

# **D1.4 – Functional design of the prototype**

Arnau Berenguer (EURECAT), Carles Riera (ESADE) & Julià Vicens (EURECAT)

Project **PLEC2021-007850** funded by:



Project ref. no.	PLEC2021-007850
Project title	REMISS
Project duration	1 <sup>st</sup> September 2021 – 31 <sup>st</sup> August 2024 (36 months)
Related Activity/Task	T1.4
Document due date	30/09/2021 (M30)
Actual delivery date	31/08/2024 (M36)
Deliverable leader	EURECAT
Document status	Final



### **Statement of originality**

This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both.

### **How to quote this document**

Berenguer, A., Riera, C. & Vicens J. (2024) Functional design of the prototype.



This deliverable is licensed under a CC BY-NC-SA 4.0

## Revision History

Version	Date	Document History	Contributors
1.0	01.02.24		Julià Vicens (EURECAT)
5.0	26.07.24		Arnau Berenguer (EURECAT)
19.0	29.08.24		Carles Riera (ESADE)
	30.08.24		Julià Vicens (EURECAT)

## Executive Summary

This report delves into the creation and integration of platform components specifically designed to generate actionable insights for case studies focusing on elections and immigration. Our primary objective is to define the functionalities of a prototype capable of producing these insights through meticulously crafted data processes and API interactions, all supported by an intuitive user interface.

**Data Ingestion and Transformation.** We begin by detailing the data ingestion module, which sets up robust processes for collecting and processing relevant data on elections and immigration. This module ensures that raw data is meticulously transformed into usable formats, ready for in-depth analysis and action generation. The accuracy and efficiency of this transformation are crucial for the prototype's overall effectiveness.

**APIs for Model Invocation.** Next, we outline the creation of APIs designed to invoke various models developed throughout the project. These APIs ensure seamless interaction between the data processing components and the analytical methods, enabling the platform to function smoothly and efficiently. This seamless interaction is vital for delivering timely and accurate insights.

**Backend and Frontend Design.** We then focus on the backend and frontend design. The backend systems are developed to support data storage, processing, and API management, ensuring that the platform's infrastructure is robust and reliable. On the frontend, we design a user-friendly interface that emphasizes clear and effective visualizations of propagation models and trust/credibility scores. This interface is crucial for users to easily interpret and act upon the generated insights.

**Development and Integration.** Finally, we describe the development and deployment of these components within a standardized DevOps framework, adhering to Continuous Integration and Continuous Deployment (CI/CD) principles. This approach ensures that the platform remains consistent, efficient, and capable of continuous improvement and updates. The integration under a DevOps framework guarantees the platform's reliability, scalability, and adaptability, aligning with the best practices in modern software development.

In summary, this report outlines the steps taken to create a comprehensive platform that generates actionable insights for case studies in elections and immigration. By meticulously defining and developing the necessary components, and integrating them under a standardized DevOps framework, we aim to deliver a reliable and scalable solution that meets the highest standards of modern software development. This platform not only provides valuable insights but also ensures that these insights are easily accessible and actionable for end-users.

