performing inference locally and sending only results or minimal metadata to the backend. This REST-based approach strikes a good balance between simplicity, reliability, and extensibility, making it a practical solution for both mobile and embedded clients in our system.

5.4.3 Smartphone Client

To validate the video-based incident resolution flow, a simple prototype was implemented directly in the FixAI mobile application. This client was responsible for capturing a short video and sending it to the backend for analysis.

The implementation was straightforward: a dedicated screen was added to the mobile app that activated the device's camera. When a user was detected to be in proximity to a reported incident, based on their GPS coordinates and heading, the app automatically recorded a short 5-second video clip and sent it to the server using a REST request.

Location data was accessed using the `expo-location` library, which also provided orientation (heading). The camera functionality was handled by `expo-camera`. This combination allowed the app to determine the user's current position, estimate their field of vision, and detect when they were approaching an incident zone.

This prototype served as an early proof of concept and played a key role in validating the complete workflow: from capturing footage in the field to backend processing and LLM-based analysis of whether the incident had been resolved.

5.4.4 ATCLL (PIXKIT) Integration

In addition to the mobile client prototype, a more robust and autonomous solution was developed in partnership with ATCLL using the PIXKIT device. This integration aimed to evaluate the feasibility of real-time incident verification in the field using edge computing and sensor data.

Figure 5.22 shows the complete hardware setup. The PIXKIT device includes:

- A V2X camera capable of RTSP video streaming.
- A GNSS module for high-precision geolocation.
- An onboard processing unit running both an RTSP server and an MQTT broker.

The system operates as follows: the PIXKIT continuously publishes MQTT messages containing its geolocation metadata, including latitude, longitude, speed, and heading, to a specific topic. Our system subscribes to this topic, extracting the relevant data to compute the current position and estimated field of view of the device using the same vision cone model discussed earlier.

With this geospatial data, the system queries the `/incidents/check-nearby` endpoint to determine if any open incidents lie within the PIXKIT's viewing area. If any are detected, the system triggers a short recording session from the camera stream (via RTSP). A 5-second video clip is extracted, stored locally, and immediately sent to the backend via a REST endpoint for LLM-based analysis.

The RTSP stream is handled by a dedicated module that keeps the connection alive and manages concurrent video writers to record only when required, avoiding unnecessary overhead. This

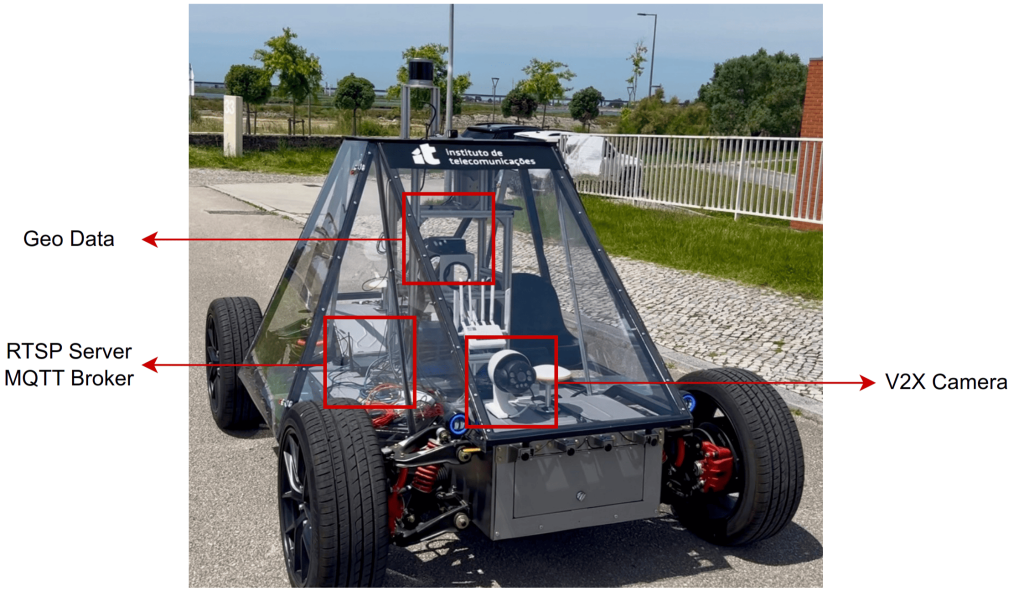


Figure 5.22: PIXKIT Physical Components.

lightweight approach allowed processing to be done on the PIXKIT itself, significantly reducing bandwidth and latency, since only short video clips are transmitted.

Figure 5.23 illustrates the full architecture of the system, highlighting the RTSP video stream, MQTT location updates, and REST communication with the backend.

Finally, a proof-of-concept demonstration was developed to visualize the functionality in action. As seen in Figure 5.24, the figure is divided into three parts:

- **Top-left:** A physical photo of the PIXKIT device used in the field.
- **Bottom-left:** The actual camera feed captured by the PIXKIT, showing its perspective.
- **Right:** The real-time representation of the PIXKIT's location in the system, rendered with H3 hexagons. A red hexagon highlights the position of a previously reported incident, currently under re-evaluation.

