2025

Internship Project

MBA BA
II Year I Trimester

# Gemma: AI-Driven EMS Analytics

## [HEALTHCARE DATA ANALYSIS]

This project presents EMS Analytics with AI, an intelligent system for emergency medical services. It integrates a large language model (Gemma 7B instruct) with hybrid machine learning to extract insights, assess risk, and provide real-time decision support. The interactive dashboard visualizes operational KPIs, geospatial patterns, and supports natural language queries, enabling faster and informed EMS responses.

# Contents

List of figures ..... 3
Project cover sheet ..... 5
Executive Summary ..... 6
Project Milestones ..... 6
Introduction ..... 7
Problem statement ..... 8
Objectives ..... 8
Scope ..... 9
Current Capabilities ..... 9
Future Enhancements / Potential Work ..... 9
Specifications ..... 10
1. Functional Specifications ..... 10
1.1 Data Ingestion and Preprocessing ..... 10
1.2 Exploratory Data Analysis (EDA) ..... 10
1.3 Geospatial Analysis ..... 10
1.4 Operational Efficiency Analysis ..... 11
1.5 Risk Classification ..... 11
1.6 AI-Assisted Protocol Guidance ..... 11
1.7 Dashboard and User Interaction ..... 11
2. Technical Specifications ..... 11
2.1 Programming Languages ..... 11
2.2 Backend Framework ..... 12
2.3 Data Processing and Geospatial Libraries ..... 12
2.4 Machine Learning and Risk Modeling ..... 12
2.5 AI and Retrieval-Augmented Generation (RAG) Components ..... 12
2.6 Visualization Tools ..... 12
2.7 Data Storage Formats ..... 13
2.8 File Handling and Document Processing Utilities ..... 13
2.9 Development Environment ..... 13
2.10 API Schema Definition and Type Management ..... 13
Project Architecture ..... 14

1. Data Cleaning and Preprocessing ..... 15
2. Exploratory Data Analysis (EDA) Engine ..... 15
3. Parallel Analytical Modules ..... 15
3.1 Geospatial Hotspot Analysis ..... 15
3.2 Operational Efficiency Analysis ..... 15
3.3 Risk Classification ..... 16
4. Analytical Output Integration and AI-Assisted Querying ..... 16
5. Architectural Design Principles ..... 16
System Architecture ..... 17
Backend Software Components ..... 17
1. Data Processing Module ..... 17
2. Analytical Services ..... 17
3. AI and Retrieval Service ..... 18
4. API Layer ..... 18
Frontend Software Components ..... 18
1. Visualization Components ..... 18
2. Interaction and Query Interface ..... 18
Data Flowchart ..... 20
Data dictionary ..... 21
Extract, Transform and Load pipeline (ETL) ..... 23
1. Extract: ..... 23
2. Transform: ..... 23
Data Cleaning ..... 23
3. Load: ..... 24
Data Analysis ..... 25
Exploratory Data Analysis (EDA) ..... 25
Geospatial Hotspot Analysis ..... 32
Step-by-Step Methodology ..... 32
Operational Efficiency Analysis: ..... 37
Step-by-Step Methodology ..... 37
Risk Classification Analysis: ..... 51
Step-by-Step Methodology ..... 51

Gemma Chatbot ..... 58
Step-by-Step Methodology ..... 58
Dashboards ..... 68
Key Findings ..... 73
Recommendations ..... 74
Conclusion ..... 75

## List of figures

Figure 1.Project architecture ..... 14
Figure 2 System architecture ..... 17
Figure 3 Distribution of patient age ..... 25
Figure 4 Top 10 injury places ..... 26
Figure 5 Top 10 primary impressions ..... 27
Figure 6 Top 10 hospitals by incident transports ..... 28
Figure 7 call volume by hour distribution ..... 29
Figure 8 call volume by days ..... 30
Figure 9 incidents per month ..... 31
Figure 10 Computing on scene times code snippet ..... 33
Figure 11 H3 indexing code snippet ..... 33
Figure 12 Aggregating H3 cells code snippet ..... 34
Figure 13 Adding polygon shapes code snippet ..... 35
Figure 14 HTML map made using chloropleth map ..... 36
Figure 15 Time columns computation code snippet ..... 38
Figure 16 KPI computation code ..... 39
Figure 17 Trends over time code ..... 41
Figure 18 Distributions of time columns code ..... 42
Figure 19 Delay bucket analysis code ..... 43
Figure 20 KPI cards ..... 45
Figure 21 Distribution of response time ..... 46
Figure 22 Distribution of on-scene time ..... 47
Figure 23 Call cycle time distribution ..... 48
Figure 24 Operational Trends over year ..... 49
Figure 25 Response time by hour of the day ..... 49
Figure 26 Delay buckets based on SLA ..... 50
Figure 27 Medical text construction code snippet ..... 51
Figure 28 K-means clustering code snippet ..... 52
Figure 29 Sampling from clusters code snippet ..... 52
Figure 30 Label extraction code snippet ..... 53

Figure 31 Cluster diagnostics code snippet ..... 53
Figure 32 Test- Train split code snippet..... 54
Figure 33 Training the model code snippet ..... 55
Figure 34 Distribution of classes ..... 56
Figure 35 Confusion matrix for light gradient boost classifer..... 57
Figure 36 Classification report ..... 58
Figure 37 Extraction code snippet..... 59
Figure 38 Chunking code snippet..... 60
Figure 39 embedding code snippet..... 61
Figure 40 Chat end point using FastAPI code snippet ..... 62
Figure 41 Intent classification code snippet ..... 63
Figure 42 selecting RAG when necessary ..... 63
Figure 43 prompt design code snippet..... 65
Figure 44 Gemma chatbot ..... 66
Figure 45 Working of chatbot ..... 67
Figure 46 Dashboard page 1 ..... 70
Figure 47 Dashboard page 2 ..... 70
Figure 48 Dashboard page 3 ..... 71
Figure 49 Dashboard page 4 ..... 71
Figure 50 Dashboard page 5 ..... 72
Figure 51 dashboard page 6..... 72

# Project cover sheet

| | | | |
|---|---|---|---|
| **Project Title** | Gemma: AI-Driven EMS Analytics | **Course** | MBA BA I Year – II Trimester |
| **Start Date** | 24-11-2025 | **Guide/Client** | Dr. Mansoor |
| **End Date** | 13-12-2025 | **Student** | Sirigiri Varshitha |

Signature:

Guide/Client Name: Dr. Mansoor

Date:

## Executive Summary

This project, EMS Analytics with AI, develops an intelligent system to support emergency medical services. It combines a large language model (Gemma 7B instruct) with traditional machine learning to extract structured information from EMS data, assess incident severity, and generate risk scores. The system also integrates geospatial analysis and operational KPIs to provide actionable insights for EMS teams, enabling faster and more informed decision-making.

The solution features an interactive React and FastAPI dashboard with a Q/A assistant powered by Gemma. Users can explore incident trends, view risk assessments, and query the system in natural language. The design ensures that AI recommendations are transparent, sourced, and actionable, making it a reliable decision-support tool for real-world EMS operations. This project demonstrates the effective application of AI in critical, time-sensitive environments.

.

## Project Milestones

| SNO | Milestone | Detailed Description | Duration |
|---|---|---|---|
| 1 | Domain studies | • Studying domain<br>• Listing out problems in the sector | 1 day |
| 2 | Data collection | • Studying requirements<br>• Data collection<br>• Project architecture design | 3 days |
| 3 | Data cleaning | • Cleaning | 3 days |
| 4 | Exploratory data analysis (EDA) | • Feature engineering<br>• Univariate analysis<br>• Bivariate analysis | 2 days |
| 5 | Data analysis | • Geospatial hotspot analysis<br>• Operational efficiency analysis<br>• Risk classification | 5 days |
| 6 | Dashboard and chatbot | • React dashboard<br>• FastAPI backend server<br>• Gemma powered chatbots | 1 week |
| 8 | Interpretations and report writing | • Final report<br>• Project presentation | 2 days |

# Introduction

Emergency Medical Services (EMS) are a cornerstone of modern healthcare, providing urgent care and rapid response to life-threatening situations. From accidents and cardiac events to trauma and medical emergencies, EMS teams are often the first point of professional medical intervention. The decisions they make in the field—such as prioritizing incidents, dispatching resources, and coordinating patient transport—can significantly influence patient outcomes and overall system efficiency.

Traditionally, EMS operations have relied on experience, manual reporting, and basic data tracking to guide decision-making. While these methods provide some insight, the increasing volume and complexity of emergency incidents make it challenging to identify trends, allocate resources efficiently, or predict high-risk scenarios. Delays, incomplete data, and the lack of integrated analytics tools can hinder situational awareness, limit performance evaluation, and reduce the overall effectiveness of EMS systems.

In recent years, there has been a growing recognition of the value of data-driven approaches in EMS. Collecting and analyzing historical incident data, response times, and resource utilization patterns can provide actionable insights to improve operational planning and patient care. Integrating advanced analytics and intelligent systems has the potential to enhance decision-making, identify critical areas requiring attention, and optimize the allocation of personnel and equipment.

This project is motivated by the need to leverage these opportunities, aiming to build a system that supports EMS personnel with better insights, structured information, and context-aware guidance. By understanding patterns, trends, and risk factors in EMS data, the system seeks to improve situational awareness, support informed decision-making, and ultimately enhance the effectiveness and responsiveness of emergency medical services.

## Problem statement

*"Handling and analyzing EMS data is complex, and there is a need for a system that provides research insights, statistical analysis, and quick AI-assisted protocol reference for responders, without replacing professional judgment.*

EMS generates large volumes of incident and response data every day. Manually exploring this information to identify trends, assess risk factors, or review relevant protocols is time-consuming and error-prone. While AI can assist by extracting structured insights and suggesting potential protocols, it is not fully reliable and cannot replace professional judgment. The system is designed to help researchers, analysts, and EMS personnel access quick references, review patterns, and explore data efficiently. It can suggest possible protocols during transit or planning, but all critical decisions remain the responsibility of trained responders.

## Objectives

- To assign risk levels to EMS incidents using structured features such as incident type, patient demographics, disposition, and timestamps.
- To compute response times, identify unusually long responses, and generate visualizations like histograms, boxplots, and line plots to evaluate operational efficiency.
- To identify regions with higher incident activity and summarize geographic clusters for reference and research.
- To provide AI-powered suggestions on possible EMS protocols through a Q/A chatbot as a quick reference, without replacing professional judgment.

# Scope

## Current Capabilities

- Analyze EMS incidents with descriptive statistics and visualizations:
    - Response times, on-scene durations, and operational KPIs
- Identify geospatial hotspots using H3 hex indexing to highlight high-incident regions
- Access AI-assisted EMS protocol guidance via a chatbot (Gemma 7B)
    - Provides structured feature extraction, risk insights, and recommendations
    - Serves as a research and reference tool (not for life-critical decisions)
- Export cleaned and aggregated datasets for further analysis in CSV or GeoJSON formats

## Future Enhancements / Potential Work

- Integrate hospital coordinates to compute distance-to-hospital metrics and implement a nearest-hospital guidance system for EMS units
- Expand dashboard capabilities:
    - Real-time incident updates
    - Predictive analytics for incident trends and resource allocation
    - Interactive maps with clickable incidents and dynamic filters
- Incorporate additional AI-driven insights:
    - Enhanced risk scoring with more historical data
    - Full RAG retrieval in chatbot for SOP and incident context
    - Automated narrative reports for anomaly detection
- Enhance data quality and validation:
    - Fill missing or incomplete fields (primary injury, injury details)
    - Standardize protocol and operational entries for improved reliability
- Improve operational decision support:
    - Advanced geospatial analysis (routing, resource optimization)
    - Human-in-loop review pipelines for retraining models
    - Automated batch extraction of incidents for large-scale analysis

# Specifications

## 1. Functional Specifications

The EMS Insight AI system provides the following core functionalities:

### 1.1 Data Ingestion and Preprocessing

- Ingest EMS incident datasets in CSV format.
- Perform data cleaning to address missing values, inconsistent formatting, and incomplete records.
- Standardize operational timestamp fields into a uniform datetime representation.
- Normalize and standardize categorical attributes such as city names, protocol identifiers, and injury descriptors.
- Compute derived operational metrics, including:
    - Response time
    - Turnout time
    - On-scene time
    - Call cycle time
- Export cleaned and aggregated datasets in CSV and GeoJSON formats for downstream analysis.

### 1.2 Exploratory Data Analysis (EDA)

- Generate descriptive statistics summarizing incident volume, response times, and operational performance.
- Support temporal analysis across years and months.
- Provide analytical summaries that serve as the foundation for advanced analytical modules.

### 1.3 Geospatial Analysis

- Process incident latitude and longitude data.
- Perform spatial aggregation using hexagonal indexing techniques.
- Identify and visualize geographic hotspots of EMS incident activity.

- Produce geospatial outputs suitable for map-based visualization.

### 1.4 Operational Efficiency Analysis

- Analyze EMS operational performance using derived time-based metrics.
- Evaluate response and on-scene duration trends across incidents and locations.
- Support identification of operational inefficiencies for research and analytical purposes.

### 1.5 Risk Classification

- Implement a risk classification model using structured EMS features.
- Assign relative risk categories to incidents for comparative analysis.
- Support model evaluation and analytical interpretation of risk distributions.

### 1.6 AI-Assisted Protocol Guidance

- Enable natural language querying of EMS protocols and SOP documents.
- Retrieve relevant protocol content using embedding-based similarity search.
- Generate context-aware AI responses to support protocol reference and analytical understanding.
- Present AI outputs in a structured and readable format.

### 1.7 Dashboard and User Interaction

- Provide an interactive dashboard for visualizing analytical outputs.
- Integrate results from EDA, geospatial analysis, operational efficiency analysis, and risk classification.
- Embed an AI-powered chat assistant for protocol and analysis-related queries.

## 2. Technical Specifications

This section details the software components, libraries, and tools used to implement the system.

### 2.1 Programming Languages

- **Python**
  Used for backend data processing, exploratory analysis, geospatial modeling, operational analysis, risk classification, and AI integration.
- **JavaScript (React)**
  Used for developing the interactive frontend dashboard and rendering visual analytics.

### 2.2 Backend Framework

- **FastAPI**
  Used to develop backend services and RESTful APIs for serving analytical results and AI-assisted responses to the frontend.

### 2.3 Data Processing and Geospatial Libraries

- **Pandas**
  Used for data ingestion, cleaning, normalization, standardization, and feature engineering.

- **GeoPandas**
  Used for geospatial data handling and spatial operations.

- **H3**
  Used for hexagonal spatial indexing and hotspot aggregation.

- **Shapely (Polygon)**
  Used to construct and manipulate polygon geometries representing spatial regions, including aggregated H3 hexagonal cells and administrative or analytical boundaries. These polygon structures enable spatial operations such as containment checks, spatial aggregation, and visualization preparation for hotspot analysis.

### 2.4 Machine Learning and Risk Modeling

- **Scikit-learn**
  Used for preprocessing, model development, training, and evaluation of the risk classification component, as well as clustering-based exploratory analysis.

### 2.5 AI and Retrieval-Augmented Generation (RAG) Components

- **Gemma 7B Instruct**
  Used for AI-assisted protocol guidance and structured feature extraction.

- **SentenceTransformers**
  Used to generate vector embeddings for EMS protocol documents.

- **FAISS**
  Used as the vector database for similarity-based retrieval within the RAG pipeline.

### 2.6 Visualization Tools

- **Recharts**
  Used within the React dashboard for KPI charts and analytical visualizations.

- **Folium**
  Used to generate geospatial map visualizations.

### 2.7 Data Storage Formats

- **CSV**
  Used for storing raw and cleaned EMS datasets.

- **JSON**
  Used for storing analytical summaries, extracted protocol content, and AI-generated outputs.

### 2.8 File Handling and Document Processing Utilities

- **os and pathlib**
  Used for operating system interactions, directory traversal, and platform-independent file path management across backend services and APIs.

- **json**
  Used for serialization and structured data exchange between system components.

- **pdfplumber**
  Used to extract textual content from EMS protocol and SOP PDF documents and convert them into structured JSON format for AI processing.

- **tqdm**
  Used for monitoring progress during batch preprocessing and embedding generation tasks.

- **requests**
  Used for HTTP communication with AI services.

- **re (Regular Expressions)**
  Used for text preprocessing, pattern matching, and categorical normalization.

### 2.9 Development Environment

- **VS Code**
  Used as the primary development environment.

- **Git**
  Used for version control and source code management.

- **Browser-Based Deployment**
  The frontend dashboard is deployed as a web-based application accessible through standard browsers.

### 2.10 API Schema Definition and Type Management

To ensure structured data exchange, validation, and consistency across backend services and frontend consumption, the system employs schema definition and type management utilities:

- **Pydantic (BaseModel)**
  Used to define data schemas for API request and response bodies. Pydantic models enforce data validation, type checking, and structured serialization, ensuring reliable communication between the FastAPI backend and the frontend dashboard.

- **Typing Module (List, Dict, Any, Optional)**
  Used to provide explicit type annotations across backend functions, API endpoints, and data models. These annotations improve code readability, maintainability, and reduce runtime errors by clarifying expected data structures.