# Table of Contents

<b>REVISION HISTORY .....</b>	<b>5</b>
<b>EXECUTIVE SUMMARY .....</b>	<b>6</b>
<b>TABLE OF CONTENTS .....</b>	<b>7</b>
<b>LIST OF FIGURES .....</b>	<b>8</b>
<b>LIST OF TABLES .....</b>	<b>9</b>
<b>1. INTRODUCTION .....</b>	<b>10</b>
<b>2. DATA INGESTION AND TRANSFORMATION .....</b>	<b>10</b>
<b>3. APIS FOR MODEL INVOCATION .....</b>	<b>10</b>
<b>4. BACKEND AND FRONTEND DESIGN .....</b>	<b>12</b>
4.1 FRAMEWORKS USED .....	13
4.2 BACKEND .....	13
4.2.1 <i>MongoPlotFactory</i> .....	14
4.2.2 <i>TimeSeriesFactory</i> .....	14
4.2.3 <i>TweetTableFactory</i> .....	14
4.2.4 <i>TextualFactory</i> .....	14
4.2.5 <i>MultimodalPlotFactory</i> .....	15
4.2.6 <i>ProfilingPlotFactory</i> .....	15
4.2.7 <i>PropagationPlotFactory</i> .....	15
4.3 FRONTEND .....	15
4.3.1 <i>RemissComponent</i> .....	15
4.3.2 <i>RemissState</i> .....	15
4.3.3 <i>RemissDashboard</i> .....	16
4.3.4 <i>ControlPanelComponent</i> .....	16
4.3.5 <i>GeneralPlotsComponent</i> .....	16
4.3.6 <i>FilterablePlotsComponent</i> .....	17
<b>5. DEVELOPMENT AND INTEGRATION .....</b>	<b>17</b>
<b>6. CONCLUSIONS .....</b>	<b>19</b>
<b>TERMINOLOGY AND ACRONYMS .....</b>	<b>20</b>

## List of Figures

Figure 1 – Block diagram with the functional design of the prototype .....	19
--	----



## List of Tables

# 1. Introduction

This report explores the design and integration of platform components aimed at generating actionable insights for case studies centred on elections and immigration. The primary goal is to define the functionalities of a prototype capable of transforming raw data into meaningful insights through well-defined data processes, model invocation via APIs, and a user-friendly interface. This platform's effectiveness hinges on the interaction between its backend and frontend systems, ensuring that users can easily access and act upon the insights produced. By adhering to a standardized DevOps framework, the platform is designed to remain scalable, reliable, and efficient, aligned with the current demands of software development.

# 2. Data Ingestion and Transformation

The data ingestion process begins by importing JSONL dumps from Twarc, where each line may contain multiple tweets. The system's first step is to flatten these into a one-tweet-per-line format. Following this, the data undergoes preprocessing, which includes standardizing the date format and integrating metadata from Verificat to identify fake news spreaders.

Once preprocessing is complete, the system generates a series of output files with the same base name as the original, but with different extensions based on their content:

1. **\*.preprocessed.jsonl:** This file contains the raw data with each tweet on a separate line, augmented with Verificat's metadata.
2. **\*.mongoimport.jsonl:** This file is identical to \*.preprocessed.jsonl but it has the date fields formatted as specified by mongoimport, ready to be imported into a database.
3. **\*.media.jsonl:** This file includes only tweets that contain media (images or videos), specifically for any partners requiring this subset.
4. **\*.suspects\_or\_politicians.jsonl:** This file includes only tweets from individuals identified as "usual suspects" or politicians, according to Verificat's metadata.

# 3. APIs for Model Invocation

There are four distinct APIs for model invocation within this project. All of them are developed in Python to facilitate the development and execution of the models, leveraging the Flask web framework. Additionally, a central API serves as the sole exposed endpoint, forwarding individual requests to all three APIs. Each API has been "dockerized" to ensure a stable development and deployment environment, thereby enabling seamless integration across all components.

The first API is the developed in D5.3 by UVEG to obtain the results of the textual model, which exposes a single endpoint for processing the provided dataset and storing it in a preconfigured MongoDB database. The dataset must be stored in a specific location on the filesystem, and the API will search for the corresponding filename provided via query parameters. The endpoint details are as follows:

- **HTTP Method:** POST
- **Endpoint:** /process
- **Parameters:**

- file\_name: Name of the CSV file containing the text data to be processed
- db\_name: Name of the database where the collection containing the processing results will be stored
- col\_name: Name of the collection where the processing results will be stored
- text\_label: Label identifying the column in the CSV file that contains the texts to be analyzed
- **Example Request:**
  - curl -X POST [http://127.0.0.1:5006/process?file\\_name=prueba.csv&db\\_name=test&col\\_name=textual&text\\_label=text](http://127.0.0.1:5006/process?file_name=prueba.csv&db_name=test&col_name=textual&text_label=text)