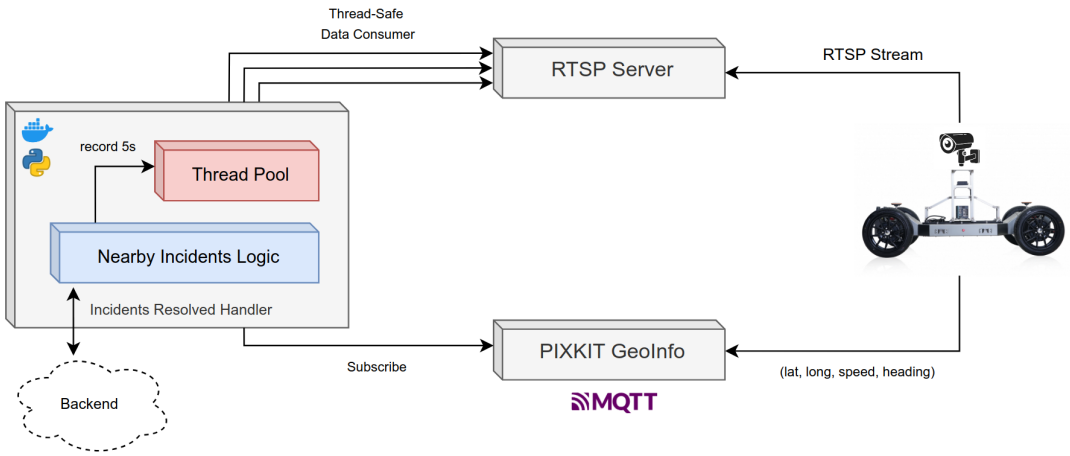


Figure 5.23: PIXKIT Integration Diagram.

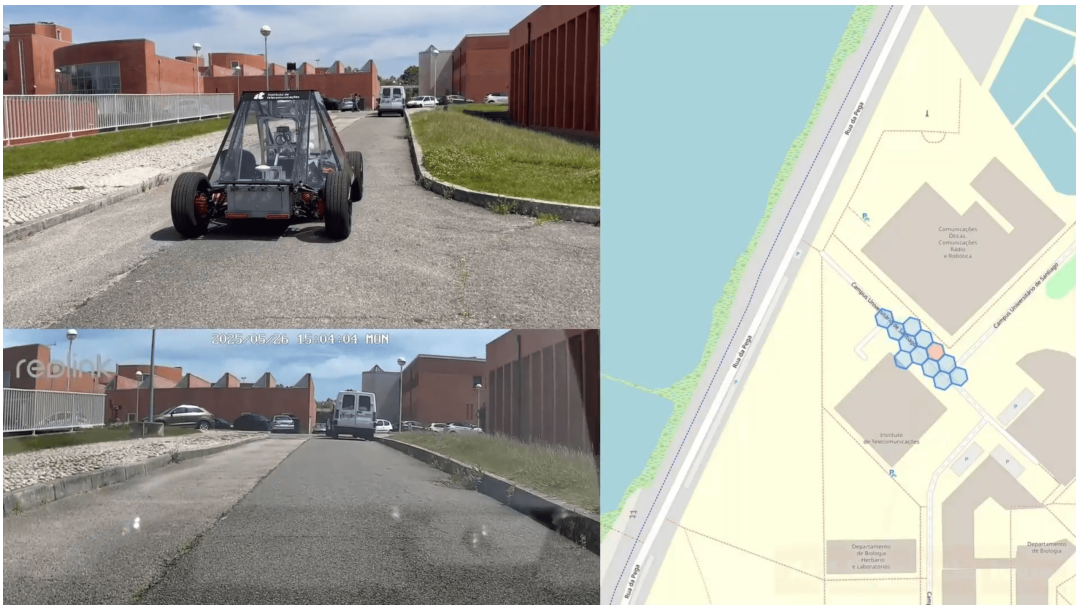


Figure 5.24: Demonstration of PIXKIT Camera, Vision and Hexagon-Based Incident Evaluation (Video Demo: **here**).

This integration proved the system's potential to autonomously verify incidents using low-power IoT hardware, combining computer vision and geospatial analysis in a seamless pipeline.

5.5 Infrastructure and Deployment

The deployment of the system is orchestrated through a **Kubernetes-based infrastructure**, chosen to address the architectural priorities of **scalability**, **fault tolerance**, **service isolation**, and **operational flexibility**. Given the system's design, composed of stateless backend services, event-driven consumers, and distributed stateful components, Kubernetes offers a robust and declarative environment for managing these diverse workloads.

A key motivation for selecting Kubernetes lies in its native support for **horizontal scaling**, which is central to the system's ability to dynamically adjust resource allocation based on demand. Stateless components, such as the **backend** and **LLM consumers**, benefit from Kubernetes' replica management and service abstraction mechanisms, enabling seamless scaling without modifying application logic. This aligns directly with the modular architecture of the system, as described in Section 4.1.1.