## Project Architecture

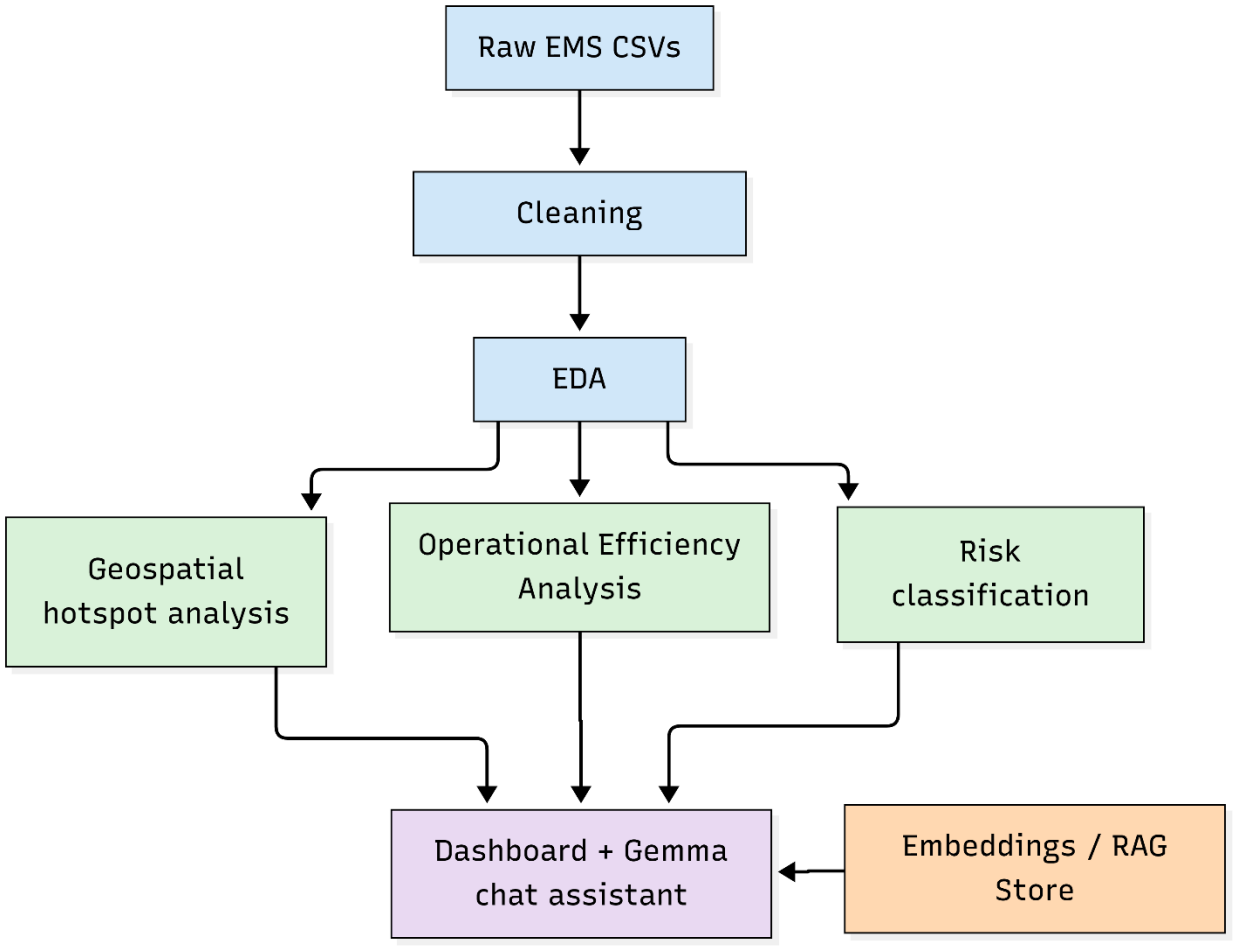



*Figure 1.Project architecture*

The EMS Insight AI project is designed as a modular analytical pipeline that transforms raw EMS incident data into structured insights through sequential and parallel analysis stages. The architecture emphasizes clarity of data flow, analytical transparency, and reproducibility, enabling systematic exploration of EMS operational data.

## 1. Data Cleaning and Preprocessing

The analytical pipeline begins with raw EMS incident data provided in CSV format. These datasets undergo a comprehensive preprocessing stage to ensure data quality and consistency. During this stage, temporal fields are normalized to a uniform format, categorical variables are standardized, and incomplete or inconsistent entries are addressed. Derived operational metrics, including

response time, turnout time, on-scene duration, and total call cycle time, are computed to support downstream analytical tasks. This stage produces a clean, structured dataset that serves as the foundation for all subsequent analysis.

## 2. Exploratory Data Analysis (EDA) Engine

Following preprocessing, the cleaned dataset is processed by the Exploratory Data Analysis (EDA) engine. This component performs descriptive and exploratory analysis to uncover patterns and trends within the data. Key outputs include statistical summaries of incident volumes, temporal distributions, and operational performance metrics. The EDA engine establishes a baseline understanding of the dataset and informs the interpretation of results generated by later analytical modules.

## 3. Parallel Analytical Modules

After EDA, the analytical workflow branches into three independent but complementary modules. Each module operates on the same preprocessed dataset, ensuring consistency across analytical outputs.

### 3.1 Geospatial Hotspot Analysis

The geospatial analysis module applies spatial indexing and aggregation techniques to identify geographic regions with elevated EMS incident activity. This component enables the detection of spatial hotspots and supports spatial pattern analysis, providing location-based insights into incident distribution.

### 3.2 Operational Efficiency Analysis

The operational efficiency module evaluates EMS performance using derived time-based metrics. By analyzing response times, on-scene durations, and overall call cycle metrics, this module identifies operational trends and potential inefficiencies across different temporal and spatial dimensions.

### 3.3 Risk Classification

The risk classification module assigns relative risk categories to EMS incidents based on structured features derived during preprocessing and analysis. This module supports comparative risk assessment and prioritization for analytical and research purposes.

## 4. Analytical Output Integration and AI-Assisted Querying

Outputs from the geospatial, operational, and risk analysis modules are aggregated and presented through a unified analytical interface. In addition, the project integrates an AI-assisted querying mechanism powered by a language model and a retrieval-augmented generation (RAG) store. This component enables natural language querying of EMS protocols and analytical summaries, with responses grounded in retrieved reference documents and analytical context.

## 5. Architectural Design Principles

The project architecture follows a modular and extensible design, allowing individual analytical components to evolve independently without disrupting the overall pipeline. The architecture is intended for research, exploratory analysis, and decision support, with a focus on interpretability and traceability rather than real-time or life-critical operational deployment.

## System Architecture

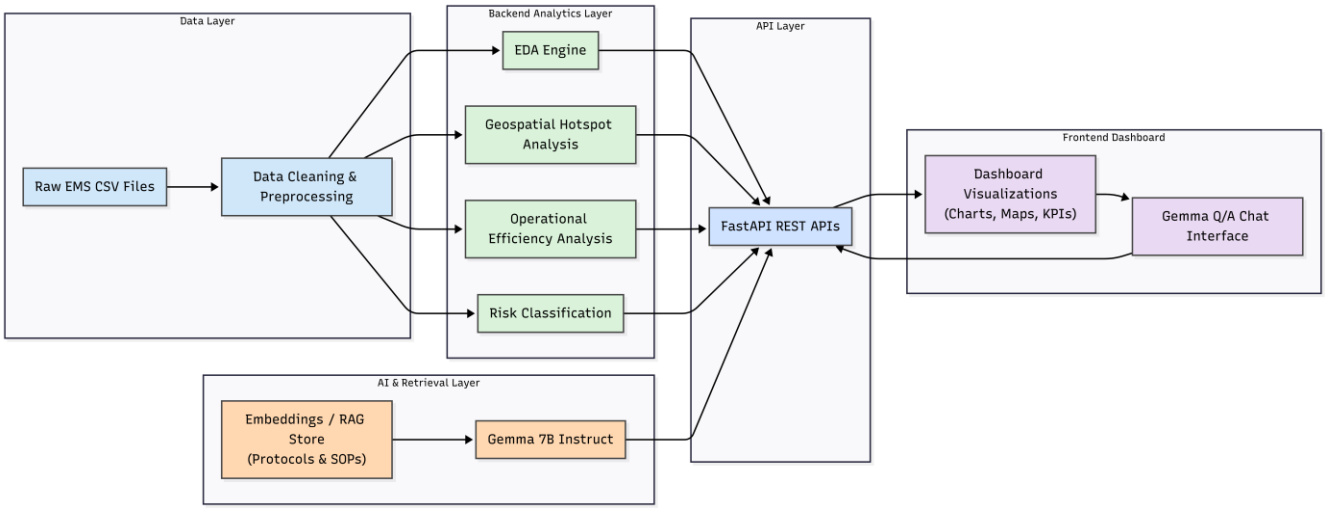



*Figure 2 System architecture*

The EMS Insight AI system is implemented using a client–server software architecture that separates data processing, analytics, and AI orchestration from user interaction and visualization. The system is composed of a backend service layer responsible for computation and data serving, and a frontend dashboard responsible for visualization and user interaction. Communication between components is achieved through well-defined RESTful APIs.

## Backend Software Components

The backend forms the core processing layer of the system and is implemented using FastAPI. It encapsulates the following software components:

### 1. Data Processing Module

This module handles ingestion of raw EMS CSV files and executes preprocessing routines, including timestamp normalization, categorical standardization, duplicate handling, and computation of derived operational metrics. The processed datasets are persisted as structured CSV and JSON files for reuse across analytical services.

### 2. Analytical Services

The backend exposes individual analytical services corresponding to each analysis module:

- **EDA Service** generates descriptive statistics and summary metrics.
- **Geospatial Analysis Service** performs spatial indexing, aggregation, and hotspot detection.
- **Operational Efficiency Service** evaluates time-based performance indicators.
- **Risk Classification Service** computes incident-level risk categories.

Each service operates independently but consumes the same preprocessed datasets, ensuring consistency across outputs.

### 3. AI and Retrieval Service

This component integrates the Gemma 7B Instruct language model with a Retrieval-Augmented Generation (RAG) mechanism. The RAG store maintains vectorized representations of EMS protocols and reference documents. Upon receiving a query, relevant documents are retrieved and injected into the AI prompt to produce context-aware and grounded responses.

### 4. API Layer

The API layer exposes backend functionality to external clients through REST endpoints. These endpoints allow retrieval of analytical results, geospatial outputs, risk classifications, and AI-assisted responses. All responses are returned in standardized JSON format.

## Frontend Software Components

The frontend is a browser-based application implemented using React, serving as the presentation and interaction layer of the system.

### 1. Visualization Components

Frontend components render analytical outputs received from the backend, including charts, tables, and geospatial visualizations. These components are responsible solely for presentation and do not perform analytical computation.

### 2. Interaction and Query Interface

The dashboard provides interactive controls such as filters and a chat interface. User actions are translated into API requests sent to the backend, which returns processed analytical or AI-generated responses for display.

**Component Interaction and Data Exchange:**

1. The frontend initiates requests based on user interaction.
2. Requests are routed to the appropriate backend API endpoint.
3. Backend services process the request by accessing preprocessed datasets or invoking analytical or AI components.
4. Results are returned to the frontend as structured JSON responses.
5. The frontend renders the results without modifying underlying data.

**Architectural Characteristics:**

- **Separation of Concerns:** Analytical computation and AI processing are isolated from visualization logic.

- **Modularity:** Backend services are independently maintainable and extensible.
- **Scalability:** Analytical and AI services can be scaled independently if required.
- **Traceability:** Data transformations and AI outputs are deterministic and reproducible.

## Data Flowchart

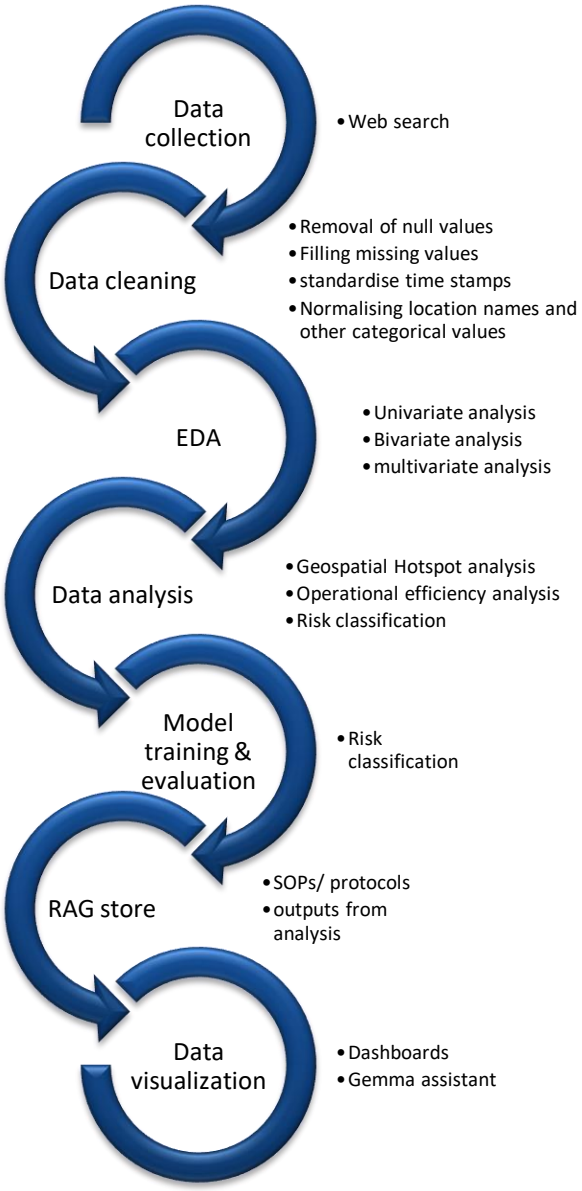



*Figure 3. Data flow diagram*

# Data dictionary

| Field_Name | Description | Data_Type |
|---|---|---|
| **incident_number** | Original EMS incident identifier assigned at call creation | String |
| **unique_id** | System-generated unique identifier used for analysis and tracking | String |
| **time_call_was_received** | Timestamp when the emergency call was received | DateTime |
| **time_vehicle_was_dispatched** | Timestamp when the EMS vehicle was dispatched | DateTime |
| **time_vehicle_was_en_route_to_scene** | Timestamp when the vehicle began travel to the scene | DateTime |
| **time_arrived_on_scene** | Timestamp when EMS arrived at the incident location | DateTime |
| **time_arrived_at_patient** | Timestamp when EMS reached the patient | DateTime |
| **time_departed_from_the_scene** | Timestamp when EMS departed the incident scene | DateTime |
| **time_arrived_to_next_destination** | Timestamp when EMS arrived at the destination (e.g., hospital) | DateTime |
| **year_call_received** | Calendar year in which the call was received | Integer |
| **month_call_received** | Calendar month in which the call was received | Integer |
| **incident_address** | Street-level address of the incident | String |
| **incident_city** | Standardized city name of the incident location | String |
| **incident_zip_postal** | Postal ZIP code of the incident location | String |
| **incident_county** | County where the incident occurred | String |
| **incident_latitude** | Latitude coordinate of the incident location | Float |

| | | |
|---|---|---|
| **incident_longitude** | Longitude coordinate of the incident location | Float |
| **incident_geocode_location** | Geocoded representation of the incident location | String |
| **primary_impression** | Primary clinical impression recorded by EMS personnel | String |
| **primary_injury** | Primary injury classification | String |
| **injury_detail** | Detailed description of the injury | String |
| **injury_place** | Location or environment where the injury occurred | String |
| **injury_date** | Date on which the injury occurred | Date |
| **protocol_used_by_ems_personnel** | EMS protocol followed during incident response | String |
| **patient_age** | Age of the patient at the time of the incident | Integer |
| **patient_gender** | Gender of the patient | String |
| **patient_home_county** | County of the patient's residence | String |
| **disposition** | Final outcome of the incident | String |
| **where_patient_was_transported** | Destination facility where the patient was transported | String |

# Extract, Transform and Load pipeline (ETL)

## 1. Extract:

The data was collected from data.marincounty.gov. It consists of 1,49,998 rows and 29 columns.

## 2. Transform:

The data was preprocessed to make it suitable for analysis. It includes data cleaning, feature engineering, and encoding.

### Data Cleaning

The EMS incident dataset underwent a systematic cleaning and preprocessing pipeline to ensure consistency, reliability, and usability for downstream analysis. The steps included:

1. **Timestamp Normalization**
    - All date and time fields (e.g., Time Call Was Received, Time Vehicle Was Dispatched, Time Arrived on Scene) were converted into a uniform ISO 8601 DateTime format (YYYY-MM-DD HH:MM:SS)..
    - **Tools used:** pandas for parsing and arithmetic on timestamps, numpy for computations.
2. **Categorical Value Standardization**
    - Inconsistent categorical entries were mapped to canonical values.
    - Examples:
        - City names: SANRAFAEL, SanRafael → San Rafael
        - Protocol names, primary injuries, and impression labels were standardized for consistent downstream analysis.
    - **Tools used:** pandas string methods, re (regular expressions) for pattern matching and corrections.

3. **Duplicate Removal**
    - Duplicate incidents (based on Incident Number or Unique ID) were identified and removed to ensure each record represented a unique EMS call.
    - **Tools used:** pandas.drop_duplicates()
4. **Handling Missing or Null Values**
    - Critical fields like timestamps, city, and incident type were validated for completeness.
    - Optional fields (e.g., Injury Detail, Injury Place) were retained as null if unavailable, to preserve all usable data.
    - **Tools used:** pandas.isnull(), fillna(), and conditional filtering.
5. **Geocoding & Location Verification**
    - Verified Incident Latitude and Incident Longitude for correctness.
    - Prepared Incident Geocode Location for spatial indexing using H3 hexagonal grids for geospatial analysis.
    - **Tools used:** geopandas for geospatial operations, h3 for hex indexing.

After cleaning total rows include 1,08,539 rows.

## 3. Load:

The dataset is loaded using pandas from python.

# Data Analysis

## Exploratory Data Analysis (EDA)

After cleaning exploratory data analysis (EDA) was performed on the dataset.

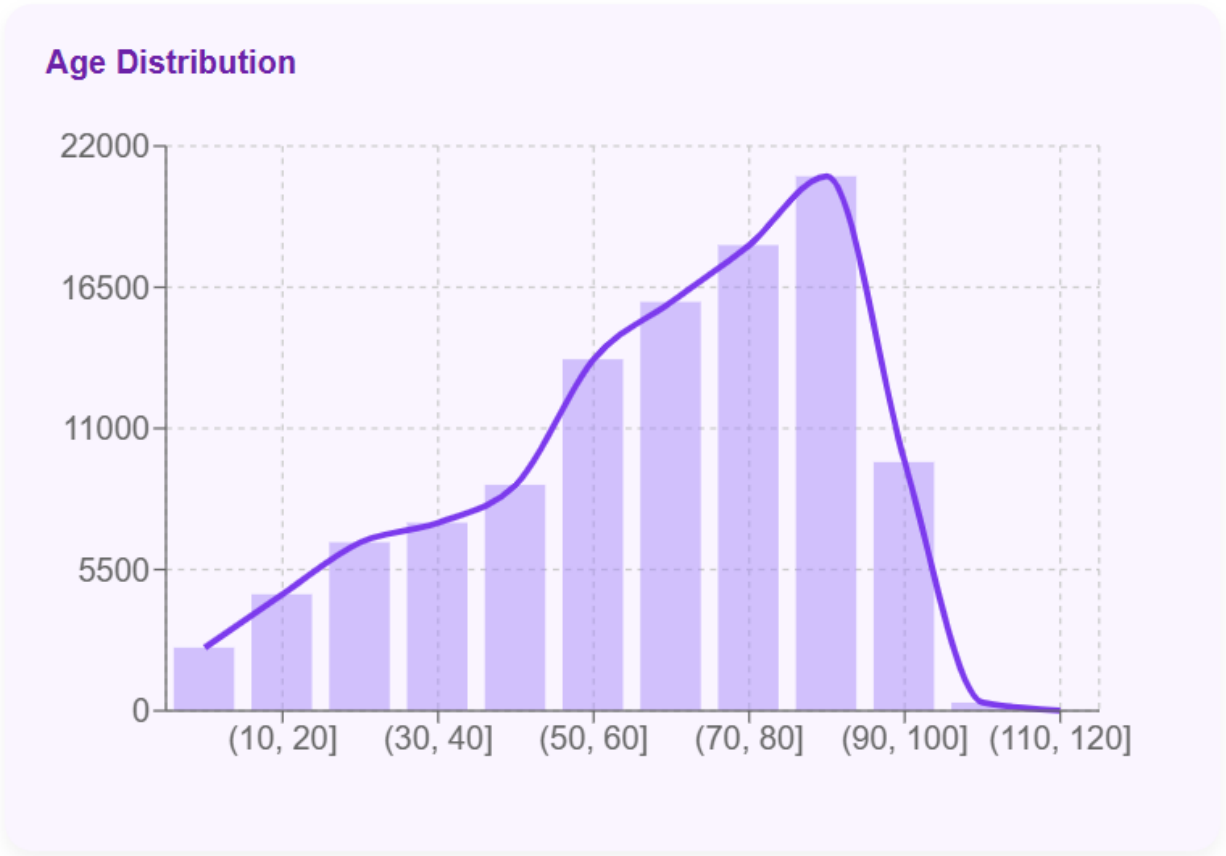



*Figure 3 Distribution of patient age*

- **Histogram & KDE (Kernel Density Estimation) Plot of patient age:**
  - The X- Axis represents the Patient age.
  - The Y-Axis represents the count of the patients.
  - Most of the population is middle-aged to elderly, with counts increasing from ages 10–20 up to around 70–80.
  - The highest concentration of people appears in the 70–80 and 80–90 age groups, where both the bar height and the line peak.
  - Younger age groups (10–40) have noticeably fewer people compared with older groups, indicating an aging population structure.