The second API is the developed by CVC providing the results of the profiling analysis. It follows the same principles as the previously mentioned API. This API exposes a single endpoint that receives a JSONL file, processes it, and stores the profiling results in a specified MongoDB database. The endpoint details are as follows:

- **HTTP Method:** POST
- **Endpoint:** /upload
- **Parameters:**
  - db\_name: Name of the database where the collection with the processed results will be stored
- **Attachments:**
  - file: File to be uploaded for subsequent processing by the model. The file must be in JSONL format.
- **Example Request:**
  - curl -X POST [http://127.0.0.1:5000/upload?db\\_name=cambriils](http://127.0.0.1:5000/upload?db_name=cambriils) -F 'file=@./cambriils.jsonl'

The third API is the one developed by ESADE producing the results of the propagation Model. Like the other APIs, it exposes a single endpoint supporting one HTTP method. This endpoint receives a JSONL file containing tweets and related information, processes it using the model, and stores the propagation metrics in a specified MongoDB database. The endpoint details are as follows:

- **HTTP Method:** POST
- **Endpoint:** /process\_dataset
- **Parameters:**
  - db\_name: Name of the database where the results of the dataset processing will be stored
- **Attachments:**
  - file: File to be uploaded for subsequent processing by the model. The file must be in JSONL format.
- **Example Request:**
  - curl -X POST [http://127.0.0.1:5000/process\\_dataset?db\\_name=cambriils](http://127.0.0.1:5000/process_dataset?db_name=cambriils) -F 'file=@./cambriils.jsonl'

On the other hand, there is the fourth API with the multimodal model developed by the CVC. This API is not like the other ones. This is more than a simple API, it is also a graphical web application built using the framework Gradio, which allows to build interactive applications for machine learning models and notebooks. This app provides endpoints for remote requests but it is not its intended use. This app is meant to be used through its graphical user interface with human intervention. The user must provide a tweet's text and attached image and the

application will return the results of the multimodal analysis, for posterior manual storing into the MongoDB database.

Lastly, there is the main API, also referred to as the Integration API. Its primary purpose is to provide a single public endpoint that receives the dataset, converts it to the required formats, and forwards requests to the other APIs. This reduces the system's complexity by consolidating three separate requests into one. The Integration API follows the same design philosophy, exposing a single endpoint that only supports the POST HTTP method. The endpoint details are as follows:

- **HTTP Method:** POST
- **Endpoint:** /processdata
- **Parameters:**
  - db\_name: Name of the database where the results from the textual, profiling, and propagation models will be stored
- **Attachments:**
  - file: File to be uploaded for processing by all methods. This file will be forwarded to the respective APIs and must be in JSONL format.
- **Example Request:**
  - `curl -X POST http://localhost:5000/processdata?db_name=cambrils -F 'file=@./cambrils.jsonl'`

## 4. Backend and Frontend Design

The system is designed with a robust and flexible architecture centered around a set of factory classes. These factories are responsible for managing the flow of data from the database, cleaning and formatting it for downstream processes, and generating the plots and visual outputs used for analysis and reporting. By handling the core data operations in these classes, the system ensures a clear separation of concerns, making the overall architecture more manageable and scalable.

Complementing the factory classes is a collection of component classes that share a common interface, allowing them to be seamlessly integrated into different parts of the interface. These components are designed to be modular and interchangeable, giving the flexibility to mix and match functionality based on specific needs. This composability is a key strength of the system, enabling different configurations of components to be used in diverse scenarios without requiring significant rewrites of the underlying code. The uniform interface across these components ensures consistent behavior, making it easier to extend the system as new requirements emerge.

The system also includes two specialized modules related to propagation analysis. The first module contains the logic for handling propagation-related metrics, dealing with how content spreads through networks. The second module implements the actual propagation machine learning model, which is able to predict if a tweet will be propagated based in its previous history.