Moreover, the inclusion of distributed storage technologies such as **Apache Cassandra** and object storage systems required infrastructure capable of managing persistent volume claims, ordered pod deployment, and stable network identities. Kubernetes provides these guarantees through its `StatefulSet` and `StorageClass` primitives, which ensure **consistency**, **durability**, and **recoverability** of stateful services. This design decision is further supported by the distributed data model and write-intensive access patterns discussed in Section 2.2.3.

Kubernetes also brings operational advantages, such as **declarative infrastructure definitions**, **automatic service discovery**, **rolling updates**, and **resource monitoring**. These capabilities simplify the management of a multi-service system and enhance resilience against failure by enabling controlled restarts, load balancing, and self-healing behavior across the cluster.

In alignment with the project's operational constraints and deployment goals, the lightweight Kubernetes distribution *K3s* was chosen as the orchestration runtime. As discussed in Section 2.2.5.6, K3s offers a **fully compliant Kubernetes API surface** while significantly reducing the operational overhead associated with traditional Kubernetes setups. Its compact footprint,

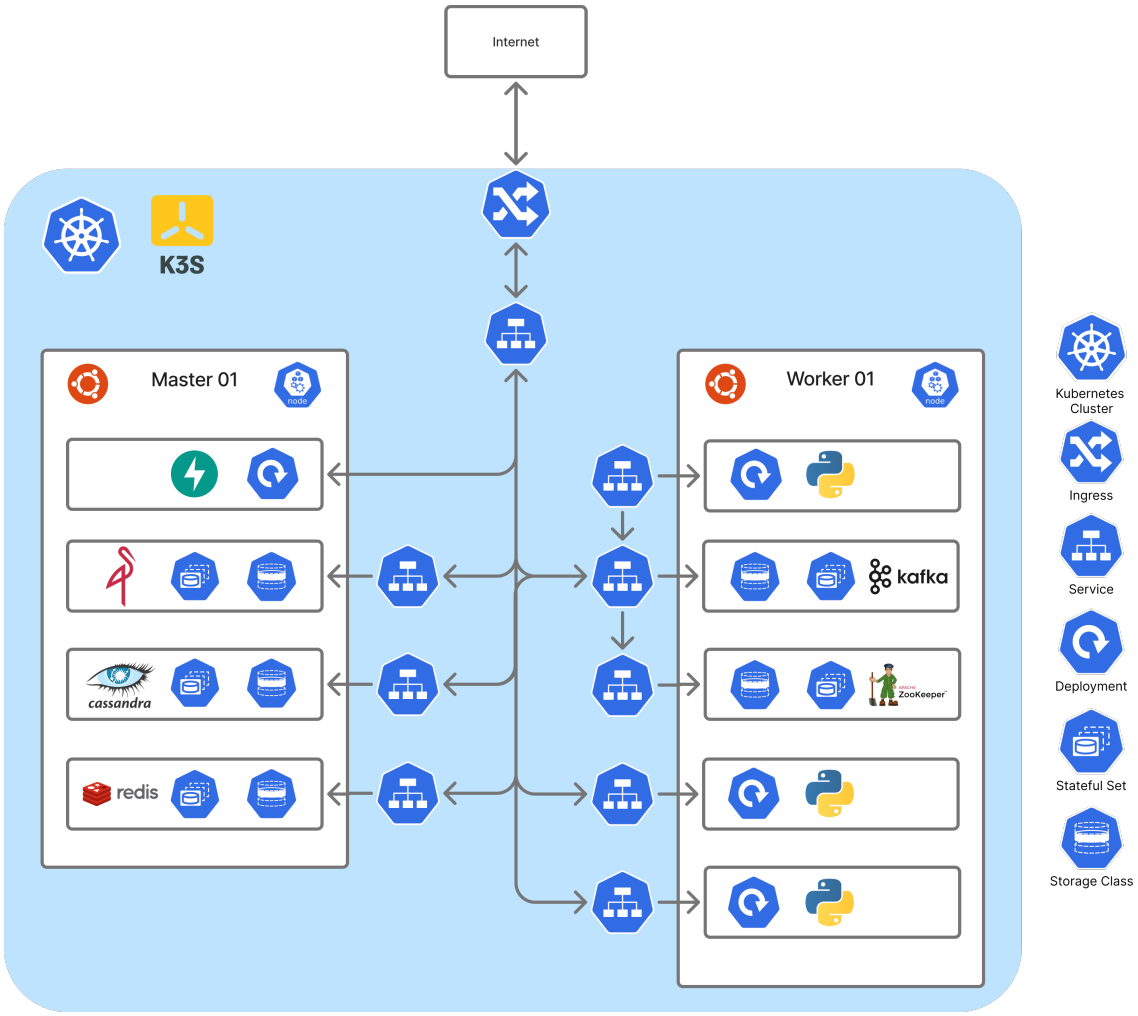


Figure 5.25: Kubernetes Implementation Diagram.

single-binary installation model, and streamlined control plane make it particularly suitable for development environments, as well as for deployments on limited-resource infrastructure such as virtual machines. These characteristics allowed the project to maintain a **production-grade orchestration layer** without incurring the complexity of managing a full-scale Kubernetes installation, while still supporting all the native Kubernetes features. K3s therefore provided the necessary infrastructure capabilities while preserving **system portability** and **administrative simplicity**.