- After about 90 years, the counts drop sharply, showing that very few individuals are older than 90–100.

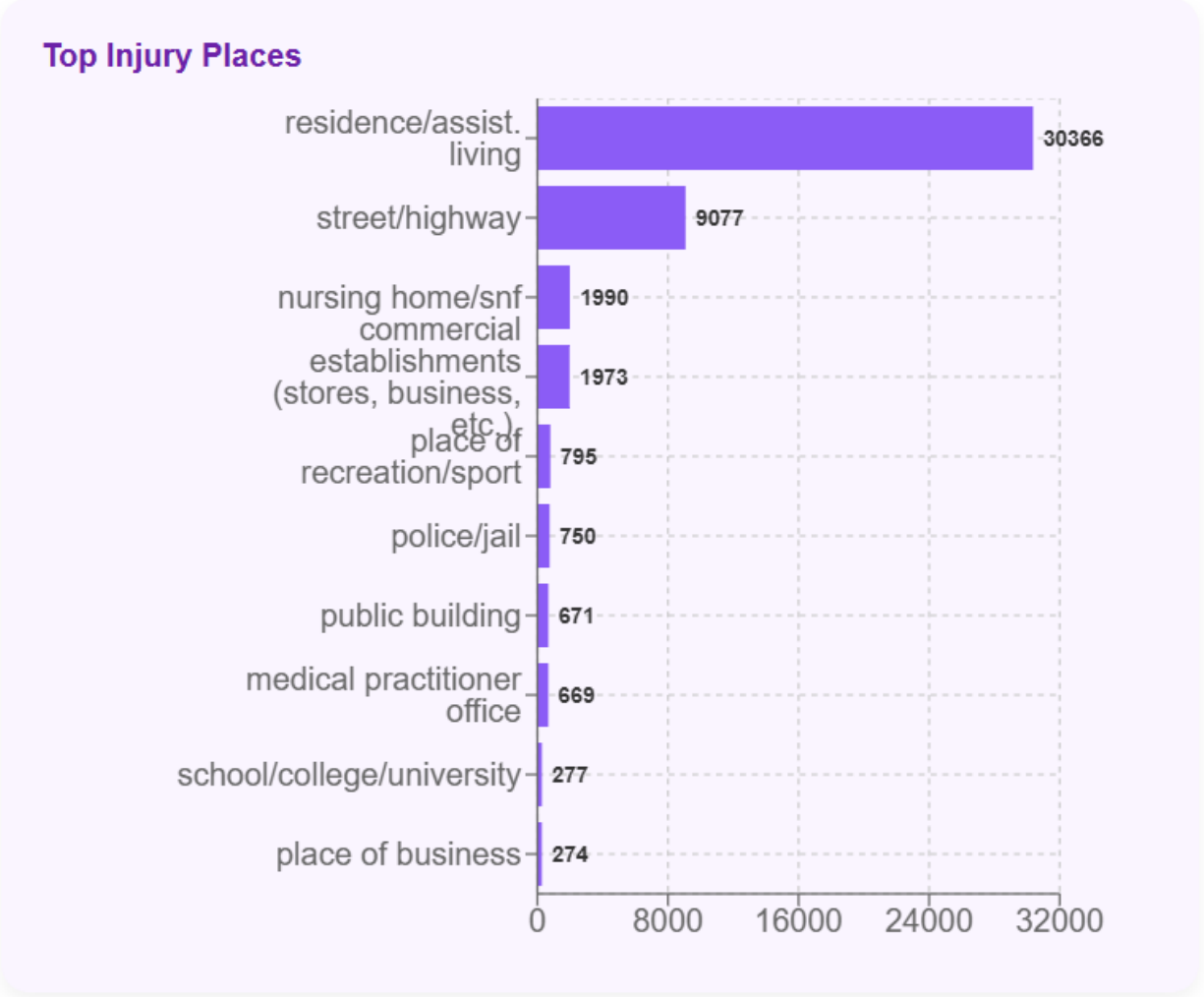



*Figure 4 Top 10 injury places*

➢ **Bar chart for top 10 injury places:**

- Residences and assisted living facilities account for by far the most injuries (around 30,366), greatly exceeding any other location.
- Streets and highways are the second most common injury locations (about 9,077), while nursing homes/SNFs and commercial establishments each contribute roughly 2,000 injuries.
- All other places—recreation/sport areas, police/jail, public buildings, medical offices, schools/universities, and business workplaces—have comparatively low counts, each under 1,000, indicating they are much less frequent sites of injury.

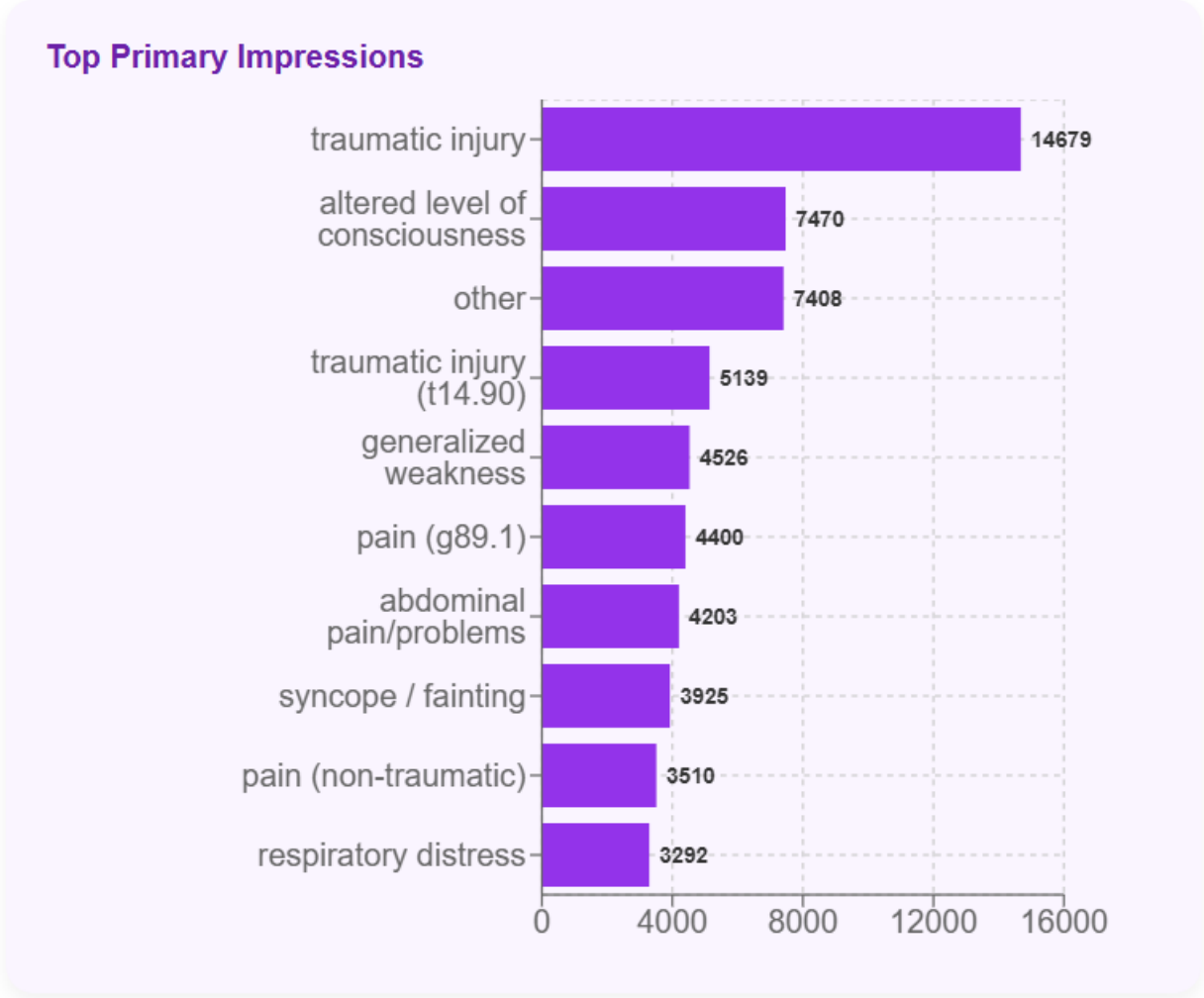



*Figure 5 Top 10 primary impressions*

➢ **Bar chart for top 10 primary impressions:**

- The chart ranks the most common EMS primary impressions, with "traumatic injury" clearly leading at about 14,679 cases, roughly double any other category.
- "Altered level of consciousness" and "other" are the next most frequent, each around 7,400–7,500 cases, indicating many calls are for nonspecific or neurologic concerns.
- Mid-range categories such as coded "traumatic injury (t14.90)", generalized weakness, pain (g89.1), and abdominal pain/problems each fall in the 4,000–5,000 case range, showing a large burden of general medical complaints.
- Syncope/fainting, non-traumatic pain, and respiratory distress are still substantial but lower, at roughly 3,300–3,900 cases, suggesting they are important but less frequent than injuries and generalized symptoms.

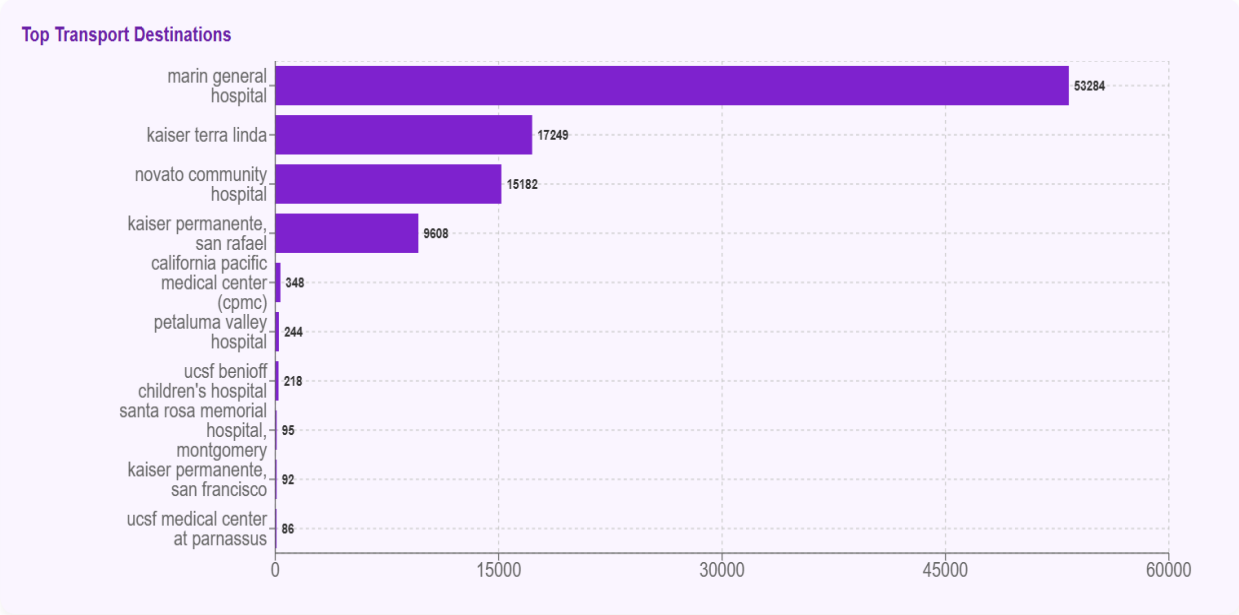



*Figure 6 Top 10 hospitals by incident transports*

- **Bar chart for top 10 transport destinations (hospitals):**

- The chart ranks the most common EMS transport destinations, measured by number of patient transports.
- Marin General Hospital dominates as the primary destination with 53,284 transports, far exceeding any other facility.
- Kaiser Terra Linda and Novato Community Hospital are the next major destinations, with about 17,249 and 15,182 transports respectively, indicating they are important regional centers but still secondary to Marin General.
- All remaining hospitals each receive fewer than 10,000 transports (most under 400), showing they play much smaller roles in overall EMS patient distribution.

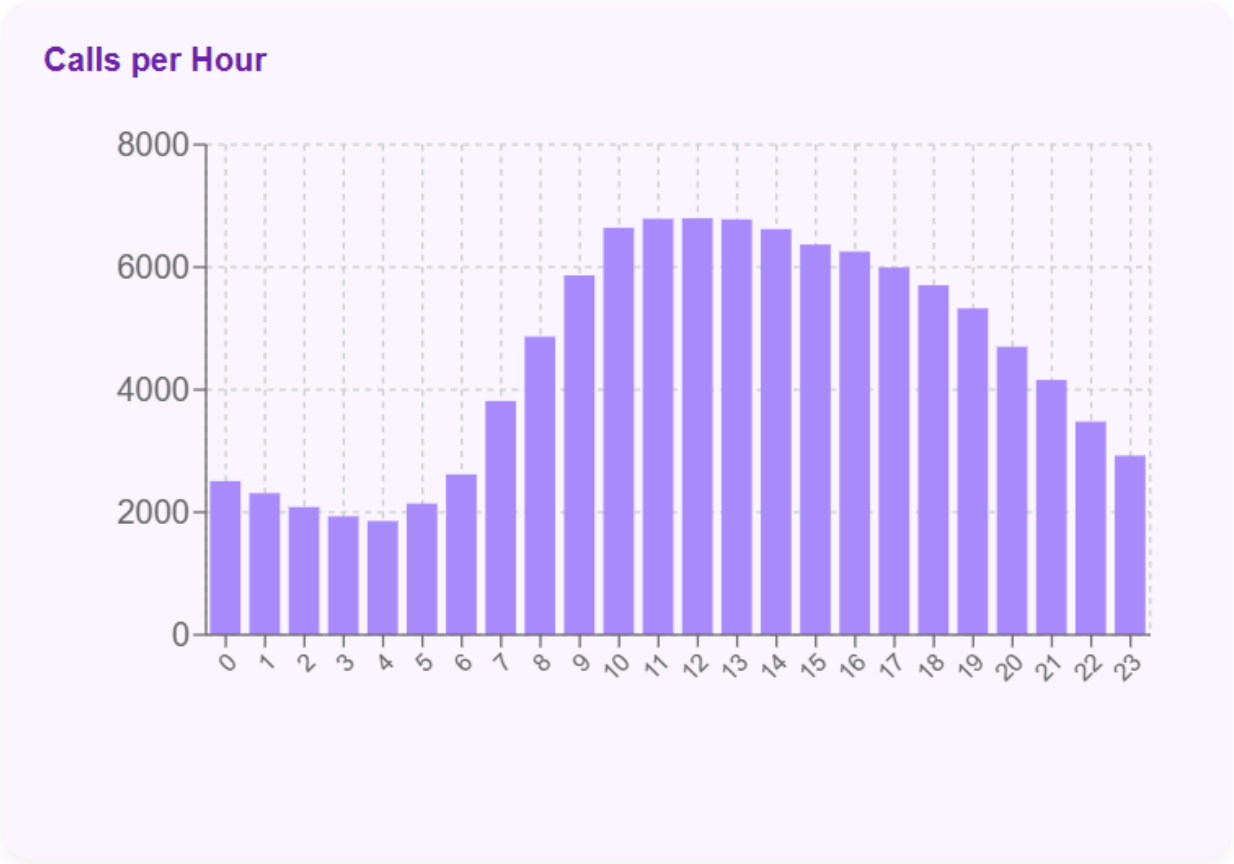



*Figure 7 call volume by hour distribution*

➢ **Bar chart for call volume by hours:**

- Call volume is lowest in the early morning hours (around 2–5 AM) and starts rising after about 6–7 AM.
- Calls peak late morning to early afternoon (roughly 10 AM–2 PM), where bar heights are around their maximum.
- After mid-afternoon, call counts gradually decline through evening and night, though they remain higher than the early-morning minimum.
- The maximum call volume was observed at 11 and 12 PM respectively.