To ensure the reliability and correctness of the system, extensive unit testing is conducted. Each module is accompanied by its own dedicated test file, which contains a suite of unit tests designed to validate the functionality of the code. This test-driven approach helps catch bugs early in development and ensures that changes or additions to the codebase do not

introduce regressions. By maintaining a high level of test coverage, the system can be confidently expanded and refactored while preserving stability.

Additionally, the system includes a set of management scripts that ease various operational tasks. The primary entry point is `app.py`, which handles the overall execution of the system server. The `prepopulate.py` script is responsible for computing the propagation metrics and storing them in the database, ensuring that relevant data is readily available for analysis. Lastly, the `preprocess.py` script automates the conversion of raw Twarc JSONL dumps into a format suitable for further processing or direct import into the database.

## 4.1 Frameworks used

The system is built using a combination of frameworks that have been carefully selected for their seamless integration and efficiency in handling both data processing and visualization tasks. At the forefront of the user interface is Dash, a Python-based web framework that enables the creation of highly interactive and customizable dashboards. Dash was chosen for its ability to rapidly develop data-driven web applications, which are essential for visualizing large-scale data such as social media activity. It integrates smoothly with Plotly, a powerful plotting library, enabling dynamic and responsive data visualizations to be embedded directly into the dashboard. The close relationship between Dash and Plotly (both developed by the same team) ensures tight integration and a consistent development experience, minimizing the friction typically associated with connecting different tools. Plotly is used extensively for generating a wide range of visualizations in the system. From line charts and bar plots to more complex visual representations like 3D network graphs. Its ability to render high-quality, interactive plots in web applications enhances the user experience, allowing users to explore the data in depth without needing specialized tools. This combination of Dash for the dashboard interface and Plotly for the visualizations provides a smooth workflow for creating interactive, visually appealing reports that update in real-time based on underlying data changes.

On the data storage side, MongoDB was selected as the primary database. MongoDB's document-based structure is well-suited for handling unstructured or semi-structured data like tweets, which often come in the form of JSON documents with varying schemas. The system interfaces with MongoDB using PyMongo, a Python library that simplifies the process of executing queries, retrieving data, and performing updates. For situations where tabular data representation is needed—particularly when dealing with larger datasets that require efficient querying and data manipulation—PyMongoArrow is employed. PyMongoArrow allows for the conversion of MongoDB query results into tabular formats like Apache Arrow or Pandas, enabling smooth integration with modules that require tabular data for analysis.

The system includes a specialized propagation module focused on analyzing the spread of tweets through user networks. To model and analyze these networks, the system uses iGraph, a robust library for graph-based analysis. iGraph provides powerful tools for working with large graphs, including efficient algorithms for calculating closeness, spanning trees, and other key network metrics. By leveraging iGraph, the propagation module is able to efficiently handle the computational complexity involved in simulating and analyzing large-scale networks.

## 4.2 Backend

### 4.2.1 MongoPlotFactory

The MongoPlotFactory class is the abstract base class (ABC) designed to interface with a MongoDB database to retrieve and preprocess data for visualization purposes, from which all plot factories inherit from. It connects to a MongoDB instance using a specified host and port (defaulting to localhost and port 27017) and manages datasets, date ranges, and hashtag frequencies. The class provides utility methods for validating the existence of datasets and collections, retrieving date ranges from the dataset, and calculating hashtag frequencies either globally or filtered by author ID and date range. It also includes utilities for fetching user IDs and usernames from the dataset by querying for tweets. By handling these database operations centrally, the `MongoPlotFactory` provides a standardized interface for data retrieval, which can be extended by subclasses to generate plots or visualizations from the data.

### 4.2.2 TimeSeriesFactory

The TimeSeriesFactory class extends MongoPlotFactory to generate time series plots of tweet and user activity from a MongoDB dataset. It integrates a Histogram class to compute or load histograms of tweet or user data based on filters like hashtags, date range, and time units (e.g., day). The class offers two key methods: `plot\_tweet\_series` and `plot\_user\_series`, which compute and plot the number of tweets or users over time. If specific filters are provided, the class dynamically computes the data; otherwise, it loads precomputed histograms.