Overall, Kubernetes was selected not merely as a container orchestrator, but as a foundational platform that fulfills both the **functional** and **non-functional requirements** of the system. Its ecosystem and tooling support, combined with a well-defined abstraction model, made it the most appropriate choice for deploying and managing the application in a production-ready, scalable manner.

5.5.1 Kubernetes-Based Orchestration Setup

The Kubernetes-based orchestration shown in Figure 5.25 begins with the deployment of a K3s cluster across two Ubuntu virtual machines. One node is configured as the control plane (Master 01), and the other as a worker node (Worker 01). K3s was installed using the official installation script on both nodes and the cluster was initialized from the master, and the worker node joined

using the node token generated during initialization. Since the setup is minimal, with only two nodes available, the default scheduling taint applied to the master node was removed allowing workloads to be scheduled on both machines, optimizing resource usage across the cluster.

In order to support persistent storage across multiple nodes, a **NFS Server** was configured on the master node as the Figure 4.2 shows and the exported directory `/mnt/nfs` is shared across the Kubernetes network and provides a common mount point for dynamically provisioned volumes. The NFS server was installed, the export rules were defined in `/etc/exports`, and appropriate permissions were applied to allow shared access. This setup ensures that data persisted by one pod remains accessible if that pod is rescheduled onto a different node.

To automate the creation of Persistent Volumes, the NFS Subdir External Provisioner was deployed in the cluster, so, it means that it runs as a pod within the `kube-system` namespace and manages volume provisioning using subdirectories within the shared NFS root path. The provisioner was configured with the environment variables `NFS_SERVER` and `NFS_PATH`, pointing respectively to the master node's IP and the exported directory. Storage classes were then defined for each stateful component (e.g., Cassandra, MinIO, Zookeeper), each class specifying the NFS provisioner, volume expansion permissions, and a custom reclaim policy.

A centralized **ConfigMap**, named `app-config`, was also created to manage shared environment variables across all components in the cluster. This configuration file contains settings such as hostnames, ports, bucket names, Redis and Kafka configurations, and MinIO credentials. Referencing this ConfigMap within pod specifications standardizes configuration management, facilitates updates, and avoids hardcoding values within individual manifests. It is imported into pods via the `envFrom` field, allowing each container to access its key-value pairs as standard environment variables.

K3s includes the Traefik ingress controller by default, which was used to route HTTPS traffic from external clients to internal services. Application traffic is forwarded to the backend service through defined ingress rules, which reference internal `ClusterIP` services by port and offers two paths, the api via `/api/v1/` and the websocket via `/ws/`. This configuration provides a single, secure entry point into the cluster, while also supporting path-based routing and TLS termination.

To enable secure HTTPS communication, a TLS certificate and private key were generated and then uploaded to the master node and registered as a Kubernetes TLS secret using the `sudo k3s kubectl create secret tls` command. The secret was assigned a specific name and stored within the `default` namespace.

Ingress rules were configured to reference this secret under the `tls` field of the Ingress resource specification. The rules included the fully qualified domain name used for TLS validation, the paths prefix to be matched as mentioned earlier, and the service name and port to which traffic should be directed. By associating the secret with the ingress resource, Traefik was able to terminate HTTPS requests at the edge of the cluster, ensuring that all external communications were encrypted.

5.5.2 Cassandra Deployment Setup

The deployment of *Apache Cassandra* followed the Kubernetes **StatefulSet** pattern, which provides stable network identities, persistent volume claims, and ordered startup behavior and it was initially configured with a single replica, but its definition is prepared for future horizontal scaling. Cluster awareness in Cassandra was established through critical environment variables that define the seed node, broadcast and listen addresses, data center and rack information, and other metadata. These settings allow each pod to autonomously identify and communicate with other nodes in the cluster once replicas are scaled beyond one. Cassandra listens on port 9042 for **CQL** and port 7000 for **intra-node gossip** communication. Both ports were exposed via Kubernetes services to fulfill their respective roles in data access and internal synchronization.

Persistent volumes for Cassandra were provisioned dynamically using the `nfs-cassandra` Storage Class, which integrates with the NFS Subdir External Provisioner to mount a unique subdirectory on the shared NFS server for each pod. This guarantees storage isolation while maintaining

centralized persistence. To ensure readiness and service stability, a readiness probe invoking `cqlsh` was included, delaying pod availability in the cluster until the database is fully responsive.

Cassandra was exposed internally via two Kubernetes services: a standard `ClusterIP` service used by application components to send queries, and a `Headless Service` for intra-node communication essential for maintaining gossip and cluster state. These service abstractions enable future horizontal scaling without requiring manual updates to pod IP addresses or DNS entries, preserving the declarative nature of the deployment as explained in Section 2.2.5.2.