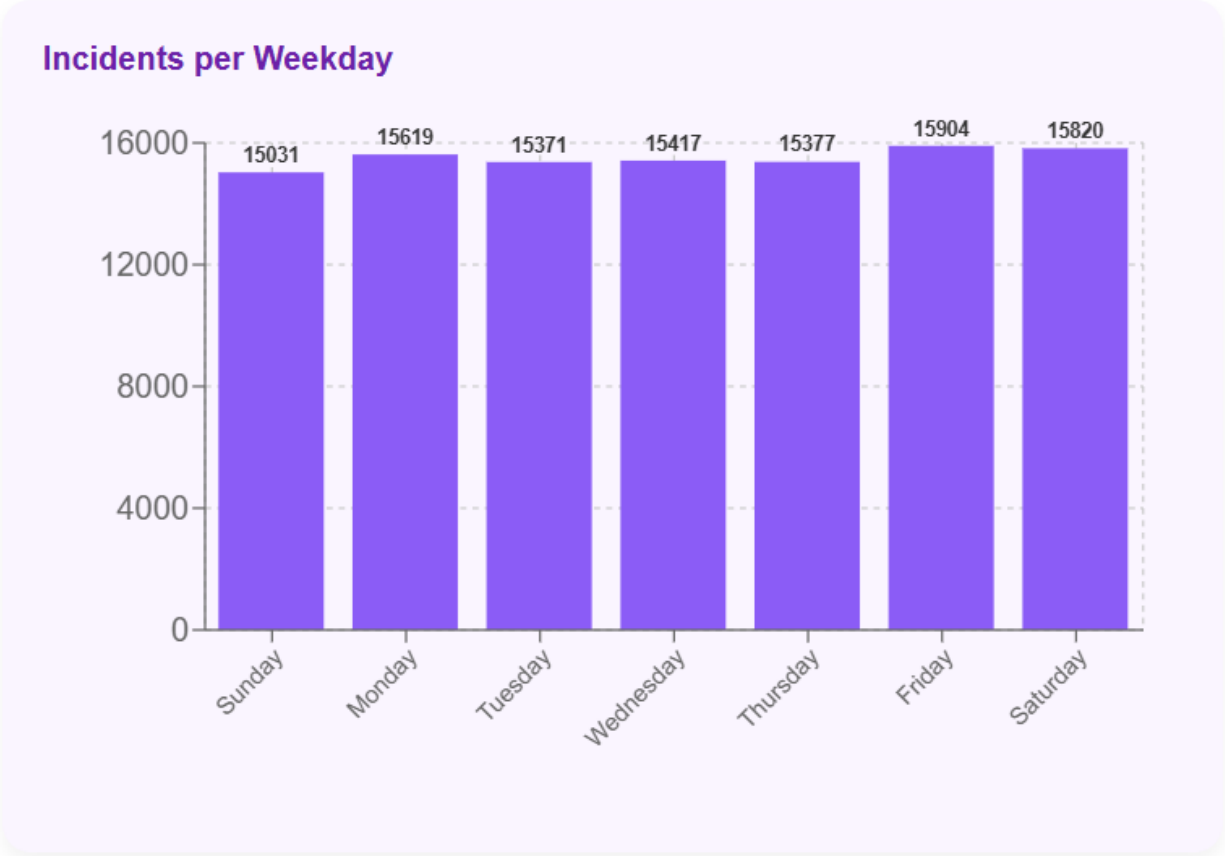



*Figure 8 call volume by days*

➢ **Bar chart for call volume by days of week:**

- Incident counts are fairly similar across all days, staying in a narrow band around 15,000–16,000.
- Friday and Saturday have the highest incident volumes (about 15,900–15,800), suggesting slightly busier end-of-week activity.
- Sunday has the lowest count (about 15,000), but the difference from weekdays is small, indicating no strong weekday–weekend pattern.

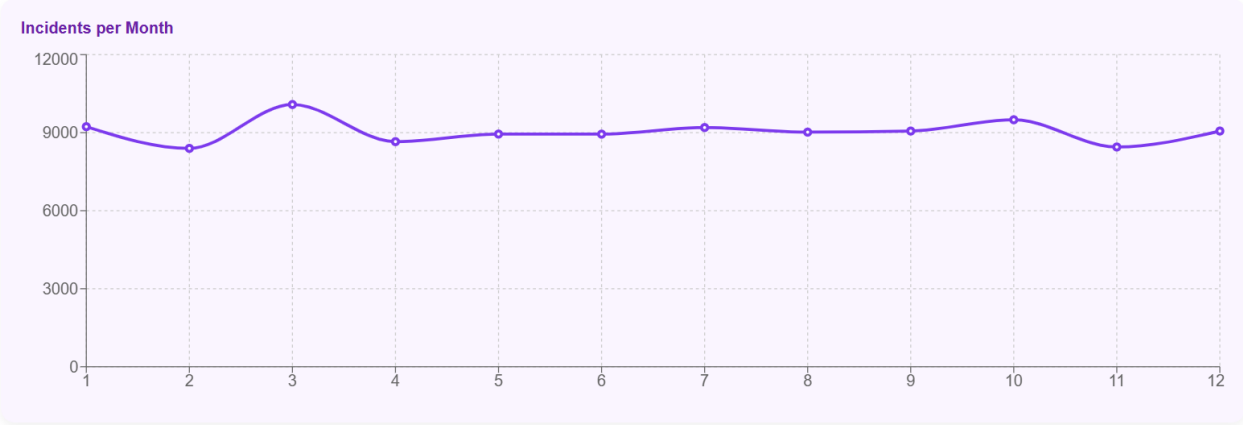
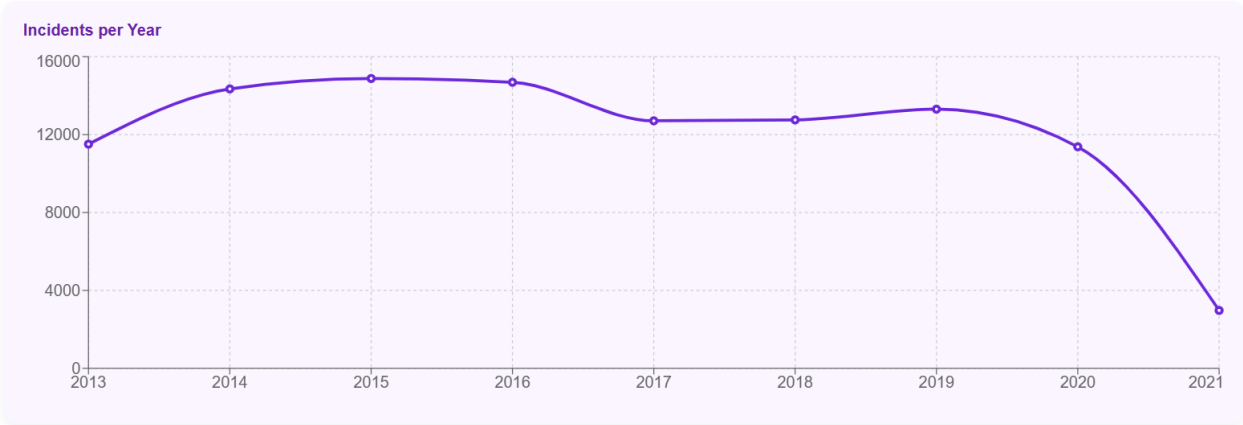



*Figure 9 incidents per month*

➢ **Line chart for call volume by month:**

- Incident counts stay within a relatively narrow band, roughly between about 8,500 and 10,000, indicating no extreme seasonal spikes or drops.
- There is a noticeable local peak around month 3 and a smaller rise around month 10, suggesting slightly busier periods in early spring and autumn.
- Months 2, 4, and 11 are on the lower side of the range, but overall the monthly variation in incidents is modest.




➢ **Line chart for call volume by year:**

- Incidents rise from 2013 to a peak around 2014–2015, then stay relatively high and stable through 2016.
- From 2017 onward, there is a moderate decline, with a small bump again around 2019 before dropping in 2020 and sharply in 2021.

- The low value in 2013 is expected because the data starts in March, so that year does not include a full 12 months.
- The very low value in 2021 is also expected because the data only goes up to April, so it represents roughly one-third of a normal year's incidents.

## Geospatial Hotspot Analysis

The objective of this geospatial analysis was to identify spatial hotspots of EMS activity and operational load using a hexagonal spatial indexing approach. Instead of relying on generic clustering or kernel density estimation, the analysis explicitly used the H3 hierarchical hexagonal grid system to aggregate incident-level data into consistent spatial units and derive interpretable hotspot metrics.

### Step-by-Step Methodology

**Step 1: Incident Data Loading and Validation**

EMS incident data were loaded from a cleaned CSV file. Only records containing valid latitude and longitude values (Incident_Latitude, Incident_Longitude) were retained, as these fields were mandatory for spatial indexing and aggregation.

This step ensured that all downstream geospatial computations were based on valid point-level incident locations.

**Step 2: On-Scene Time Computation**

For each EMS incident, on-scene time was computed using timestamp differences:

- Time_Arrived_on_Scene
- Time_Departed_from_the_Scene

The on-scene duration was calculated in minutes as:

On-scene time = (Time departed from scene − Time arrived on scene)

To handle data quality issues and extreme values, the resulting on-scene time was clipped to a valid range of 0 to 200 minutes. This feature represents operational workload and response complexity at the incident level.

```python
def compute_on_scene_time(df):
    print("[2] Computing on-scene time...")

    df["Time_Arrived_on_Scene"] = pd.to_datetime(df["Time_Arrived_on_Scene"])
    df["Time_Departed_from_the_Scene"] = pd.to_datetime(df["Time_Departed_from_the_Scene"])

    df["on_scene_time_min"] = (
        df["Time_Departed_from_the_Scene"] - df["Time_Arrived_on_Scene"]
    ).dt.total_seconds() / 60

    df["on_scene_time_min"] = df["on_scene_time_min"].clip(lower=0, upper=200)

    return df
```

*Figure 10 Computing on scene times code snippet*

**Step 3: H3 Hexagonal Spatial Indexing**

Each incident was assigned to a hexagonal spatial cell using the H3 indexing system at resolution 8. The H3 index was generated directly from the incident latitude and longitude coordinates.

This step transformed point-based incident data into a discrete spatial representation, enabling consistent aggregation across space while avoiding distortions caused by irregular administrative boundaries.

```python
def add_h3(df, resolution=8):
    print("[3] Adding H3 hex indices...")

    df["h3"] = df.apply(
        lambda r: h3.latlng_to_cell(
            r["Incident_Latitude"],
            r["Incident_Longitude"],
            resolution
        ),
        axis=1
    )

    return df
```

*Figure 11 H3 indexing code snippet*

**Step 4: Aggregation at the H3 Cell Level**

Incident-level records were aggregated by H3 cell to compute spatial summaries. For each hexagonal cell, the following metrics were calculated:

- Total number of incidents (incident count)
- Average on-scene time
- Minimum on-scene time
- Maximum on-scene time

This aggregation converted raw incident points into hex-level operational indicators that capture both demand intensity and operational effort within each spatial unit.

```
def compute_h3_aggregates(df):
    print("[4] Computing H3 aggregates...")

    agg = df.groupby("h3").agg(
        incidents=("Incident_Number", "count"),
        avg_on_scene=("on_scene_time_min", "mean"),
        min_on_scene=("on_scene_time_min", "min"),
        max_on_scene=("on_scene_time_min", "max"),
    ).reset_index()

    return agg
```

*Figure 12 Aggregating H3 cells code snippet*

**Step 5: Hexagonal Geometry Construction**

For visualization and spatial analysis, polygon geometries were generated for each H3 cell by converting H3 boundaries into geographic polygons. The resulting hexagonal geometries were stored in a GeoDataFrame using the WGS84 coordinate reference system (EPSG:4326).

This step enabled direct mapping and spatial visualization of aggregated EMS metrics.

```python
def add_hex_geometry(agg):
    def hex_to_poly(h):
        # H3 returns list of (lat, lon)
        boundary = h3.cell_to_boundary(h)
        coords = [(lon, lat) for lat, lon in boundary]
        return Polygon(coords)

    agg["geometry"] = agg["h3"].apply(hex_to_poly)
    return gpd.GeoDataFrame(agg, geometry="geometry", crs="EPSG:4326")
```

*Figure 13 Adding polygon shapes code snippet*

**Step 6: Output Generation**

The geospatial processing pipeline produced the following outputs:

1. Incident-level CSV containing original records with assigned H3 indices
2. Aggregated H3-level CSV containing computed spatial metrics
3. GeoJSON file containing hexagonal geometries and aggregated attributes

These outputs were designed for reuse in visualization, dashboarding, and downstream analytical pipelines.

**Visualization and Hotspot Identification**

The aggregated H3 GeoJSON was visualized using a choropleth map. Hexagonal cells were color-coded based on selected metrics, primarily incident count, to highlight spatial variations in EMS activity.

Cells with higher incident counts and elevated average on-scene times were interpreted as operational hotspots, indicating areas of sustained demand and increased response workload. Lower-intensity cells represented regions with comparatively fewer incidents and lower operational strain.


*Figure 14 HTML map made using chloropleth map*

**Results and Observations**

Based strictly on the aggregated H3 hex-level metrics, the top incident hotspots were identified by ranking hexagons in descending order of total incident count, with average on-scene time used as a secondary indicator of operational load.

The five highest-incident H3 cells observed in the dataset are listed below:

| RANK | H3 CELL ID | CITY | TOTAL INCIDENTS | AVG. ON-SCENE TIME (MIN) |
|---|---|---|---|---|
| **1** | 882830ba27fffff | San Rafael | 2935 | 11.07 |
| **2** | 882830b855fffff | San Rafael | 2207 | 13.92 |
| **3** | 88283085dbfffff | Mill Valley | 2192 | 13.48 |
| **4** | 882830ba23fffff | San Rafael | 2059 | 11.94 |
| **5** | 882830aa57fffff | Novato | 1849 | 11.46 |

These hexagons represent the most intense concentrations of EMS activity in the study area. Each of the top-ranked cells recorded incident volumes significantly higher than the remainder of the spatial grid, confirming strong spatial concentration rather than uniform distribution.

While incident count was the primary criterion for hotspot identification, average on-scene time provided additional context on operational workload. Several of the top hotspots exhibited elevated

average on-scene durations, indicating not only high demand but also sustained on-site engagement by EMS resources.

## Operational Efficiency Analysis:

The operational efficiency analysis was designed to quantify EMS performance using time-based KPIs derived directly from call lifecycle timestamps. The goal was not to infer outcomes or prescribe conclusions, but to compute standardized, dashboard-ready indicators that allow visual inspection and interpretation of system efficiency trends.

### Step-by-Step Methodology

**Step 1: Data Loading**

The EMS dataset is loaded from a cleaned CSV file. The pipeline requires the presence of core timestamp fields representing key stages of the EMS response lifecycle.

The following columns are parsed as datetime objects (where available):

- Time_Call_Was_Received
- Time_Vehicle_was_Dispatched
- Time_Arrived_on_Scene
- Time_Departed_from_the_Scene

Invalid or unparseable timestamps are coerced to null values.

**Step 2: Core Time Metric Computation**

Using the parsed timestamps, the following duration-based operational metrics are computed at the incident level:

- **Response Time**
  Time between vehicle dispatch and arrival on scene.
- **Turnout Time**
  Time between call receipt and vehicle dispatch.
- **On-Scene Time**
  Time between arrival on scene and departure from scene.
- **Total Call Cycle Time**
  Time between call receipt and departure from scene.

All durations are computed in minutes.

```
date_cols = [
    "Time_Call_Was_Received",
    "Time_Vehicle_was_Dispatched",
    "Time_Arrived_on_Scene",
    "Time_Departed_from_the_Scene",
]
for c in date_cols:
    if c in df.columns:
        df[c] = pd.to_datetime(df[c], errors="coerce")

# compute core times
df["response_time_min"] = (df["Time_Arrived_on_Scene"] - df["Time_Vehicle_was_Dispatched"]).dt.total_seconds() / 60 if "Time_Arr
df["turnout_time_min"] = (df["Time_Vehicle_was_Dispatched"] - df["Time_Call_Was_Received"]).dt.total_seconds() / 60 if "Time_Veh
df["call_cycle_time_min"] = (df["Time_Departed_from_the_Scene"] - df["Time_Call_Was_Received"]).dt.total_seconds() / 60 if "Time
df["on_scene_time_min"] = (df["Time_Departed_from_the_Scene"] - df["Time_Arrived_on_Scene"]).dt.total_seconds() / 60 if "Time_De
```

*Figure 15 Time columns computation code snippet*

### Step 3: Data Quality Filtering

To ensure robustness of downstream metrics:

- Records with missing core duration values are removed.
- Negative durations are excluded.
- Durations exceeding 300 minutes are filtered out to eliminate extreme or erroneous values.

This step ensures that all operational metrics are computed from valid, realistic observations.

## Step 4: KPI Computation

```python
def compute_kpis(df: pd.DataFrame) -> Dict[str, Any]:
    avg_response = df["response_time_min"].mean()
    p90 = df["response_time_min"].quantile(0.9)
    avg_on_scene = df["on_scene_time_min"].mean()
    avg_cycle = df["call_cycle_time_min"].mean()
    avg_turnout = df["turnout_time_min"].mean() if "turnout_time_min" in df.columns else None

    # calls per day
    calls_per_day = None
    if "Time_Call_Was_Received" in df.columns:
        daily_counts = df.groupby(df["Time_Call_Was_Received"].dt.date).size()
        if len(daily_counts):
            calls_per_day = float(daily_counts.mean())

    # busiest hour
    busiest_hour = None
    if "Time_Call_Was_Received" in df.columns:
        hours = df["Time_Call_Was_Received"].dt.hour.value_counts()
        if not hours.empty:
            busiest_hour = f"{int(hours.idxmax()):02d}:00"

    # busiest city
    busiest_city = None
    if "Incident_City" in df.columns:
        city_counts = df["Incident_City"].value_counts()
        if not city_counts.empty:
            busiest_city = str(city_counts.idxmax())

    # SLA 8 min compliance
    sla_8_min_pct = ((df["response_time_min"] <= 8).sum() / len(df) * 100) if "response_time_min" in df.columns else None

    return {
        "avg_response_time": safe_round(avg_response),
        "p90_response_time": safe_round(p90),
        "avg_on_scene_time": safe_round(avg_on_scene),
        "avg_total_cycle_time": safe_round(avg_cycle),
        "calls_per_day": safe_round(calls_per_day),
```

*Figure 16 KPI computation code*

| KPI Name | Description | Unit | Data Type |
|---|---|---|---|
| **Average Response Time** | Mean time between vehicle dispatch and arrival on scene | Minutes | Float |
| **90th Percentile Response Time** | Response time below which 90% of incidents fall | Minutes | Float |
| **Average On-Scene Time** | Mean duration spent on scene per incident | Minutes | Float |
| **Average Total Call Cycle Time** | Mean time from call receipt to departure from scene | Minutes | Float |
| **Average Turnout Time** | Mean time from call receipt to vehicle dispatch | Minutes | Float |

| | | | |
|---|---|---|---|
| **Calls per Day** | Average number of EMS calls handled per day | Count | Float |
| **Busiest Hour** | Hour of day with highest call volume | Hour (HH:00) | String |
| **Busiest City** | City with the highest number of incidents | Category | String |
| **SLA 8-Minute Compliance** | Percentage of incidents with response time ≤ 8 minutes | Percentage | Float |

These KPIs form the headline indicators displayed in the operational efficiency dashboard.

**Step 5: Temporal Trend Aggregation**

Operational metrics are aggregated over time to support trend visualization.

- If sufficient date coverage is available, metrics are aggregated daily
- Otherwise, aggregation is performed monthly

For each time period, the following are computed:

- Average response time
- Average on-scene time
- Average total cycle time
- Average turnout time
- Incident count

```python
# ---------------- TIME TRENDS ----------------
def compute_time_trends(df: pd.DataFrame, prefer="D") -> List[Dict[str, Any]]:
    if df.empty or "Time_Call_Was_Received" not in df.columns:
        return []

    total = len(df)
    non_null = df["Time_Call_Was_Received"].notna().sum()
    use_month = (non_null / total if total else 0) < 0.5

    if use_month:
        df = df.copy()
        df["month"] = df["Time_Call_Was_Received"].dt.to_period("M").astype(str)
        grouped = df.groupby("month").agg(
            avg_response=("response_time_min", "mean"),
            avg_on_scene=("on_scene_time_min", "mean"),
            avg_cycle=("call_cycle_time_min", "mean"),
            avg_turnout=("turnout_time_min", "mean"),
            incident_count=("Incident_Number", "count")
        ).reset_index()
        grouped = grouped.rename(columns={"month": "period"})
        for c in ["avg_response","avg_on_scene","avg_cycle","avg_turnout"]:
            grouped[c] = grouped[c].round(2)
        return grouped.to_dict(orient="records")
    else:
        df = df.copy()
        df["date"] = df["Time_Call_Was_Received"].dt.date.astype(str)
        grouped = df.groupby("date").agg(
            avg_response=("response_time_min", "mean"),
            avg_on_scene=("on_scene_time_min", "mean"),
            avg_cycle=("call_cycle_time_min", "mean"),
            avg_turnout=("turnout_time_min", "mean"),
            incident_count=("Incident_Number", "count")
        ).reset_index()
        for c in ["avg_response","avg_on_scene","avg_cycle","avg_turnout"]:
            grouped[c] = grouped[c].round(2)
        return grouped.to_dict(orient="records")
```

*Figure 17 Trends over time code*

**Step 6: Distribution Construction**

To support distributional analysis:

- Response time, on-scene time, and call cycle time arrays are extracted.
- Each metric is capped at its 99th percentile to limit the impact of extreme values.
- Histogram bins (30 bins) are generated from the capped distributions.