### 4.2.3 TweetTableFactory

The TweetTableFactory class, extending MongoPlotFactory, is responsible for generating tables of tweet data from MongoDB, enriched with various metrics and metadata. It retrieves tweet information from multiple collections, including raw tweet data, multimodal content, profiling information, textual analysis, and network metrics. The class builds MongoDB aggregation pipelines to filter tweets based on hashtags, date range, and other criteria. After gathering the relevant data, it merges the results into a Pandas DataFrame, applying filters and transformations, such as handling multimodal flags, profiling status, and calculating cascade sizes. The final table includes user information, retweets, party affiliation, suspicion level, and other attributes, sorted by retweet count for easy identification of top-performing tweets. The class supports flexible querying and is designed to provide a comprehensive, enriched view of tweet data for further analysis.

### 4.2.4 TextualFactory

The TextualFactory class, an extension of MongoPlotFactory, focuses on analyzing and visualizing emotion-related data extracted from tweet texts stored in MongoDB. It provides functionalities to plot the average values of various emotions such as fear, anger, disgust, and others across a dataset, transforming raw emotion metrics into a bar chart. The class also supports plotting emotional trends over time by computing the average emotion scores per hour of the day. This time-based analysis allows for the detection of emotional patterns,

making it useful for understanding how sentiment fluctuates throughout the day or across specific time frames.

### 4.2.5 MultimodalPlotFactory

The MultimodalPlotFactory class, derived from MongoPlotFactory, focuses on handling and visualizing multimodal automatic fact-checking results. The class provides several plotting methods to display related images, such as claim images, evidence visuals, and text-based evidence graphs. It also ensures the integrity of datasets by verifying the presence of multimodal data in a MongoDB collection and raises exceptions if any required files are missing.

### 4.2.6 ProfilingPlotFactory

The ProfilingPlotFactory class is designed for visualizing user profiles by leveraging MongoDB for data retrieval and CSV files for feature selection. It supports multiple languages (Catalan, Spanish, English) and offers methods to generate various plots. These plots include bar charts, radar charts, and donut charts that compare user characteristics and behaviour (such as emotional traits, gender distribution, and tweet patterns) against different groups like fake news spreaders, fact checkers, and control users.

### 4.2.7 PropagationPlotFactory

The PropagationPlotFactory class is designed for analyzing and visualizing propagation networks, particularly focused on Twitter data. It extends the MongoPlotFactory class and integrates various network analysis components such as Egonet, NetworkMetrics, and DiffusionMetrics. This class provides methods to generate and plot ego networks, hidden networks, propagation trees, and various time-based metrics like size, depth, and structural virality of information cascades. It also handles user metadata, computes network layouts, and offers data persistence capabilities. The class is designed to work with libraries like igraph for network calculations and Plotly for creating interactive 3D visualizations of network structures and time series data.

## 4.3 Frontend

### 4.3.1 RemissComponent

The RemissComponent class is an abstract base class designed to serve as a foundation for creating modular components in Dash. It initializes each component with a unique name, either provided or randomly generated. The class defines two key methods: `layout`, which is meant to be implemented by subclasses to define the component's visual structure, and `callbacks`, which can be overridden to add interactive functionality to the component.

### 4.3.2 RemissState



The `RemissState` class is a component designed to manage the state of a dashboard application. It extends the `RemissComponent` class and utilizes Dash's `dcc.Store` components to maintain session-based storage for key pieces of information. The class keeps track of the current dataset, hashtags, date range, user, and tweet, each stored in a separate `dcc.Store` object. The `layout` method arranges these store components in a `div` container, so they are included in the application. The `callbacks` method sets up a mechanism to clear all other stores when the dataset changes. This class serves as a central state management system, allowing other components to access and modify shared data, thus facilitating coordinated updates and interactions across the entire dashboard application.