5.5.3 Kafka & Zookeeper Deployment Setup

Following the deployment of Cassandra, the message brokering system was implemented using *Apache Kafka*, with *Zookeeper* serving as its coordination backend. Both components were deployed as separate `StatefulSet`s to support consistent pod identity, persistent storage, and ordered startup behavior which are key requirements for maintaining distributed consistency and reliable messaging semantics.

Zookeeper was configured as a single-replica `StatefulSet` using the confluentinc/cp-zookeeper image. The pod exposes the standard communication ports 2181 (client connections), 2888 (peer communication), and 3888 (leader election). Zookeeper's configuration was defined via environment variables, including the cluster unique address, allowing for future horizontal scaling by appending additional entries to the ensemble. A persistent volume is automatically provisioned through the `nfs-zookeeper StorageClass`, with each pod writing to its own subdirectory on the NFS server, as established earlier in the infrastructure. The service is exposed via both a `ClusterIP` service for internal access and a `Headless Service` to support intra-node DNS-based resolution, ensuring full support for Zookeeper's quorum-based coordination logic.

Kafka was similarly deployed as a `StatefulSet` using the `confluentinc/cp-kafka` image, configured to expose port 9092 for plaintext communication. To enable cluster-aware behavior, environment variables defined key settings such as the broker ID (computed dynamically based on the pod hostname), listener configuration, advertised addresses, and the connection string to the internal Zookeeper service. Kafka was exposed internally through two services: a `ClusterIP` service for standard access and a `Headless Service` that enables internal broker discovery, supporting seamless communication between potential multiple replicas in the future. The pod mounts its persistent data to the `/var/lib/kafka` path, backed by volumes dynamically provisioned from the `nfs-kafka StorageClass`.

This design ensures that both Kafka and Zookeeper are prepared for distributed, horizontally scalable deployment while maintaining the correct startup and communication semantics dictated by each system. The use of `Headless Services`, stateful DNS naming, and storage isolation across pods collectively support Kafka's broker-based topic partitioning and Zookeeper's fault-tolerant coordination model, essential for the system's asynchronous processing and event streaming pipeline.

5.5.4 MinIO & Redis Deployment Setup

In contrast to distributed components like Cassandra, Kafka, and Zookeeper, both MinIO and Redis were deployed using Kubernetes `StatefulSet` resources, but with single-instance replicas and without the use of `Headless Services`, therefore, they can only replicate data. These services do not participate in inter-node clustering but still require persistent volumes for maintaining critical data.

MinIO was configured to run as a standalone object storage server, and for that, it mounts a dedicated persistent volume claim using the `nfs-minio StorageClass`, backed by the same dynamically provisioned NFS server configured earlier. The pod runs on port 9000 and uses a predefined set of environment variables sourced from the shared `ConfigMap`, including administrative credentials, bucket names, and networking configuration. The configuration ensures a reproducible and centralized definition of operational parameters across environments, simplifying secret management and infrastructure-as-code strategies.

Redis was deployed in a similar fashion, using a single replica `StatefulSet` and a persistent volume claim based on the `nfs-redis StorageClass`. The pod exposes port 6379 for client interactions and persists its data to an NFS-mounted volume at `/data`. As Redis operates as an in-memory cache with durability enabled via volume mounts, this configuration provides resilience against unexpected pod restarts without introducing the complexity of clustered Redis replication.

5.5.5 Stateless Deployments: Backend & LLM Consumers

The backend service and all LLM consumers were deployed using the **Deployment** resource, which is designed for stateless workloads that can be safely terminated and restarted without local data persistence. Each deployment runs a single replica, but can be horizontally scaled by simply increasing the replica count. This stateless model is consistent with the architectural goals outlined in Section 4.1.1, particularly regarding elasticity and service modularity.

The backend service is responsible for exposing RESTful endpoints to external and internal clients. It is deployed as a standard container listening on port 8000 and configured via environment variables sourced from the shared `ConfigMap`. These include credentials, service URLs, ports, and access keys that provide the necessary configuration to connect with databases, caches, and object storage.

Similarly, three specialized LLM consumer services were deployed as separate pods: the `Check Resolved Incidents Consumer`, the `Clustering Consumer`, and the `Description Consumer`. Each listens to a dedicated Kafka topic and performs model inference or preprocessing tasks in response to events. These services operate on ports 8044, 8043, and 8042 respectively, and each receives a unique `TOPIC_NAME` and `LLM_GROUP_INSTANCE_ID` through individual environment variables. Common configuration values, such as Kafka broker address and MinIO credentials, are again provided by the centralized `ConfigMap`.

Because these services are designed to be stateless and event-driven, they do not require persistent storage. Kubernetes Deployments provide a robust restart mechanism and seamless pod rescheduling across nodes in the event of failure or resource changes, thereby aligning with the resilience and scalability requirements of this architecture.

5.5.6 Final Steps for Deployment

Once all manifests were written and the services properly defined, the final step to enable full deployment was to integrate with a container registry and configure authentication. A **Continuous Delivery** (CD) pipeline, builds container images from each service and publishes them to the **GitHub Container Registry** (GHCR) under the repository's organization. To pull these private images at runtime, each Kubernetes node must be granted permission to authenticate with GHCR.