- Capped raw values are preserved for flexible visualization.

This step enables consistent comparison of operational spread and variability.

```python
# ---------------- DISTRIBUTIONS ----------------
def compute_distributions(df: pd.DataFrame) -> Dict[str, Any]:
    out = {}
    def safe_array(col):
        if col not in df.columns:
            return np.array([])
        vals = df[col].dropna().astype(float).values
        return vals

    resp = safe_array("response_time_min")
    onscene = safe_array("on_scene_time_min")
    cycle = safe_array("call_cycle_time_min")

    p99_resp = pctile_cap(resp, 99) or 0.0
    p99_scene = pctile_cap(onscene, 99) or 0.0
    p99_cycle = pctile_cap(cycle, 99) or 0.0

    resp_capped = np.where(resp>p99_resp,p99_resp,resp) if resp.size else np.array([])
    scene_capped = np.where(onscene>p99_scene,p99_scene,onscene) if onscene.size else np.array([])
    cycle_capped = np.where(cycle>p99_cycle,p99_cycle,cycle) if cycle.size else np.array([])

    out["response_hist"] = histogram_bins_from_array(resp_capped, bins=30)
    out["on_scene_hist"] = histogram_bins_from_array(scene_capped, bins=30)
    out["cycle_hist"] = histogram_bins_from_array(cycle_capped, bins=30)

    out["response_times_capped"] = [round(float(x),2) for x in resp_capped.tolist()] if resp_capped.size else []
    out["on_scene_times_capped"] = [round(float(x),2) for x in scene_capped.tolist()] if scene_capped.size else []
    out["cycle_times_capped"] = [round(float(x),2) for x in cycle_capped.tolist()] if cycle_capped.size else []

    out["caps"] = {
        "p99_response": safe_round(p99_resp),
        "p99_on_scene": safe_round(p99_scene),
        "p99_cycle": safe_round(p99_cycle)
    }
```

*Figure 18 Distributions of time columns code*

**Step 7: Response Time Percentiles**

Additional response-time percentiles are computed to characterize performance spread:

- 50th percentile
- 75th percentile
- 90th percentile
- 95th percentile
- Maximum observed response time

These values are exported as a dedicated percentile summary.

**Step 8: SLA Delay Bucket Analysis**

For incidents exceeding the 8-minute SLA:

1. Delay beyond SLA is computed.
2. Only delayed incidents are retained.
3. Delays are categorized into predefined buckets:
    - 0–5 minutes late
    - 5–10 minutes late
    - 10–15 minutes late
    - 15–30 minutes late
    - 30+ minutes late

The number of incidents in each delay bucket is counted.

```python
# ---------------- DELAY BUCKETS WITH SLA (ONLY LATE CALLS) ----------------
def compute_delay_buckets(df: pd.DataFrame, sla: float = 8) -> list[dict]:
    if df.empty or "response_time_min" not in df.columns:
        return []

    # 1️⃣ Compute delay over SLA
    df['delay'] = df['response_time_min'] - sla
    df['delay'] = df['delay'].apply(lambda x: x if x > 0 else 0)

    # 2️⃣ Keep only delayed incidents
    df_late = df[df['delay'] > 0]
    if df_late.empty:
        return [{"delay_bucket": label, "count": 0} for label in ["0–5 min","5–10","10–15","15–30","30+"]]

    # 3️⃣ Define delay buckets
    bins = [0,5,10,15,30,999]  # minutes late
    labels = ["0–5 min","5–10","10–15","15–30","30+"]

    # 4️⃣ Categorize delays into buckets
    cat = pd.cut(df_late['delay'], bins=bins, labels=labels, right=False)

    # 5️⃣ Count number of incidents in each bucket
    dist = cat.value_counts().reindex(labels).fillna(0).astype(int)

    # 6️⃣ Return as list of dicts
    return [{"delay_bucket": label, "count": int(dist[label])} for label in labels]
```

*Figure 19 Delay bucket analysis code*

**Step 9: City-Level Aggregation**

Operational metrics are aggregated at the city level using the Incident_City field.

For each city, the pipeline computes:

- Total number of incidents
- Average response time
- 90th percentile response time
- Average turnout time

This enables spatial comparison of operational performance without introducing geospatial modeling.

**Step 10: Hourly Response Analysis**

Incidents are grouped by the hour of call receipt.

For each hour (formatted as HH:00), the following are computed:

- Average response time
- Total number of incidents

This step captures diurnal patterns in workload and response performance.

**Step 11: JSON Output Generation**

All computed metrics are exported as structured JSON files, including:

- kpis.json
- time_trends.json
- distributions.json
- response_percentiles.json
- delay_buckets.json
- city_summary.json
- hourly_response.json

These outputs serve as the sole input to the operational efficiency dashboard.

## Results and Observations

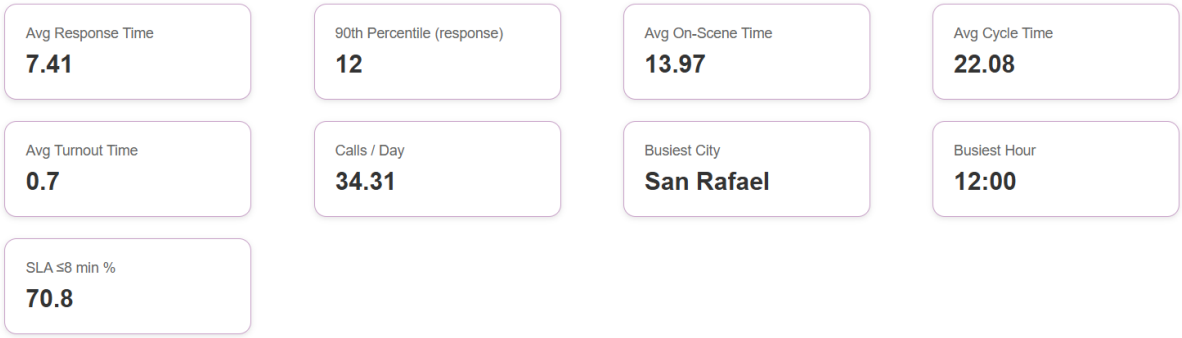



*Figure 20 KPI cards*

- The EMS system handles an average response time of 7.41 minutes, with a 90th percentile response time of 12 minutes, meaning 90% of calls are reached within 12 minutes.
- Crews spend about 13.97 minutes on scene, and the total cycle time from dispatch to unit availability again averages 22.08 minutes, suggesting roughly one call turnaround every 20–25 minutes per unit.
- The average turnout time (from dispatch to wheels rolling) is 0.7 minutes, indicating crews leave the station in well under a minute on average, which reflects strong station readiness.
- The system manages about 34.31 calls per day, with San Rafael identified as the busiest city and 12:00 (noon) as the busiest hour, highlighting where and when demand is most concentrated.
- The SLA ≤ 8 min % of 70.8 shows that about seven out of ten calls meet the target of arriving within eight minutes, while nearly three out of ten exceed this benchmark, revealing room for response-time improvement.

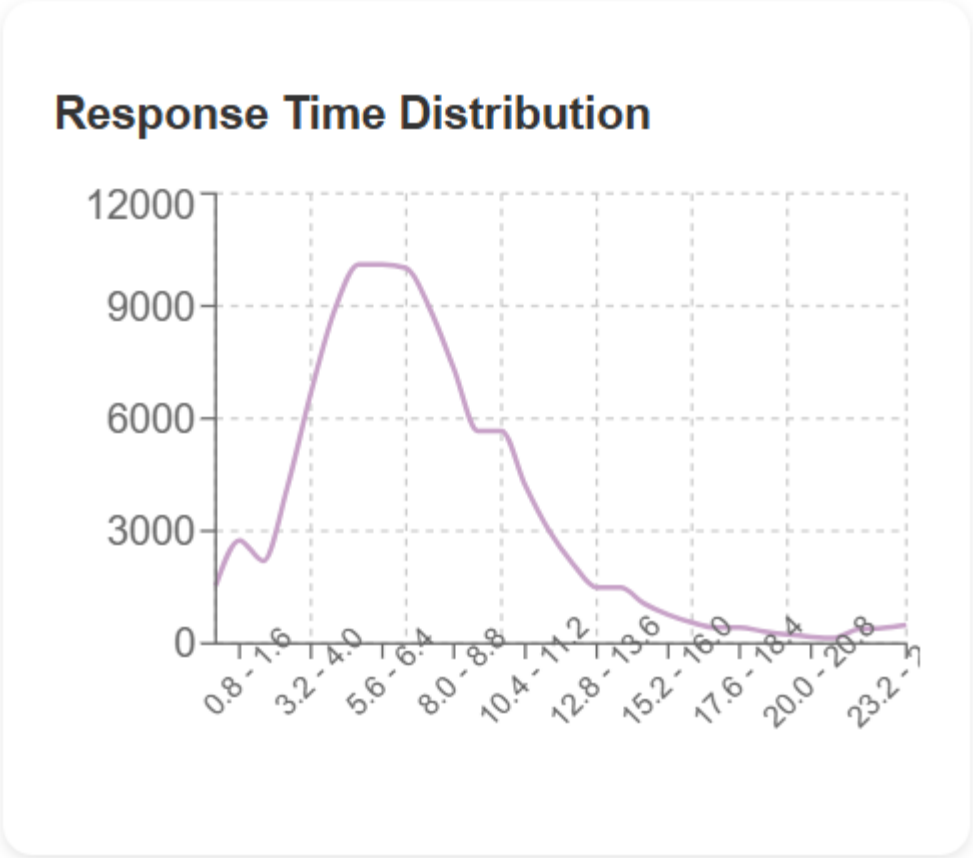



*Figure 21 Distribution of response time*

- Most responses cluster between about 3-8 minutes, where the curve reaches its highest point, confirming that typical responses fall well within the 8-minute SLA target.
- The frequency drops sharply after around 8-10 minutes, with relatively few calls taking longer than 12-14 minutes, and only a small tail extending beyond 20 minutes.
- Overall, the shape suggests a right-skewed distribution: fast responses are common, while very long response times are rare but still present and worth monitoring.

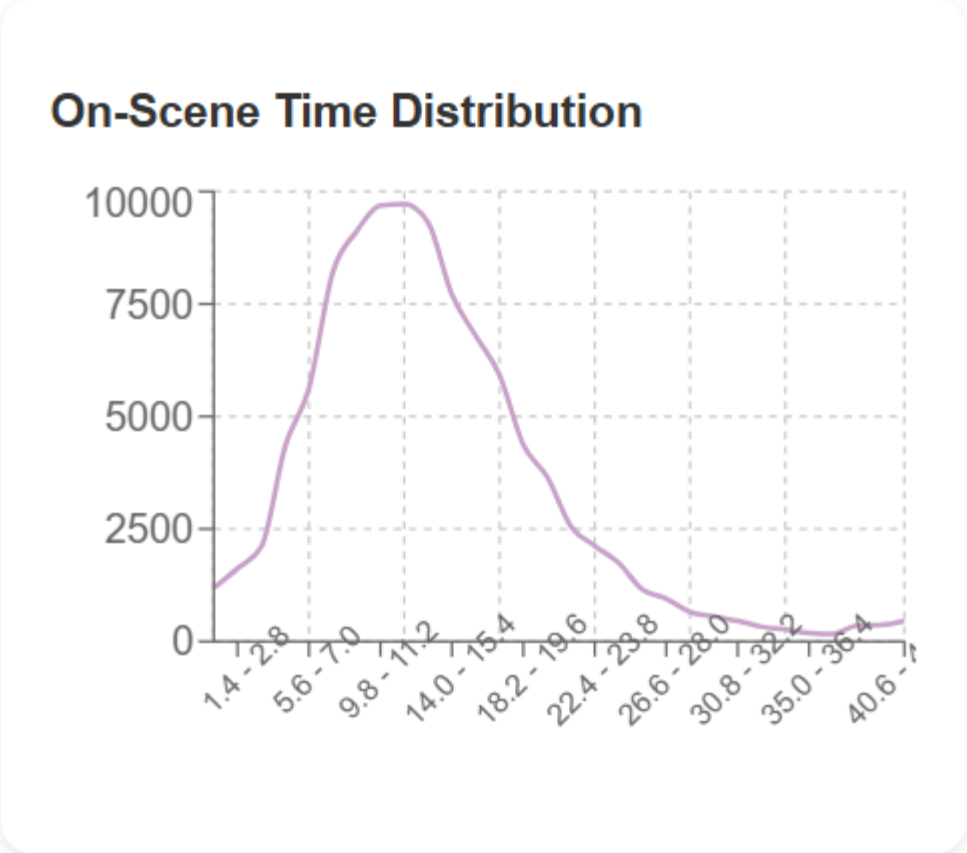



*Figure 22 Distribution of on-scene time*

- Most calls have on-scene times roughly between 9-18 minutes, where the curve is highest, aligning with the average of about 14 minutes you reported.
- The frequency falls off steadily after about 18-20 minutes, indicating longer on-scene times are progressively less common.
- There is a long right tail extending beyond 30-40 minutes, showing a small subset of complex or resource-intensive cases where crews remain on scene for a prolonged period.

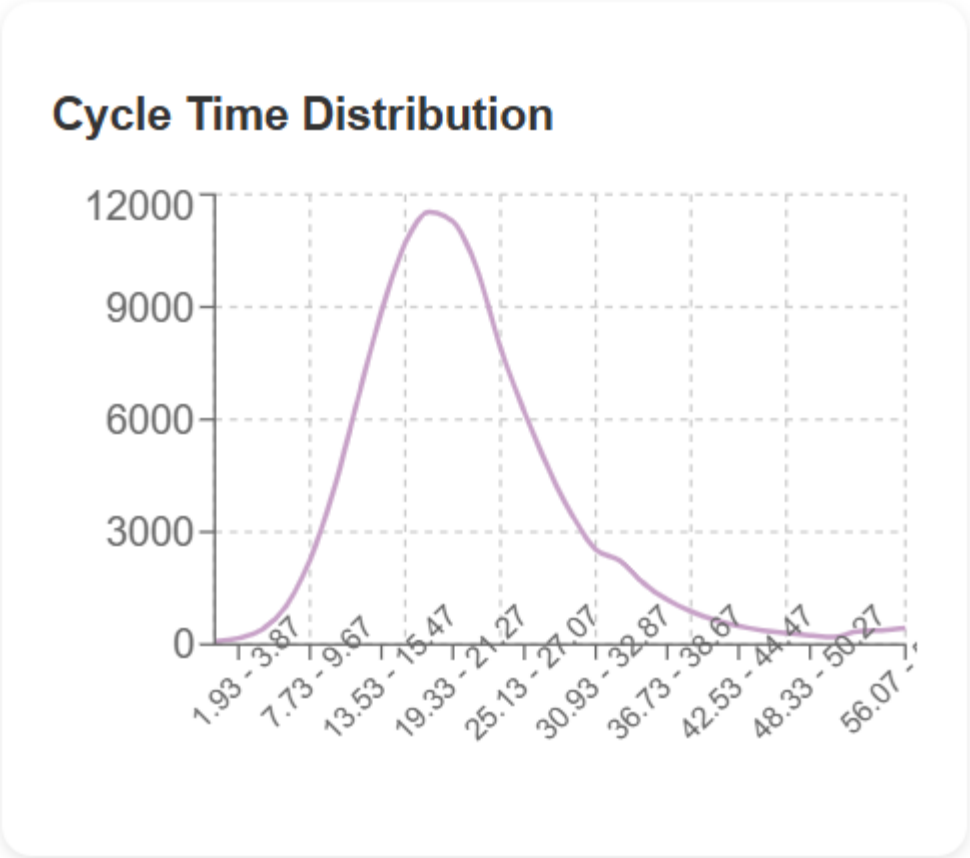



*Figure 23 Call cycle time distribution*

- Most calls have cycle times roughly between 15-25 minutes, where the curve peaks, which matches the earlier reported average cycle time of about 22 minutes.
- The frequency falls off quickly after around 25-30 minutes, indicating that significantly longer cycles are relatively uncommon.
- A long right tail extends out beyond 45-55 minutes, representing a small proportion of complex or distant calls that keep units busy for much longer than typical.

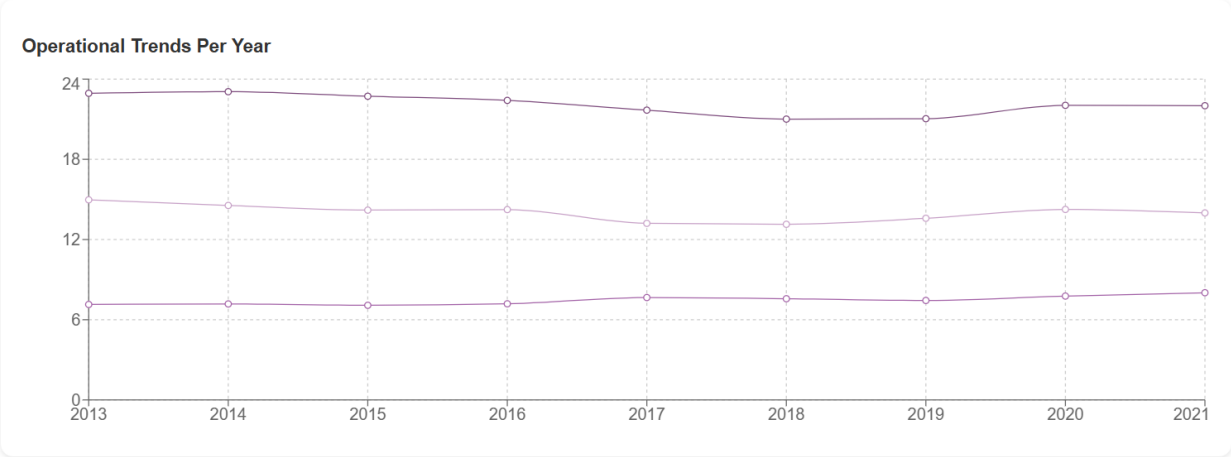



*Figure 24 Operational Trends over year*

- This multi-line chart tracks yearly operational metrics from 2013–2021: response time (lowest line), on-scene time (middle line), and cycle time (top line).
- Focusing on full years 2014–2020, average cycle time generally decreases through 2017–2018, indicating improved overall efficiency, then rises slightly in 2019–2020.
- On-scene time trends slightly downward from 2014 to around 2017–2018, followed by a modest increase afterward, suggesting crews began spending somewhat longer on scene again.
- Response time shows a slow upward drift across the full years, implying units take a bit longer to arrive at incidents, even as cycle time had improved; 2013 and 2021 should be viewed as partial, less-comparable reference points.