### 4.3.3 RemissDashboard

The `RemissDashboard` class is a component for the dashboard application, built using Dash. It extends the `RemissComponent` class and serves as the main container for various sub-components of the dashboard. The class integrates multiple specialized components for different functionalities such as general plots, control panels, ego networks, tweet tables, and filterable plots. It also includes a dataset selector dropdown and an upload component for adding new datasets to the system. The `layout` method defines the overall structure of the dashboard, organizing these components into a responsive grid layout using Bootstrap components. The class manages the state of the dashboard through a `RemissState` object and sets up callbacks for interactivity. This class essentially acts as the central orchestrator of the rest of the components of the system.

### 4.3.4 ControlPanelComponent

The `ControlPanelComponent` class is a component designed to manage and display the current filters for the dashboard. It extends the `RemissComponent` class and integrates several sub-components:

1. A date picker for selecting date ranges
2. A wordcloud component for visualizing hashtag frequencies
3. A filter display component for showing current selections

The class manages the state of these controls and provides methods to update the wordcloud based on selected dataset, author, and date range. It also handles the storage and retrieval of current selections (dataset, hashtags, dates, etc.). The `layout` method arranges these elements in a vertical stack using Bootstrap cards. The `callbacks` method sets up various interactive features, such as updating the date range when the dataset changes, updating hashtag storage when a wordcloud element is clicked, and refreshing the wordcloud based on current selections. This component serves as the main control hub for the dashboard, allowing users to filter and interact with the displayed data across other components.

### 4.3.5 GeneralPlotsComponent

The `GeneralPlotsComponent` class is a composite component that aggregates several plot components to display static information about a dataset in a dashboard application. It extends the `RemissComponent` class and incorporates four sub-components: a cascade ccdf plot, a cascade count plot, an average emotion barchart, and emotion per hour plot. The



layout method arranges these sub-components in a responsive grid layout using Bootstrap's row and column system. The callbacks method sets up interactivity for each sub-component. This class serves as a container for general, dataset-wide visualizations, providing a comprehensive overview of the data's characteristics in terms of information spread and emotional content across time.

#### 4.3.6 FilterablePlotsComponent

The FilterablePlotsComponent class is a composite component that aggregates several plot components which can be filtered based on user input, date ranges, and other criteria. It extends the RemissComponent class and incorporates four main sub-components: a time series plot showing frequencies for tweets and users over time for the dataset and time span selected, a set of plots displaying profiling data for the selected twitter user, a set of plots showing results from the automatic multimodal fake-news detector, and a selection of propagation related charts. The layout method arranges these sub-components in a vertical stack using Bootstrap's row and column system. The callbacks method sets up interactivity for each sub-component. This class serves as a container for all the plots that depends on filters selected by the user.

## 5. Development and Integration

### DevOps Framework

The DevOps framework implemented in this project encompasses the entire software lifecycle, from initial development to deployment and maintenance. Key aspects include:

- **Collaboration and Communication:** Enhanced collaboration between developers, operations and between partners to ensure alignment and rapid feedback.
- **Monitoring and Logging:** Continuous monitoring of system performance and logging of events to detect issues early and ensure system health.

### Continuous Integration

Continuous Integration (CI) involves the frequent integration of code changes into a shared repository, ensuring that new code is tested and validated. The CI process in this project includes:

- **Source Code Management:** Use of version control systems (Git with GitHub) to manage and track code changes across multiple branches, repositories and contributors.
- **Testing:** Implementation of a robust suite of unit and integration tests to verify code quality and functionality

### Continuous Deployment

Continuous Deployment (CD) extends Continuous Integration by automating the deployment of validated code changes to production environments. The CD pipeline of this project includes:

- **Deployment automation:** Use of deployment automation tools like Docker and Docker Swarm to ensure consistent and repeatable deployments across different environments.

- **Staging environments:** Deployment to staging test environments with production-like settings for further testing and validation before production releases.