This was achieved by creating a Kubernetes `docker-registry` secret named `ghcr-secret` using the following command shown in Figure 5.26:

```
k3s kubectl create secret docker-registry ghcr-secret \
  --docker-server=ghcr.io \
  --docker-username=<YOUR_GH_USER> \
  --docker-password=<YOUR_PAT> \
  --docker-email=<YOUR_EMAIL>
```

Figure 5.26: Kubernetes Docker Registry Secret Command.

This secret is referenced within each `Deployment` or `StatefulSet` manifest as illustrated in Figure 5.27:

To deploy a new version of any application, it is sufficient to update the container tag in the corresponding manifest and then trigger a restart of the relevant pod. Kubernetes will pull the updated image from GHCR and redeploy the service seamlessly, allowing for controlled, versioned rollouts across the cluster.

```
spec:
  imagePullSecrets:
    - name: ghcr-secret
  containers:
    - name: myapp
      image: ghcr.io/<YOUR_ORG>/myapp:<TAG>
      imagePullPolicy: IfNotPresent
```

Figure 5.27: Kubernetes Image Pull Secret Configuration.

5.6 Quality Assurance

5.6.1 Code Quality Standards

Code quality was a cornerstone of our project's development, maintained through a well-structured GitHub organization and a robust set of tools and practices. Our work was distributed across several dedicated repositories, ensuring modularity and clear ownership:

- **Documentation:** Project and technical documentation.
- **Mobile:** React Native implementation for the mobile application.
- **Sensors-Process-Unit:** Containerized processing unit for PIXKIT data, interacting with the main application.
- **Backend:** Core server logic and APIs, including asynchronous jobs via Kafka Message Queue.
- **Deploy:** Production environment configurations, including Kubernetes settings and deployment variables.
- **H3-Viewer:** Custom H3 index viewer for testing and real-time PIXKIT visualization.
- **Desktop-App:** Electron-wrapped React implementation for the desktop application.
- **RTMP-Server:** Initial Go-based RTMP server solution for PIXKIT (deprecated as processing moved in-vehicle).
- **Performance-Tests:** Performance testing suite utilizing k6.

For continuous code quality assurance, **SonarCloud** was integrated across all repositories. This enabled static code analysis, proactively identifying and addressing technical debt, vulnerabilities, and coding standard deviations.

Our **GitHub Workflow** leveraged *long-lived branches* (`main` and `dev`). Pull Requests (PRs) adhered to a strict template, and feature branches followed a consistent naming convention (e.g., `feature/<name>`, `bugfix/<name>`, `hotfix/<name>`, `task/<name>`). Code was merged first into `dev`, then into `main`. A key aspect of our Continuous Integration was the automatic generation of a new container image via `ghcr` upon every merge to `main`, facilitating streamlined manual deployments to our production environments.

5.6.2 Agile Methodology (Backlogs, Sprints, Workflows)

Our project embraced an Agile methodology, primarily centered around weekly sprints, to facilitate iterative development, continuous feedback, and adaptive planning. This approach fostered transparency and collaboration across the team.

The workflow management was meticulously organized using GitHub Boards, providing a clear visual representation of our progress. We maintained four distinct boards, each dedicated to a specific project area: **Mobile**, **Website**, **Backend**, and **Documentation**. Additionally, a

Overview board offered an overarching view of the project's status. Each board was structured with four standard columns: `Todo`, `In Progress`, `Review`, and `Done`. This structure allowed for a clear subdivision of tasks, ensuring that all team members could easily track the status of work items and contribute to their progression.

A core practice within our workflow was the issue-driven development. Every new task, bug, or feature was first logged as a GitHub Issue. Upon resolution, each Pull Request (PR) was meticulously linked to its corresponding Issue. This practice ensured full traceability from concept to implementation, providing a clear understanding of the rationale and context behind every code change, thereby enhancing project transparency and maintainability.

5.6.3 Team Meetings and Retrospectives

Effective communication and continuous improvement were ensured through a structured meeting cadence and diligent record-keeping.

Meeting Minutes: Detailed minutes from internal team discussions were consistently documented and managed within our project's documentation repository. This practice ensured a centralized and accessible record of decisions, action items, and key discussions.

Weekly Supervisory Meetings: Regular weekly meetings with our supervisors were a cornerstone of our project's progression. These sessions were highly structured, with the team consistently preparing and presenting slides to outline: Past challenges encountered and their resolutions. Future objectives and anticipated challenges for upcoming iterations. In-depth scientific discussions on critical project aspects, including geolocation intricacies, scalability strategies, integration with ATCLL, and other relevant technical considerations. The guidance and insights from our supervisors were instrumental, proving to be an essential component in the successful oversight and direction of the project.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The functional validation of the developed platform confirms that the core objectives have been successfully met, with all primary components operating as intended in realistic urban contexts. The system integrates diverse technologies – ranging from geospatial indexing and artificial intelligence to scalable data storage and edge computing – into a cohesive and reliable solution for smart city problem reporting and management.