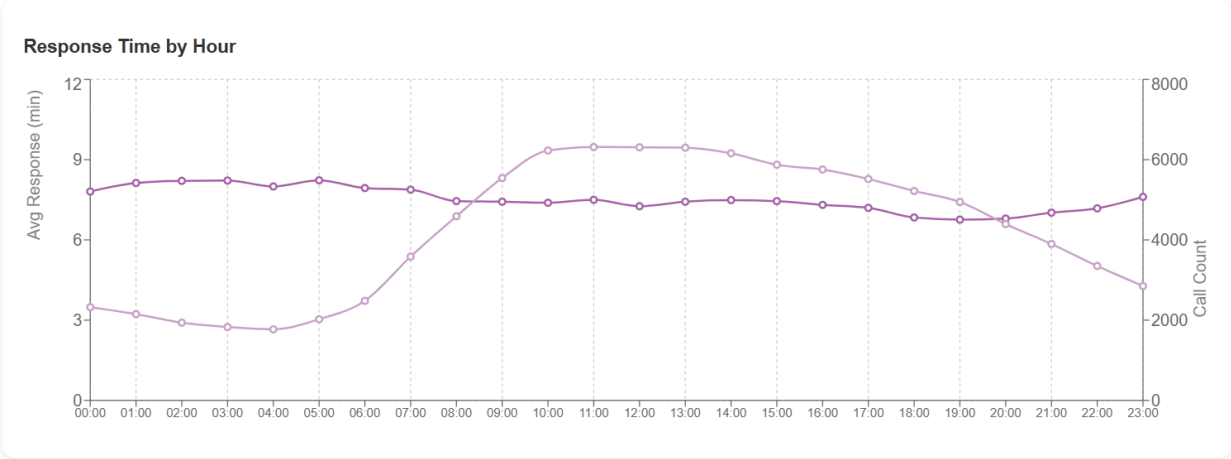



*Figure 25 Response time by hour of the day*

- This dual-axis chart shows average response time by hour (left axis) and call volume by hour (right axis), making it a bivariate time-of-day view.
- Response times stay fairly stable, around 7–8 minutes across most hours, with only small fluctuations between late night and daytime.
- Call volume is lowest in the early morning (around 02:00–05:00), rises sharply after 06:00, and peaks late morning to early afternoon, then gradually declines into the evening and night.
- Despite call counts increasing dramatically during the daytime peak, response times do not rise proportionally, suggesting the system generally absorbs higher demand without major degradation in performance.

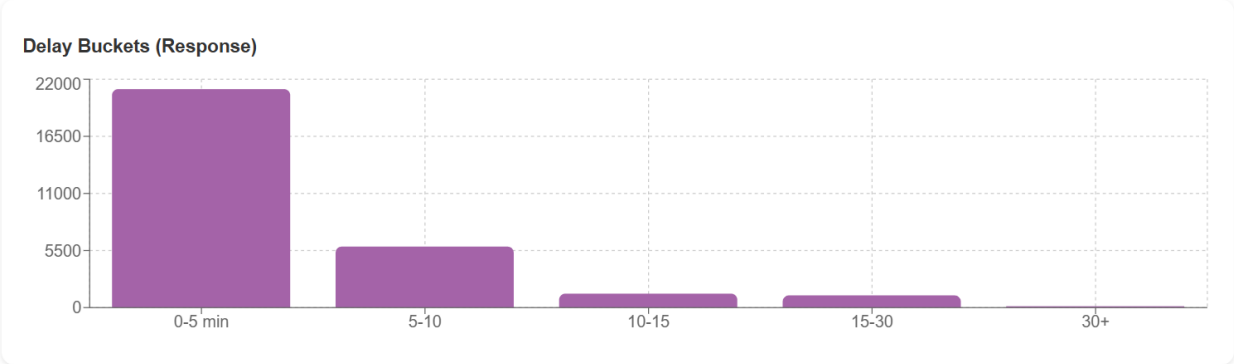



*Figure 26 Delay buckets based on SLA*

- This bar chart shows response delays grouped into time buckets (0-5, 5-10, 10-15, 15-30, and 30+ minutes).
- The vast majority of calls fall in the 0-5 minute bucket, indicating that most responses start very quickly after dispatch.
- A smaller but still notable share is in the 5-10 minute range, while only a few calls experience delays longer than 10-15 minutes.
- Very long delays (15-30 minutes and 30+ minutes) are rare, but their presence highlights a small subset of outlier incidents that may warrant operational review.

## Risk Classification Analysis:

The objective of the risk scoring pipeline is to assign a categorical risk level (LOW, MEDIUM, HIGH) to EMS incidents by combining medical text information with operational features. The pipeline is designed to generate labels in a weakly supervised manner using clustering and LLM-assisted summarization, followed by supervised model training and evaluation using a stratified data split.

### Step-by-Step Methodology

**Step 1: Data Loading**

The pipeline begins by loading the cleaned EMS dataset from a CSV file. All records are retained at this stage to ensure complete coverage during clustering and label generation.

**Step 2: Medical Text Construction**

For each incident, a consolidated medical text field is constructed by combining:

- Primary medical impression
- EMS protocol used
- Patient age group (derived from numeric age)

Patient age is discretized into clinically meaningful categories (infant, child, teen, adult, elderly). The resulting medical_text field serves as the semantic representation of incident severity and context.

```
# --------------------- BUILD MEDICAL TEXT ---------------------
def build_medical_text(df: pd.DataFrame) -> pd.DataFrame:
    df = df.copy()
    df["primary_impression"] = df.get("Primary_Impression", "").fillna("").astype(str)
    df["protocol_used"] = df.get("Protocol_Used_by_EMS_Personnel", "").fillna("").astype(str)
    df["Patient_Age"] = df.get("Patient_Age", pd.Series([np.nan]*len(df)))
    df["age_group"] = df["Patient_Age"].apply(preprocess_age)
    df["medical_text"] = (
        df["primary_impression"].str.lower().str.strip().fillna("") + " | " +
        df["protocol_used"].str.lower().str.strip().fillna("") + " | " +
        df["age_group"].astype(str)
    )
    return df
```

*Figure 27 Medical text construction code snippet*

### Step 3: Text Vectorization and Clustering

The medical text is converted into numerical representations using TF–IDF vectorization with unigrams and bigrams.

K-means clustering is then applied to the TF–IDF vectors to group incidents into a fixed number of clusters. Clustering is performed on the entire dataset to ensure global consistency of cluster structure.

Each incident is assigned a cluster_id corresponding to its medical-text-based cluster.

```
# ---------------------- CLUSTERING -------------------------
def cluster_medical_text(df: pd.DataFrame, n_clusters: int = N_CLUSTERS) -> pd.DataFrame:
    tfidf = TfidfVectorizer(max_features=8000, stop_words="english", ngram_range=(1,2))
    X = tfidf.fit_transform(df["medical_text"].fillna(""))
    joblib.dump(tfidf, os.path.join(OUT_DIR, "tfidf_vectorizer.joblib"))
    kmeans = KMeans(n_clusters=n_clusters, random_state=RANDOM_STATE, n_init=10)
    df["cluster_id"] = kmeans.fit_predict(X)
    joblib.dump(kmeans, os.path.join(OUT_DIR, "kmeans_model.joblib"))
    return df
```

*Figure 28 K-means clustering code snippet*

### Step 4: Cluster Sampling

From each cluster, a fixed number of representative incidents are randomly sampled. These samples are used to characterize the cluster content without exposing the full dataset to the labeling stage.

```
# ---------------------- SAMPLE -------------------------------
def sample_clusters(df: pd.DataFrame, samples_per_cluster: int = SAMPLES_PER_CLUSTER):
    out = {}
    for cid, sub in df.groupby("cluster_id"):
        n = min(len(sub), samples_per_cluster)
        out[int(cid)] = sub.sample(n, random_state=RANDOM_STATE).to_dict(orient="records")
    return out
```

*Figure 29 Sampling from clusters code snippet*

### Step 5: LLM-Based Cluster Summarization

Each cluster sample is passed to a large language model (LLM), which generates a concise natural-language summary describing the typical medical scenario represented by the cluster. The LLM is used exclusively for cluster-level summarization, not for per-incident labeling. All cluster summaries are persisted for auditability.

### Step 6: Risk Label Assignment

Each cluster summary is mapped to a categorical risk label (LOW, MEDIUM, HIGH) using a deterministic keyword-based extraction process. Once a risk label is assigned to a cluster, the label is propagated to all incidents belonging to that cluster, producing an incident-level risk_label field. This approach results in a weakly supervised labeling scheme grounded in semantic grouping rather than manual annotation.

```
# ---------------------- LABEL EXTRACTION ----------------------
def extract_label(summary: str) -> str:
    s = (summary or "").lower()
    high_kw = ["cardiac arrest", "life-threatening", "severe", "airway", "major", "unconscious", "seizure", "obvious death", "dead on arrival", "expired"]
    low_kw = ["low risk", "minor", "no injury", "epistaxis", "nosebleed", "public assist", "lift assist", "non traumatic", "non-traumatic", "no treatment"]
    med_kw = ["moderate", "stable", "transport", "observation", "requires transport", "non life-threatening"]
    if any(k in s for k in high_kw):
        return "HIGH"
    if any(k in s for k in low_kw):
        return "LOW"
    if any(k in s for k in med_kw):
        return "MEDIUM"
    if "hospital" in s or "ed" in s or "transport" in s:
        return "MEDIUM"
    return "MEDIUM"
```

*Figure 30 Label extraction code snippet*

### Step 7: Cluster Diagnostics

For transparency and validation, diagnostic metadata is generated for each cluster, including:

- Assigned risk label
- Cluster size
- Example incidents from the cluster

These diagnostics support traceability between cluster semantics and assigned risk categories.

```
# ---------------------- CLUSTER DIAGNOSTICS --------------------
def cluster_diagnostics(df: pd.DataFrame, label_map: Dict[int, str], out_dir: str):
    diagnostics = {}
    for cid, sub in df.groupby("cluster_id"):
        lbl = label_map.get(int(cid), "MEDIUM")
        example_rows = sub.head(3)[["Incident_Number","primary_impression","protocol_used","age_group"]].to_dict(orient="records")
        diagnostics[int(cid)] = {
            "assigned_label": lbl,
            "cluster_size": int(len(sub)),
            "examples": example_rows
        }
    with open(os.path.join(out_dir, "cluster_diagnostics.json"), "w", encoding="utf-8") as fh:
        json.dump(diagnostics, fh, indent=2, ensure_ascii=False)
    print(f"Wrote cluster diagnostics to {os.path.join(out_dir, 'cluster_diagnostics.json')}")
```

*Figure 31 Cluster diagnostics code snippet*

### Step 8: Stratified Train–Test Split

After all incidents are labeled, the dataset is split into training and test subsets using a stratified split:

- 70% training data
- 30% test data

Stratification is performed on the risk_label to preserve the LOW/MEDIUM/HIGH class proportions in both subsets. This ensures fair evaluation and prevents class imbalance artifacts.

```
# Stratified split to maintain class proportions
df_train, df_test = train_test_split(
    df,
    test_size=0.3,
    random_state=RANDOM_STATE,
    stratify=df["risk_label"]  # <-- keeps LOW/MEDIUM/HIGH proportions equal
)

print(f"Train set: {len(df_train):,} rows")
print(f"Train distribution:\n{df_train['risk_label'].value_counts()}\n")
print(f"Test set: {len(df_test):,} rows")
print(f"Test distribution:\n{df_test['risk_label'].value_counts()}\n")

train_path = os.path.join(out_dir, "train_set.csv")
test_path = os.path.join(out_dir, "test_set.csv")
df_train.to_csv(train_path, index=False)
df_test.to_csv(test_path, index=False)
```

*Figure 32 Test- Train split code snippet*

**Step 9: Feature Engineering for Classification**

The supervised classifier uses a combination of medical, operational, and categorical features:

**Numeric operational features**:

- Response time
- Turnout time
- On-scene time
- Total call cycle time
- Patient age

**Medical text features**:

- TF–IDF representations derived from the constructed medical text

**Categorical features**:

- Patient gender
- Incident county
- Primary medical impression

Categorical variables are encoded using one-hot encoding, and all feature groups are concatenated into a unified feature matrix.

**Step 10: Supervised Model Training**

A multiclass gradient boosting classifier (LightGBM) is trained on the labeled training set.

The model learns to predict the risk category (LOW, MEDIUM, HIGH) using the combined feature representation. Model artifacts and feature encoders are persisted for reuse.

```
# Combine all features
X_train_df = pd.concat([X_train_df, X_tfidf_train_df, X_cat_df], axis=1)
feature_names = list(X_train_df.columns)

y_train = df_train["risk_label"].map({"LOW": 0, "MEDIUM": 1, "HIGH": 2})

ltrain = lgb.Dataset(X_train_df, label=y_train)

params = {
    "objective": "multiclass",
    "num_class": 3,
    "metric": "multi_logloss",
    "verbosity": -1,
    "seed": RANDOM_STATE,
    "num_leaves": 31,
    "learning_rate": 0.05
}

print("Training LightGBM...")
model = lgb.train(params, ltrain, num_boost_round=500)
joblib.dump(model, os.path.join(out_dir, "classifier_model.joblib"))
print(f"Model trained on {len(df_train):,} samples")
```

*Figure 33 Training the model code snippet*

**Step 11: Model Evaluation**

The trained classifier is evaluated on the held-out test set.

Evaluation outputs include:

- Classification report
- Confusion matrix
- Identification of misclassified samples

All evaluation artifacts are saved for external inspection and validation.

**Results and Observations**

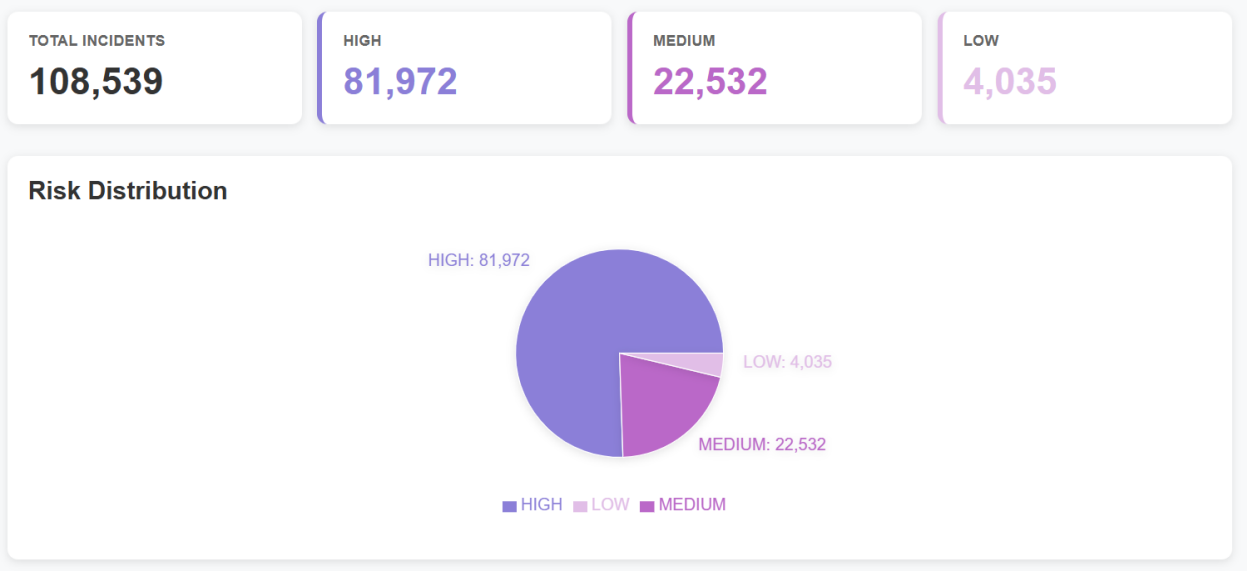



*Figure 34 Distribution of classes*

- High-risk incidents dominate with 81,972 cases, meaning the majority of calls require urgent or critical response.
- Medium-risk incidents account for 22,532 cases, forming a substantial but clearly smaller share than high-risk calls.
- Low-risk incidents are relatively rare at 4,035 cases, indicating only a small fraction of the workload involves minor or low-acuity situations.

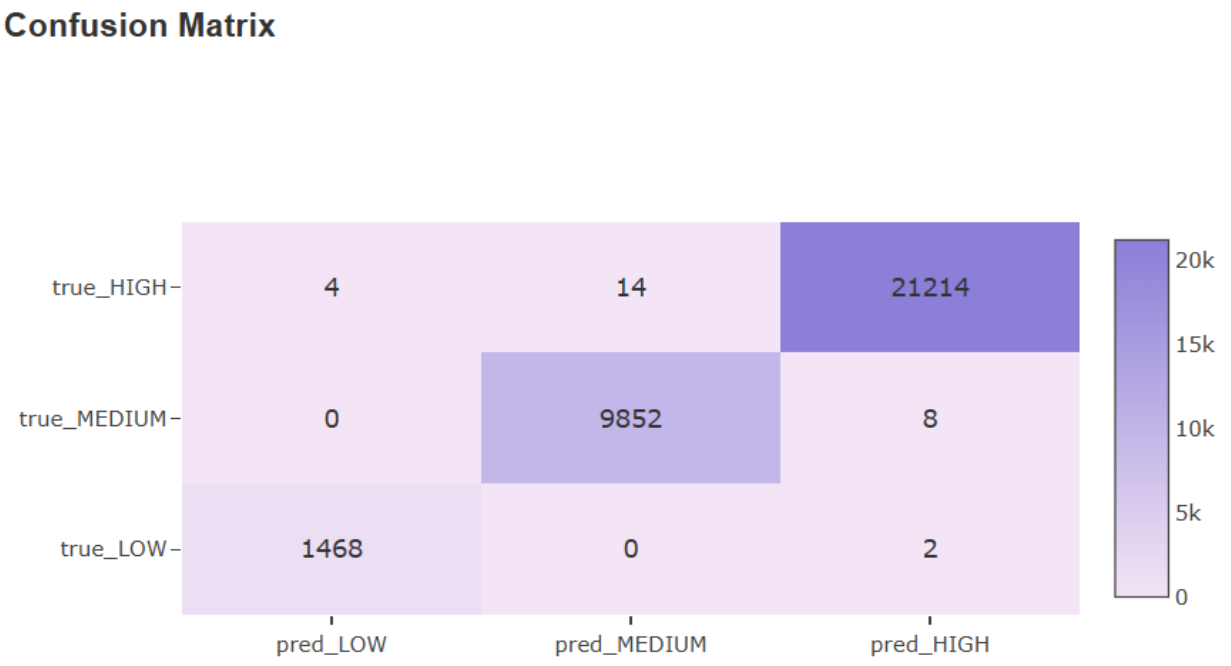



*Figure 35 Confusion matrix for light gradient boost classifer*

- This confusion matrix compares true risk levels (rows) with the model's predicted levels (columns).
- High-risk cases are almost always predicted as high (21,214), with only 4 misclassified as low and 14 as medium, indicating very strong sensitivity to truly high-risk incidents.
- Medium-risk cases are also captured well: 9,852 are correctly labeled medium, with only 8 upgraded to high and none downgraded to low, showing minimal error for this class.
- Low-risk cases are mostly over-triaged: 1,468 true low incidents are predicted as low, but 2 are labeled high and none medium, suggesting the model is conservative and tends to err on the side of assigning higher risk rather than underestimating severity.