### Monitoring and Logging

Continuous monitoring and logging are essential for maintaining system reliability and performance. This project's approach is to store the application's logs in a file in real time for further analysis if ever needed.

### System Architecture Overview

The system architecture for this project has been designed to ensure robust data management, ease of deployment, and efficient processing. The key components are as follows:

1. **Data Storage:** A MongoDB cluster is employed for data storage, utilizing replication and sharding techniques to enhance efficiency and maintain data integrity.
2. **Containerized APIs and Models:** All APIs and models are containerized using Docker. This approach ensures replication across various environments and simplifies the deployment process.
3. **Orchestration and Deployment:** A Docker Swarm cluster is utilized to orchestrate and manage the deployment of containers across multiple nodes as needed.
4. **Dashboard and Webserver:** A Python-based webserver is responsible for serving a dashboard application, which facilitates user interaction with the APIs and data stored in the MongoDB database.

### Data Processing Workflows

The system's data processing workflows are illustrated in the accompanying diagram and can be summarized as follows:

1. **Initial Data Submission:** The process begins when a dataset in the "JSONLines" format, containing various tweets, is submitted to the primary endpoint of the main API.
2. **Data Distribution and Conversion:** Upon receiving the dataset, the main API forwards it to the "profiling" and "propagation" APIs. Simultaneously, the dataset is converted from JSONL to CSV format and sent to the "textual" API.
3. **Data Processing and Storage:** Each API processes the dataset in its respective format and stores the results in the MongoDB database. These results can later be queried via the dashboard application.
4. **Multimodal Processing:** Tweets containing images undergo additional processing through the Multimodal API. The results from this process are also stored in the MongoDB database, available for retrieval by the dashboard application.

5. **Alternative Data Submission Method:** The dashboard application includes a drag-and-drop interface for dataset uploads, which automates processing and eliminates the need for manual HTTP requests or API interactions, thus reducing the complexity. However, this feature is still under development and may not perform optimally with large datasets, potentially slowing down the application during processing.

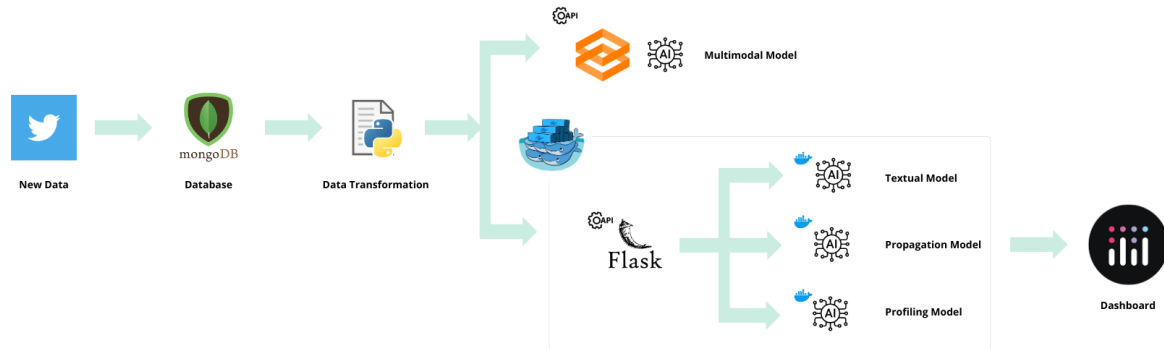


Figure 1 – Block diagram with the functional design of the prototype

## 6. Conclusions

In conclusion, this report provides a detailed overview of the processes involved in creating a platform capable of generating actionable insights for case studies in elections and immigration. From data ingestion to API integration and frontend development, each component has been carefully crafted to ensure accuracy, efficiency, and usability. By leveraging a DevOps framework, the platform is designed to be adaptable and scalable, capable of continuous improvement. Ultimately, this solution not only delivers valuable insights but ensures that these insights are easily accessible, driving informed decision-making for end-users.

## Terminology and Acronyms

API	Application Programming Interface
DevOps	Software development and IT operations
CD	Continuous Deployment