A significant outcome of this validation process is the demonstrated ability of the platform to handle real-world scenarios such as overlapping jurisdictions and asynchronous AI-driven tasks. Furthermore, the design choices, such as the use of H3 for spatial indexing and Cassandra for high-throughput data storage, have proven effective in supporting both performance and scalability requirements.

From the end-user perspective, both citizen-facing and operator-facing workflows were thoroughly tested and validated. Citizens can report incidents with minimal effort through an intuitive mobile interface that automatically captures the location, suggests a description and category, and estimates severity based on photo input. City operators benefit from a desktop interface that provides a holistic and real-time overview of urban issues, featuring smart clustering of related occurrences, incident filtering, and heatmap visualizations for prioritization of problematic city regions.

Additionally, functional validation covered advanced use cases such as integration with autonomous edge devices. In particular, the proof of concept with the *Instituto de Telecomunicações'* PIXKIT autonomous vehicle successfully demonstrated automatic status updates of previously reported issues, as demonstrated **in this demo**. This capability is crucial for providing real-time updates and insights, connecting urban sensing infrastructure directly with municipal services and thereby significantly minimizing the need for human intervention.

A comprehensive video demonstration of the platform was structured to reflect the system's usability, responsiveness, and intelligent automation capabilities. This serves as visual confirmation of the platform's maturity and readiness for practical deployment. The demonstration video can be accessed **here**.

6.2 Future Work Directions

This section outlines potential improvements to enhance our system's performance, privacy, and robustness. One key area for future development is the use of large language models (LLMs), which are currently employed in multiple parts of the system: analyzing incident-related video clips, generating structured descriptions and categories from user reports, and clustering reports that likely refer to the same real-world incident. While using Google's Gemini via a cloud API provides powerful general-purpose capabilities, it presents limitations such as per-request costs,

reliance on third-party service uptime, overgeneralization for our specific domain, and the exposure of sensitive data. A promising improvement would be adopting a self-hosted LLM to retain data locally and reduce dependency on external providers, although this would require additional infrastructure. Fine-tuning a smaller, domain-specific model could further increase effectiveness and reduce computational overhead, allowing the model to better understand patterns and terminology specific to our use case.

To address privacy concerns, techniques like face and license plate anonymization should be applied to video frames before any processing. Finally, our use of the H3 geospatial index could be improved by exploiting its hierarchical structure to optimize distributed storage and speed up spatial queries across large incident datasets.

Bibliography

[Ama24] Amazon Web Services. *Amazon Simple Queue Service (SQS)*. 2024. URL: https://aws.amazon.com/sqs/.

[Ama25] Amazon Web Services. *Amazon S3 Storage Classes*. 2025. URL: https://aws.amazon.com/s3/storage-classes/.

[Apa24] Apache Software Foundation. *Apache Kafka: A Distributed Streaming Platform*. 2024. URL: https://kafka.apache.org/.

[ASS18] Saurabh Anand, Pallavi Singh, and B. Sagar. "Working with Cassandra Database". In: (2018), pp. 531–538. DOI: 10.1007/978-981-10-7563-6_55.

[Bro+20] Tom B. Brown et al. "Language Models are Few-Shot Learners". In: *arXiv preprint arXiv:2005.14165* (2020). URL: https://arxiv.org/abs/2005.14165.

[CH10] Jeff Carpenter and Eben Hewitt. *Cassandra: The Definitive Guide*. 2010.

[CKL15] Artem Chebotko, A. Kashlev, and Shiyong Lu. "A Big Data Modeling Methodology for Apache Cassandra". In: *2015 IEEE International Congress on Big Data* (2015), pp. 238–245. DOI: 10.1109/BigDataCongress.2015.41.

[Coi25] Coimbra City Council. *@Coimbra*. Accessed 2025. URL: https://apps.apple.com/ph/app/coimbra/id1519214983?uo=2.

[Dee23] DeepMind. "Gemini: A Family of Highly Capable Multimodal Models". In: *arXiv preprint arXiv:2312.11805* (2023). URL: https://arxiv.org/abs/2312.11805.

[Dev+18] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *arXiv preprint arXiv:1810.04805* (2018). URL: https://arxiv.org/abs/1810.04805.

[Eng] Uber Engineering. "Uber H3 Documentation Website". In: (). URL: https://h3geo.org/docs/highlights/indexing.

[etc24] etcd Documentation. *etcd: Distributed Reliable Key-Value Store*. 2024. URL: https://etcd.io/docs/.

[Eur16] European Union. *General Data Protection Regulation (GDPR)*. 2016. URL: https://gdpr.eu/.

[Fas25] FastAPI Documentation. *FastAPI: Modern, Fast (High-performance), Web Framework for Building APIs with Python 3.7+*. 2025. URL: https://fastapi.tiangolo.com/.

[Gam+94] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Goo24a] Google. *Gemini API Pricing*. 2024. URL: https://ai.google.dev/pricing.