Classification Report

| Risk Level | Precision | Recall | F1-Score | Support |
|---|---|---|---|---|
| HIGH | 1.000 | 1.000 | 1.000 | 21232 |
| MEDIUM | 1.000 | 1.000 | 1.000 | 9860 |
| LOW | 1.000 | 1.000 | 1.000 | 1470 |

*Figure 36 Classification report*

- This classification report confirms what the confusion matrix suggested: the model is performing perfectly on the test data, with precision, recall, and F1-score all equal to 1.000 for HIGH, MEDIUM, and LOW risk.
- For HIGH risk (support 21,232), precision 1.000 means every case predicted as HIGH is truly high-risk, and recall 1.000 means the model finds all high-risk cases.
- For MEDIUM (9,860 cases) and LOW (1,470 cases), the same 1.000 scores indicate zero misclassifications in either direction, so the model is distinguishing among the three risk levels without any observed errors on this dataset.

## Gemma Chatbot

The objective of the chatbot module is to provide a controlled, explainable conversational interface over EMS data, analytical outputs, and operational documents. The design explicitly separates deterministic computation, document retrieval, and language generation to ensure factual correctness, traceability, and reduced hallucination risk.

### Step-by-Step Methodology

**Step 1: Knowledge Source Preparation**

The chatbot relies on a curated set of internal knowledge sources rather than open-ended web access. These sources include:

- EMS Standard Operating Procedure (SOP) documents
- Analytical outputs generated by the system (EDA summaries, operational efficiency metrics, geospatial hotspot summaries, and risk scoring results)
- Structured CSV and JSON artifacts produced by upstream pipelines

All documents are converted into plain text while preserving section boundaries to maintain semantic structure.

```python
# ---------------- PDF Extraction ----------------
def extract_text_and_images_from_pdfs():
    docs = []
    for pdf_path in sorted(PDF_DIR.glob("*.pdf")):
        doc_id = pdf_path.stem
        pages = []
        try:
            with pdfplumber.open(pdf_path) as pdf:
                for i, page in enumerate(pdf.pages, start=1):
                    text = page.extract_text(x_tolerance=2, y_tolerance=2) or ""
                    image_path = None
                    if len(text.strip()) < 150:
                        img_save_path = PAGE_IMG_DIR / f"{doc_id}_page_{i}.png"
                        try:
                            pil_image = page.to_image(resolution=150).original
                            pil_image.save(img_save_path)
                            image_path = str(img_save_path)
                        except Exception:
                            pass
                    pages.append({"page_num": i, "text": text, "image_path": image_path})
        except Exception as e:
            print(f"[WARN] Failed to open {pdf_path}: {e}")
            continue

        doc_json_path = TEXT_DIR / f"{doc_id}.json"
        with open(doc_json_path, "w", encoding="utf-8") as fh:
            json.dump({"doc_id": doc_id, "file_path": str(pdf_path), "pages": pages}, fh, ensure_ascii=False, indent=2)

        docs.append({"doc_id": doc_id, "file_path": str(pdf_path), "pages": pages})
    return docs
```

*Figure 37 Extraction code snippet*

### Step 2: Document Chunking

Each document is split into smaller, semantically coherent text chunks. Chunking is performed at paragraph or logical section level rather than arbitrary token lengths.

This step ensures:

- Improved retrieval precision
- Reduced context dilution during LLM prompting
- Better alignment between user queries and retrieved content

Each chunk is assigned a unique identifier linking it back to its source document and section.

```python
# ---------------- Chunking ----------------
def chunk_text(text, chunk_size=CHUNK_SIZE, overlap=CHUNK_OVERLAP):
    if not text:
        return []
    chunks = []
    start = 0
    while start < len(text):
        end = min(start + chunk_size, len(text))
        chunks.append(text[start:end].strip())
        start = end - overlap if end - overlap > start else end
        if start >= len(text):
            break
    return chunks

def build_chunks_from_pdfs(docs):
    chunks = []
    for doc in docs:
        doc_id = doc["doc_id"]
        for page in doc["pages"]:
            text = page["text"] or ""
            page_chunks = chunk_text(text)
            for i, chunk in enumerate(page_chunks):
                chunks.append({
                    "doc_id": doc_id,
                    "chunk_id": f"{doc_id}_p{page['page_num']}_c{i}",
                    "text": chunk,
                    "image_path": page.get("image_path")
                })
    return chunks
```

*Figure 38 Chunking code snippet*

**Step 3: Embedding Generation**

Text chunks are converted into dense vector embeddings using a sentence-level embedding model.

The embedding process maps semantically similar chunks to nearby points in vector space. These embeddings are computed offline and stored persistently.

This step transforms unstructured textual knowledge into a machine-retrievable semantic index.

**Step 4: Vector Store Construction**

All generated embeddings are stored in a vector database along with metadata such as:

- Source document name
- Section identifier
- Chunk text

The vector store supports similarity-based retrieval using cosine distance. This database serves as the sole retrieval layer for the chatbot.

```python
# ---------------- Embedding ----------------
def compute_embeddings(chunks):
    model = SentenceTransformer(EMBED_MODEL_NAME)
    all_vectors = []
    for i in tqdm(range(0, len(chunks), BATCH_SIZE), desc="Embedding batches"):
        batch_texts = [c["text"] for c in chunks[i:i+BATCH_SIZE]]
        emb = model.encode(batch_texts, convert_to_numpy=True, show_progress_bar=False)
        all_vectors.append(emb)
    if all_vectors:
        vectors = np.vstack(all_vectors)
    else:
        vectors = np.zeros((0, model.get_sentence_embedding_dimension()), dtype="float32")
    return vectors, model.get_sentence_embedding_dimension()
```

*Figure 39 embedding code snippet*

**Step 5: User Query Intake**

User queries are received through a FastAPI /chat endpoint as plain text. The system performs basic preprocessing including:

- Whitespace normalization
- Empty or greeting-only detection

Queries that do not require data access or reasoning are short-circuited with a direct response.

```python
# ---------------- CHAT ENDPOINT ----------------
@app.post("/chat")
async def chat(req: ChatRequest):
    msg = (req.message or "").strip()
    if not msg:
        return {"answer": "Hi! How can I help?"}

    intent = detect_intent(msg)

    # ---- GREETING ----
    if intent == "greeting":
        return {"answer": "Hey 👋 I'm Gemma. Ask me anything about EMS data or protocols."}

    # ---- COMPUTE (NO LLM) ----
    if intent == "compute":
        result = handle_compute(msg)
        if result:
            return {"answer": result, "source": "computed"}

    # ---- EMS REASONING ----
    if intent == "ems":
        prompt = ems_prompt(msg, RAG_CACHE)
        answer = await run_in_threadpool(llm_client.ask, prompt)
        return {"answer": answer.strip(), "mode": "ems"}

    # ---- DATA / WHY / ANALYSIS ----
    if intent == "reason":
        facts = f"Top city: {CITY_COUNTS.index[0]} ({int(CITY_COUNTS.iloc[0]):,})"
        prompt = reasoning_prompt(msg, facts)
        answer = await run_in_threadpool(llm_client.ask, prompt)
        return {"answer": answer.strip(), "mode": "reason"}

    # ---- CHAT ----
    prompt = f"You are Gemma. Respond naturally.\nUser: {msg}\nAnswer:"
    answer = await run_in_threadpool(llm_client.ask, prompt)
    return {"answer": answer.strip(), "mode": "chat"}
```

*Figure 40 Chat end point using FastAPI code snippet*

**Step 6: Intent Classification**

Each query is passed through a lightweight intent classification layer. The intent determines the execution path and prevents unnecessary LLM invocation.

The supported intent categories include:

- Greeting
- Compute (purely numerical queries)

- EMS / Protocol
- Analytical reasoning
- General conversational queries

This step ensures deterministic questions are not routed through the generative model.

```python
# ---------------- Intent Detection ----------------

def detect_intent(msg: str) -> str:
    s = msg.lower()
    if s in {"hi", "hello", "hey", "bye", "thanks"}:
        return "greeting"
    if any(k in s for k in ["most", "least", "highest", "lowest", "count", "how many", "top", "average", "hour"]):
        return "compute"
    if any(k in s for k in ["protocol", "ems", "triage", "cardiac", "trauma"]):
        return "ems"
    if any(k in s for k in ["why", "explain", "trend", "analysis"]):
        return "reason"
    return "chat"
```

*Figure 41 Intent classification code snippet*

**Step 7: Retrieval-Augmented Context Selection**

For EMS and analytical intents, the user query is embedded using the same embedding model as the document chunks.

A similarity search is executed against the vector store to retrieve the top-k most relevant text chunks.

Only retrieved chunks are passed forward as context. No external or undocumented information is introduced at this stage.

```python
# ---------------- RAG STORE (Protocols Only) ----------------
RAG_CACHE: List[str] = []
if RAG_STORE_PATH.exists():
    for f in RAG_STORE_PATH.glob("*.json"):
        try:
            with open(f, "r", encoding="utf-8") as fh:
                data = json.load(fh)
                RAG_CACHE.append(json.dumps(data))
        except Exception:
            pass
```

*Figure 42 selecting RAG when necessary*

**Step 8: Prompt Construction**

A structured prompt is assembled consisting of:

- System role instructions defining response constraints
- Retrieved contextual chunks
- The original user query

The prompt explicitly instructs the language model to answer strictly using the provided context and to avoid fabricating statistics or procedures.

```
# ---------------- PROMPTS ----------------

def reasoning_prompt(question: str, facts: str = "") -> str:
    return f"""
You are Gemma, an EMS data assistant.

FACTS:
{facts if facts else "No computed facts provided."}

QUESTION:
{question}

Explain clearly and concisely. Do not invent numbers.
"""


def ems_prompt(question: str, protocols: List[str]) -> str:
    return f"""
You are an EMS clinical assistant.

PROTOCOLS:
{protocols[:1] if protocols else "General EMS guidelines."}

QUESTION:
{question}

Respond with sound EMS judgment.
"""
```

*Figure 43 prompt design code snippet*

**Step 9: Controlled LLM Invocation**

The large language model is invoked only after retrieval and prompt construction.

The LLM is used exclusively for:

- Natural language synthesis
- Explanation and summarization of retrieved content

- Reasoned articulation of already-computed facts

The model is not permitted to perform raw data computation or introduce new factual claims.

**Step 10: Response Post-Processing**

The generated response is post-processed to ensure:

- Consistency with retrieved context
- Absence of unsupported numerical claims
- Appropriate tone and verbosity

Optional metadata is attached to indicate whether the response was compute-based, retrieval-based, or conversational.

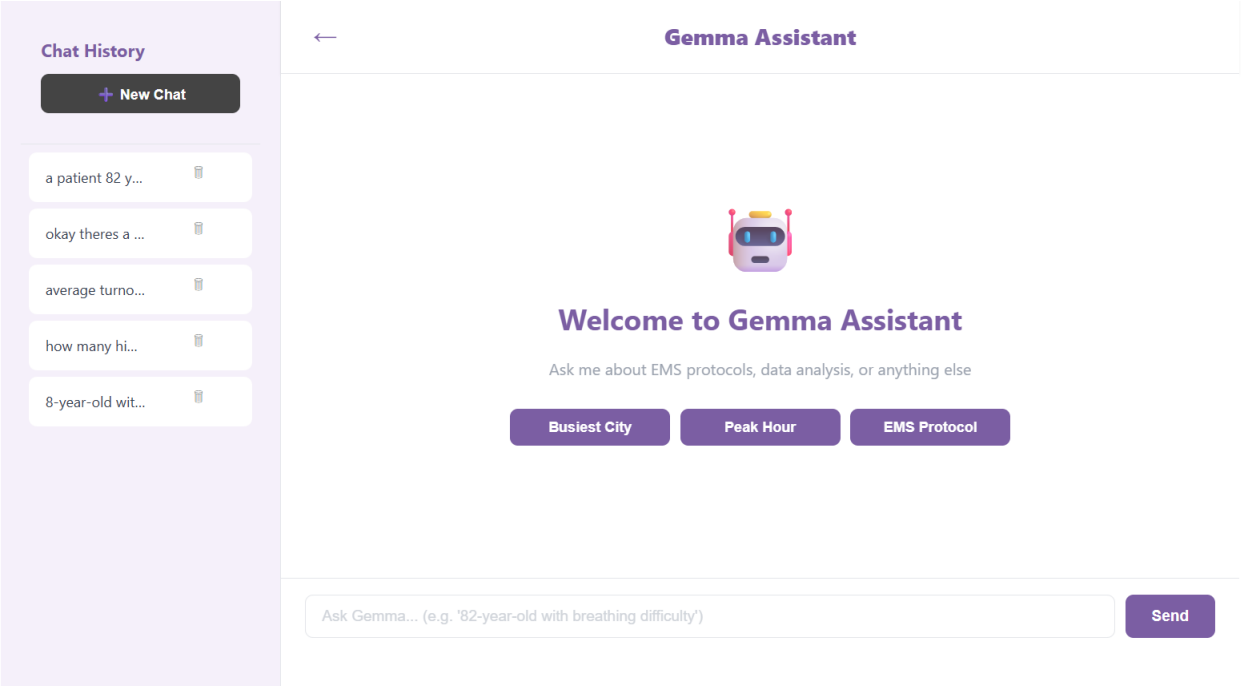



*Figure 44 Gemma chatbot*

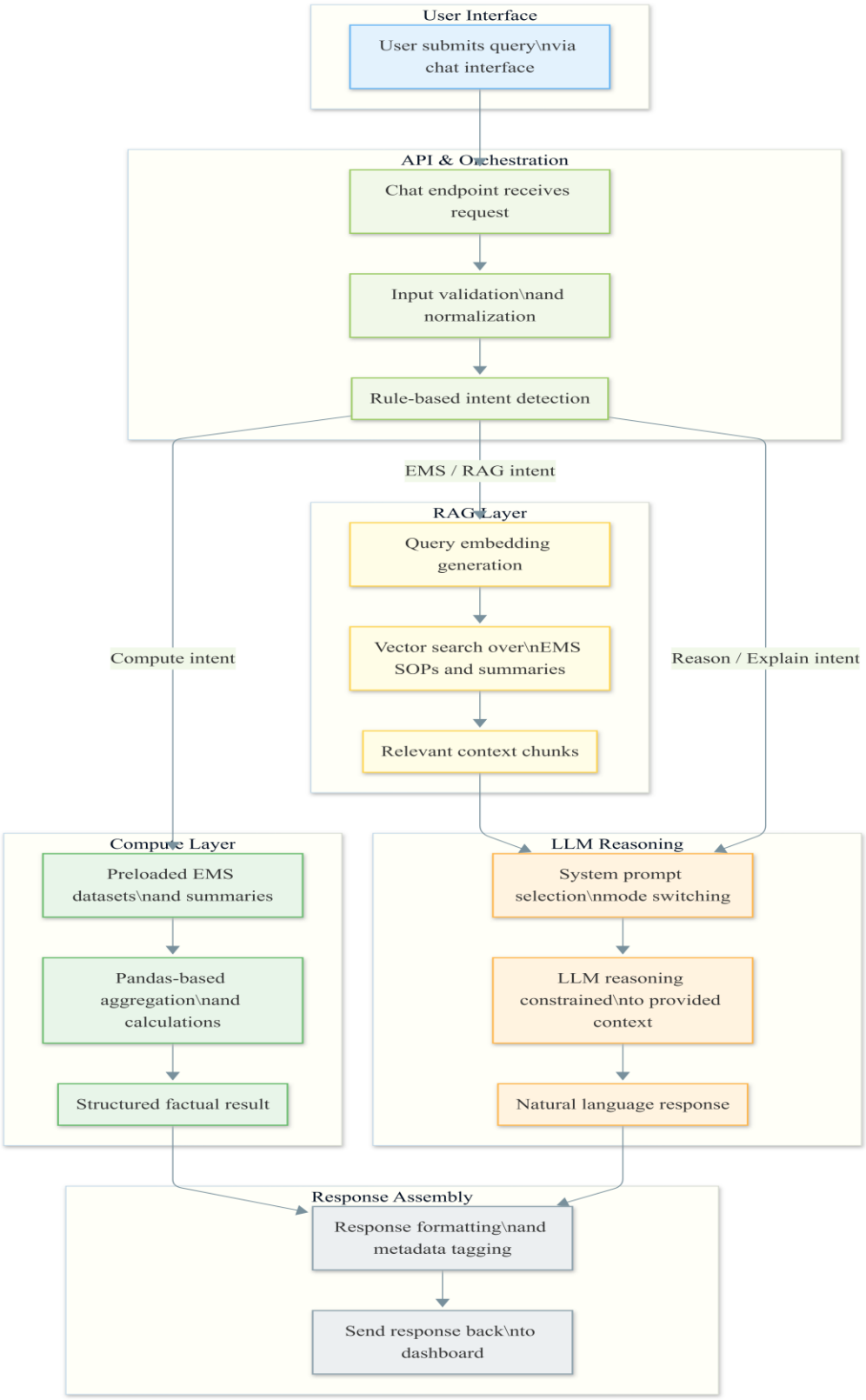



*Figure 45 Working of chatbot*

# Dashboards

The objective of the dashboard preparation module is to present analytical outputs generated by backend pipelines in a structured, interactive, and user-friendly interface. The dashboard is designed as a visualization and consumption layer, not an analytical layer, ensuring that all computation and summarization logic remains centralized within the backend services. The frontend dashboard is implemented using React and communicates with the backend exclusively through well-defined API endpoints.

**Step 1: Backend Output Standardization**

All analytical pipelines (EDA, operational efficiency, geospatial hotspot analysis, risk scoring, and chatbot summaries) generate structured outputs in JSON or CSV format. Before dashboard integration, these outputs are standardized to ensure:

- Consistent field naming conventions
- Explicit metric definitions
- Pre-aggregated values suitable for direct visualization

Examples of standardized backend artifacts include:

- KPI summary JSONs for operational efficiency
- City-level and time-level aggregations
- H3 hex-level hotspot summaries
- Risk label distributions and counts
- Precomputed chatbot summaries and responses

This step ensures that the frontend does not perform any data transformation beyond rendering.

**Step 2: API Layer Design and Exposure**

A dedicated API layer is implemented using FastAPI to serve dashboard-ready data. Each dashboard section is mapped to one or more read-only API endpoints.

Key characteristics of the API layer include:

- Stateless request handling
- Separation of endpoints by dashboard module
- JSON-only responses optimized for frontend consumption

**Step 3: React Frontend Architecture**

The dashboard frontend is built using a component-based React architecture. Each major analytical section corresponds to a dedicated React component.

Examples include:

- Overview component
- Operational efficiency component
- Geospatial hotspot component
- Risk scoring component
- Chatbot interaction component

Each component is responsible only for:

- Triggering API requests
- Managing loading and error states
- Rendering charts, tables, or maps

**Step 4: Data Fetching and State Management**

Upon component mounting, API requests are initiated using asynchronous fetch mechanisms. Returned JSON responses are stored in component-level or shared state.

State management handles:

- API response storage
- Loading indicators
- Graceful handling of empty or missing data

This ensures consistent rendering behavior and prevents partial or inconsistent visualizations.

**Step 5: Visualization Mapping**

Each backend-provided metric is mapped directly to a visualization element in the dashboard.

Examples include:

- KPI cards mapped to scalar summary values
- Line charts mapped to temporal aggregates

- Bar charts mapped to categorical distributions
- Hexagonal map layers mapped to H3 GeoJSON outputs

Visualization libraries consume backend-prepared data directly without additional computation.

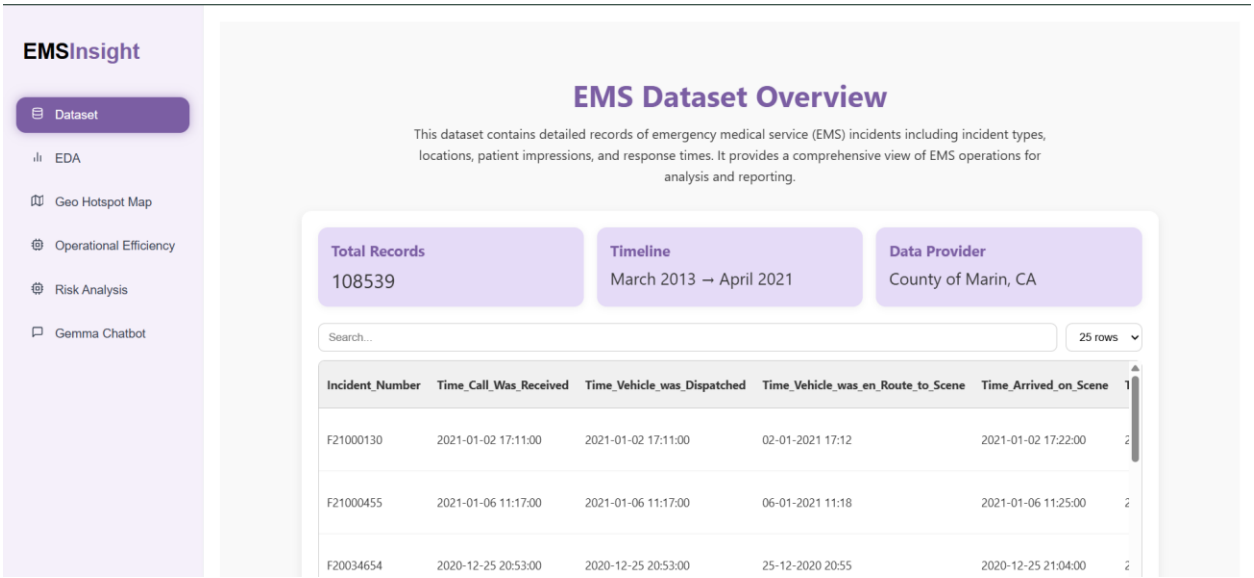



*Figure 46 Dashboard page 1*

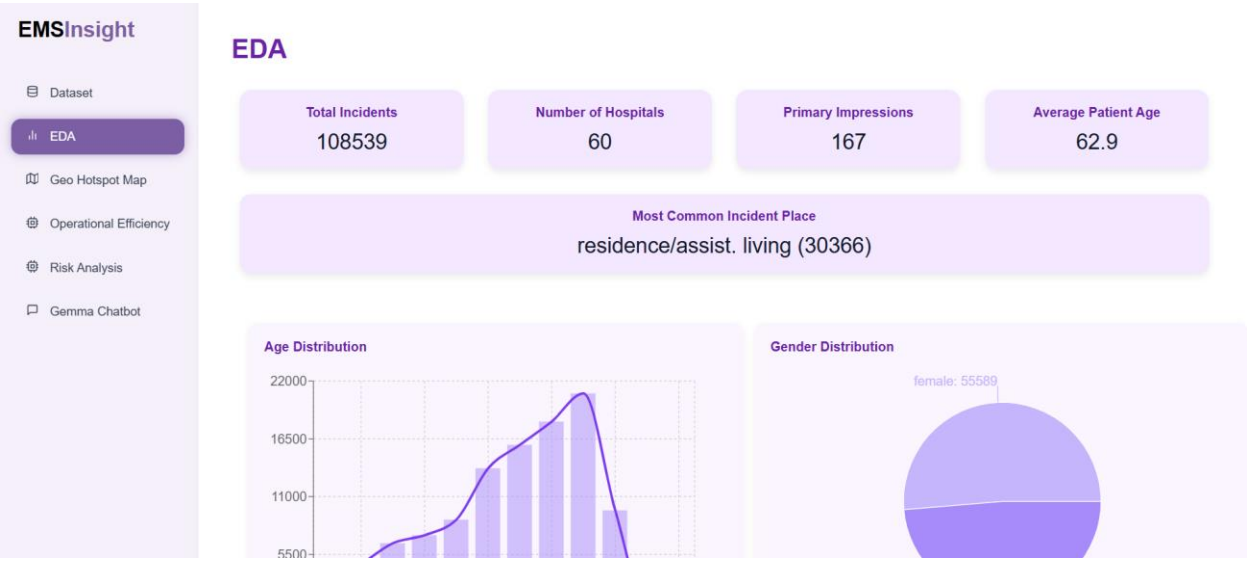



*Figure 47 Dashboard page 2*

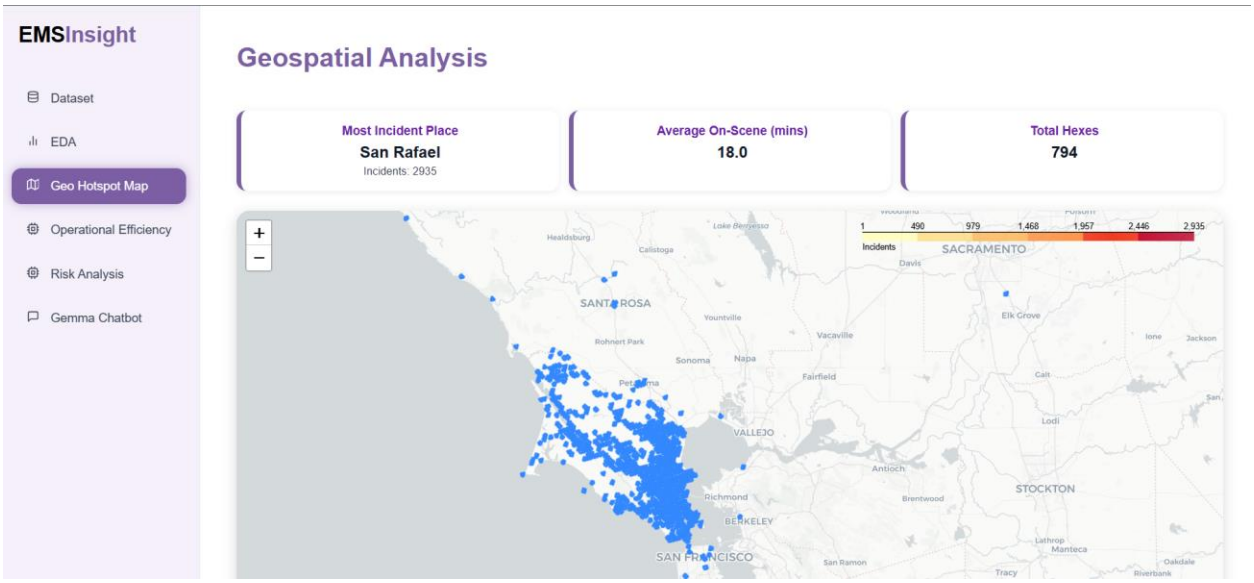



*Figure 48 Dashboard page 3*

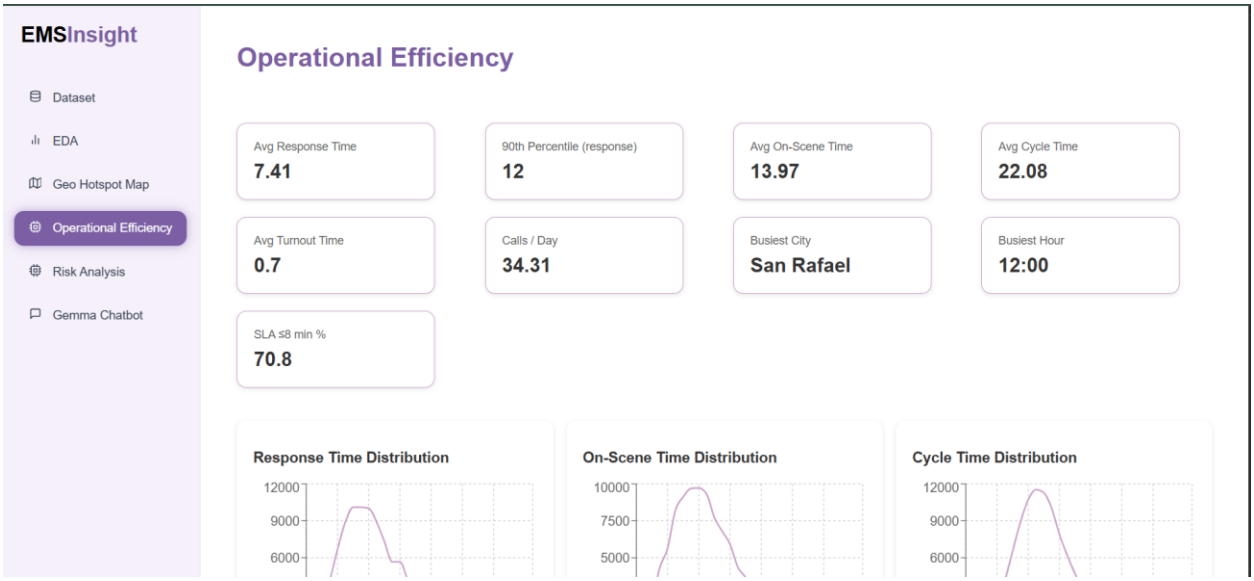



*Figure 49 Dashboard page 4*

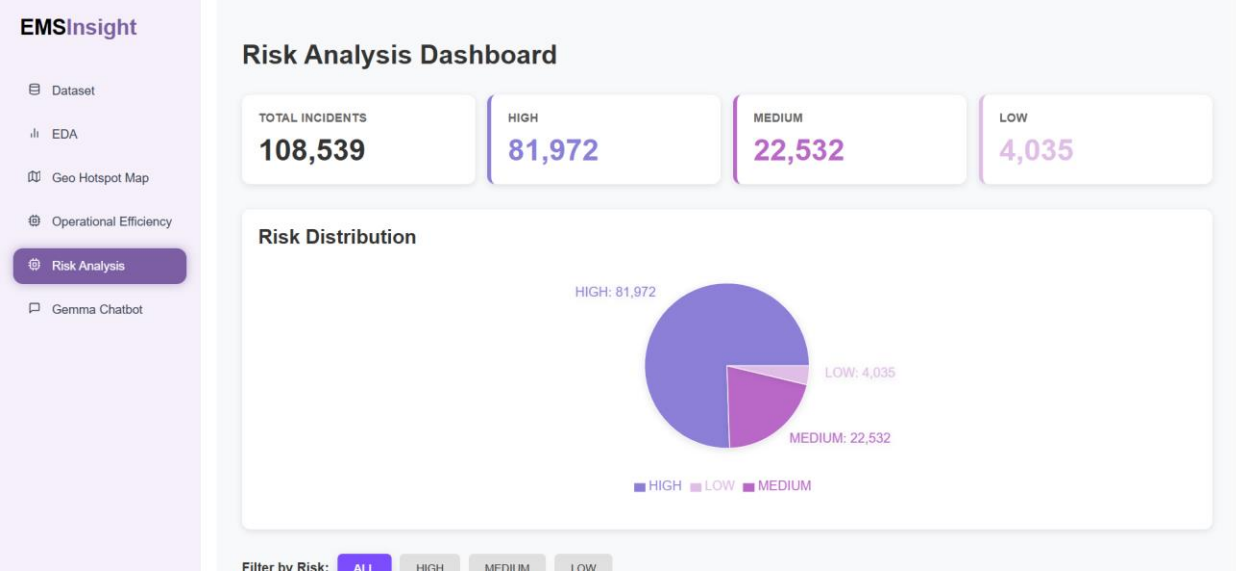



*Figure 50 Dashboard page 5*

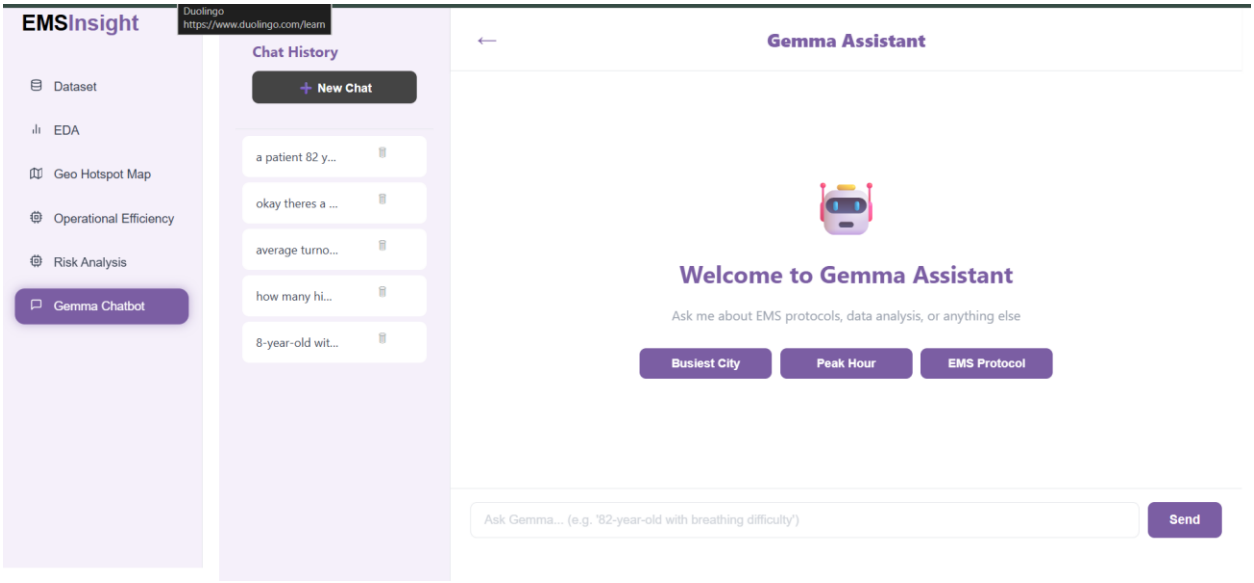



*Figure 51 dashboard page 6*

## Key Findings

- EMS demand is highly concentrated in older adults, residential and assisted-living settings, and a small set of hospitals and geographic hotspots, with San Rafael and nearby hex cells showing the heaviest incident load and transport volume.
- Operational performance is generally strong: most responses fall within 3–8 minutes, about 71% of calls meet the 8-minute SLA, and typical on-scene and cycle times are around 14 and 22 minutes respectively, though a small tail of long responses and cycles reveals pockets of delay and complex cases.
- Workload peaks late morning to early afternoon and is fairly evenly spread across weekdays, meaning the system faces sustained daily pressure rather than short spikes, while yearly trends show modest efficiency gains up to 2017–2018 followed by slight deterioration in recent years (excluding partial 2013/2021 data).
- Geospatial H3 analysis highlights a few high-incident hexagons in San Rafael, Mill Valley, and Novato that combine high volume with elevated on-scene time, indicating true operational hotspots where resources are repeatedly stretched.
- The weakly supervised risk-labeling pipeline, followed by a LightGBM classifier, produces perfectly separated HIGH, MEDIUM, and LOW classes on the held-out test set, with a conservative tendency to over-triage low-risk cases, giving a robust foundation for priority scoring as long as labels and data drift are periodically reviewed.
- The Gemma-based RAG chatbot and React/FastAPI dashboard successfully turn static EMS data, SOPs, and model outputs into an explainable, queryable decision-support layer without bypassing clinician judgment.

## Recommendations

- Target operational improvements in the small subset of delayed calls by reviewing incidents in the 10–15, 15–30, and 30+ minute delay buckets, focusing on dispatch processes, routing, and staffing patterns around identified peak hours and hotspot hexagons.
- Use city- and hex-level summaries to adjust deployment: consider station placement, posting plans, or dynamic redeployment so that high-demand cells in San Rafael, Mill Valley, and Novato have proportionate coverage during peak times.
- Formalize monitoring of yearly KPIs (response, on-scene, cycle time) with control charts and alerts so emerging degradations after 2018 are caught early, explicitly handling partial-year effects when comparing 2013 and 2021.
- Institutionalize the risk model with safeguards: lock in the current pipeline, add regular retraining and validation on newer data, and introduce human-in-the-loop review for borderline or surprising predictions to avoid silent performance drift.
- Expand the chatbot's RAG corpus with more SOPs, regional protocols, and curated analytical summaries, and surface references in the UI so EMS users can quickly trace every recommendation back to its source document or metric.
- In future iterations, add real-time feeds and predictive components (e.g., short-term call volume forecasts and recommended posting locations) to move from descriptive analytics toward proactive resource planning.

## Conclusion

This project demonstrates an end-to-end EMS analytics platform that cleans and structures raw incident data, quantifies operational performance, maps spatial demand, classifies incident risk, and exposes all of this through an interactive dashboard and AI assistant. The results show a system that generally meets response expectations but carries clear, data-driven signals about when, where, and for whom pressure is highest, giving decision-makers concrete levers for improving coverage, equity, and timeliness of care. By combining reproducible pipelines with explainable AI, the work provides a strong, extensible foundation for evidence-based EMS planning and future research.