[Goo24b] Google. *gRPC: A High-Performance, Open-Source Universal RPC Framework*. 2024. URL: https://grpc.io/.

[Gov25a] Government of Portugal. *A Minha Rua*. Accessed 2025. URL: https://www2.gov.pt/a-minha-rua.

[Gov25b] GovPilot. *GovPilot*. Accessed 2025. URL: https://www.govpilot.com/.

[HBB17] Kelsey Hightower, Brendan Burns, and J. Beda. "Kubernetes: Up and Running: Dive into the Future of Infrastructure". In: (2017).

[Hen+20] Dan Hendrycks et al. "Measuring Massive Multitask Language Understanding". In: *arXiv preprint arXiv:2009.03300* (2020). URL: https://arxiv.org/abs/2009.03300.

[Kub25a] Kubernetes CSI Documentation. *nfs-subdir-external-provisioner*. 2025. URL: https://github.com/kubernetes-sigs/nfs-subdir-external-provisioner.

[Kub25b] Kubernetes Documentation. *Container Runtimes*. 2025. URL: https://kubernetes.io/docs/setup/production-environment/container-runtimes/.

[Kub25c] Kubernetes Documentation. *Deployments*. 2025. URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/.

[Kub25d] Kubernetes Documentation. *Headless Services*. 2025. URL: https://kubernetes.io/docs/concepts/services-networking/service/#headless-services.

[Kub25e] Kubernetes Documentation. *Highly Available Kubernetes Clusters*. 2025. URL: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability/.

[Kub25f] Kubernetes Documentation. *Horizontal Pod Autoscaling*. 2025. URL: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[Kub25g] Kubernetes Documentation. *Ingress*. 2025. URL: https://kubernetes.io/docs/concepts/services-networking/ingress/.

[Kub25h] Kubernetes Documentation. *Ingress Controllers*. 2025. URL: https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/.

[Kub25i] Kubernetes Documentation. *kube-apiserver: The API Server*. 2025. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/.

[Kub25j] Kubernetes Documentation. *kube-proxy*. 2025. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/.

[Kub25k] Kubernetes Documentation. *kube-scheduler: The Kubernetes Scheduler*. 2025. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/.

[Kub25l] Kubernetes Documentation. *kubelet: The Primary NodeÄgent*. 2025. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/.

[Kub25m] Kubernetes Documentation. *Kubernetes Controller Manager*. 2025. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/.

[Kub25n] Kubernetes Documentation. *NFS Volume*. 2025. URL: https://kubernetes.io/docs/concepts/storage/volumes/#nfs.

[Kub25o] Kubernetes Documentation. *Persistent Volumes*. 2025. URL: https://kubernetes.io/docs/concepts/storage/persistent-volumes/.

[Kub25p] Kubernetes Documentation. *Pods*. 2025. URL: https://kubernetes.io/docs/concepts/workloads/pods/.

[Kub25q] Kubernetes Documentation. *Services*. 2025. URL: https://kubernetes.io/docs/concepts/services-networking/service/.

[Kub25r] Kubernetes Documentation. *StatefulSets*. 2025. URL: https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/.

[Kub25s] Kubernetes Documentation. *Storage Classes*. Accessed 2025-05-19. 2025. URL: https://kubernetes.io/docs/concepts/storage/storage-classes/.

[Kub25t] Kubernetes Documentation. *Vertical Pod Autoscaling*. 2025. URL: https://kubernetes.io/docs/tasks/run-application/vertical-pod-autoscaling/.

[Lis25] Lisbon City Council. *Na Minha Rua Lx*. Accessed 2025. URL: https://naminharualx.cm-lisboa.pt/.

[LM10] Avinash Lakshman and Prashant Malik. "Cassandra - A Decentralized Structured Storage System". In: *Operating Systems Review* 44 (Apr. 2010), pp. 35–40. DOI: 10.1145/1773912.1773922.

[Min25] Inc. MinIO. *Hyperscale Object Storage for AI*. 2025. URL: https://min.io.

[MQT24] MQTT. *MQTT: The Standard for IoT Messaging*. 2024. URL: https://mqtt.org/.

[New15] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2015.

[Ope24] OpenAI. *OpenAI API Documentation*. 2024. URL: https://openai.com/api/.

[Ope25] OpenAPI Initiative. *OpenAPI Specification*. 2025. URL: https://github.com/OAI/OpenAPI-Specification.

[Por25] Porto City Council. *ReportaPorto*. Accessed 2025. URL: https://reportaporto.cm-porto.pt/.

[Ran25] Rancher. *K3s Documentation*. 2025. URL: https://k3s.io/.

[Spr25] Spring Team. *Spring Framework*. 2025. URL: https://spring.io/.

[Tec25] TechEmpower. *TechEmpower Framework Benchmarks*. 2025. URL: https://www.techempower.com/benchmarks/.

[Uvi25] Uvicorn Documentation. *Uvicorn: The lightning-fast ASGI server*. 2025. URL: https://www.uvicorn.org/.

[VMw24] VMware. *RabbitMQ: Messaging that just works*. 2024. URL: https://www.rabbitmq.com/.