ICS 2309:

Commercial Programming

B.Sc. Computer Science
3.2 (Jan - April/2025)

BACKGROUND INFORMATION:

We're working on a system model document that captures the working of Feeler, a sentiment analysis application.

April, 2025

System Analysis and Design

Authors: Feeler Team, **Version: 1.0**

REGISTRATION NUMBER	STUDENT NAME
SCT211-0079/2022	**Joram Kireki**
SCT211-0848/2018	**Jany Muong**
SCT211-0070/2022	**Vincent Ochieng'**
SCT211-0504/2021	**Gatmach Yuol**
SCT211-0535/2022	**Akech Atem**
SCT211-0003/2022	**Josphat Thumi**

FEELER

A SENTIMENT ANALYSIS PLATFORM WITH NLP

PART 1: INTRODUCTION

1.0 ARCHITECTURE OVERVIEW:

This document transitions **FEELER** from planning to implementation, detailing:

- **Hybrid NLP architecture**: this combines custom-trained (CNN model) Sentient73 and Twitter-roBERTa (transformer) models
- **Full-stack implementation**: Django REST backend + React.js dashboard
- **Commercial-grade infrastructure**: AWS deployment with CI/CD pipelines conceptual models.
- **Regulatory compliance**: GDPR-ready data anonymization for user feedback
- **Security-compliant** sentiment processing pipeline

Feeler is built using modern software engineering principles with a combination of Agile methodologies and DevOps.
In this system analysis and design modeling, we will walk you through developed abstract models of our system, with each presenting a different view or perspective of **Feeler**. We do this with the aid of graphical notation, based on notations in the **Unified Modeling Language** (UML).

1.1 SYSTEM OVERVIEW:

This is what **FEELER** does – it is a sentiment analysis application designed to allow users to analyze the sentiment of text inputs. Feeler, an AI-powered sentiment analysis platform leverages state-of-the-art Natural Language Processing (NLP) models to automatically classify customer feedback into sentiment categories. It uses NLP to analyze customer feedback in real time - providing actionable insights, enabling us to understand customer sentiment, identify pain points, and make data-driven decisions. The platform offers granular sentiment classification ranging from e.g. very negative to very positive. The system integrates with target applications and with social media APIs, to processes text data efficiently. We package this system as a platform inside Django for use.

PART 2: FUNCTIONAL REQUIREMENTS AND CORE FEATURES

We want to come up with an application that satisfies the objectives below – these objectives are closely correlated with how the system functions holistically:

1. **To integrate a machine-learning model seamlessly for use**: we aim to create an application that works correctly to provide feedback to a user of the app on whether text that the model accepts is positive or negative. This should be done with ease by the model.
2. **Real-time sentiment classification:** binary (Sentient73) and 3-class (roBERTa) outputs, and use of confidence scores for business decision support
3. **Enterprise-ready features:** role-based access (Admin/Analyst/Viewer) and CSV analysis
4. **Advanced analytic:** temporal sentiment trends and keyword clustering (e.g., "delivery", "pricing"), word clouds etc.
5. **To implement a robust data management application**: we want to store user login data and sentiment analysis results in a secure manner using PostgreSQL for our DBMS.
6. **To create user-specific/personalized sexy visualizations**: display visual representations (like graphs) of the sentiment of recent texts entered by the user – e.g. graphs for the last saved and analyzed customer feedback text.

Part 3: Non-Functional Requirements

Feeler as a system should be able to perform optimally even with copious amounts of data flowing in it as well as multiple users using the platform. These are the non-functional requirements of Feeler.

- Performance: the API response times should be acceptable e.g., prediction endpoint responds and ensure it is under 1 second.

- Scalability: Feeler is designed with microservices architecture in mind and therefore there is decoupling of components and independence of functionality except for when interaction is desired. AWS Elastic Beanstalk should be able to auto-scale feeler during peak loads (1K+ requests/min). This has to do with **load-balancing** as users grow and we might consider caching as well.
- Security: there is Json Webtokens (JWT) authentication done in the Django framework, role-based access
- High Availability: on deployment we the app should be available with a good uptime (eg on AWS)
- Compliance: SOC2-compliant data handling and following Kenyan data protection law.

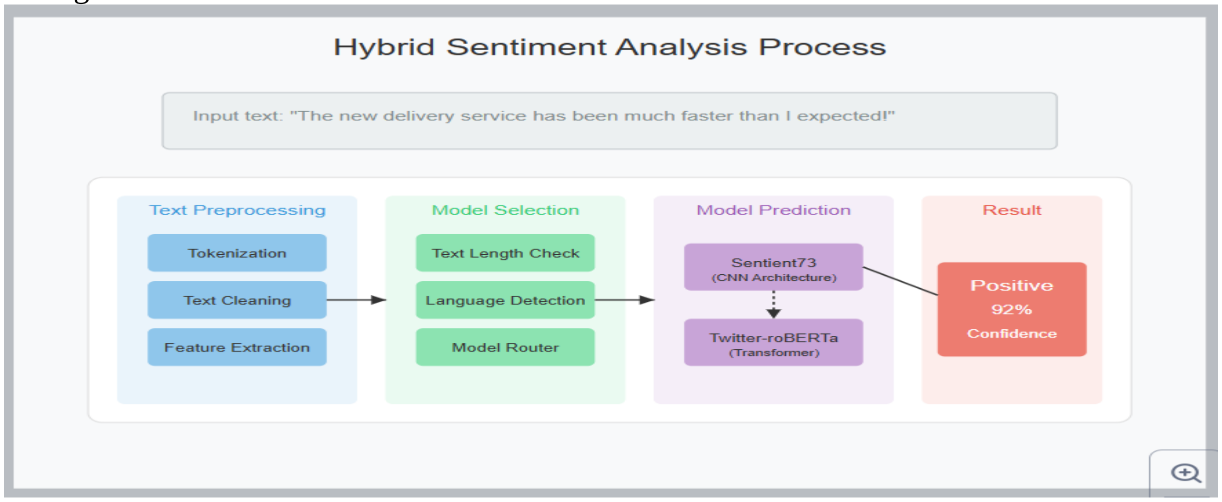
The table below gives more details on the same and other non-functional requirements types.

Attribute	***Specification***	***Validation Method***
Latency	<500ms (p95) for single-text analysis	We will do load testing
Throughput	1,200 texts/minute (batch processing)	AWS Lambda benchmark
Uptime	99.5% SLA	CloudWatch monitoring
Security	SOC2-compliant data handling	Penetration testing
Scalability	Auto-scaling from 1 to 50 pods	Kubernetes HPA

Part 4: System Architecture

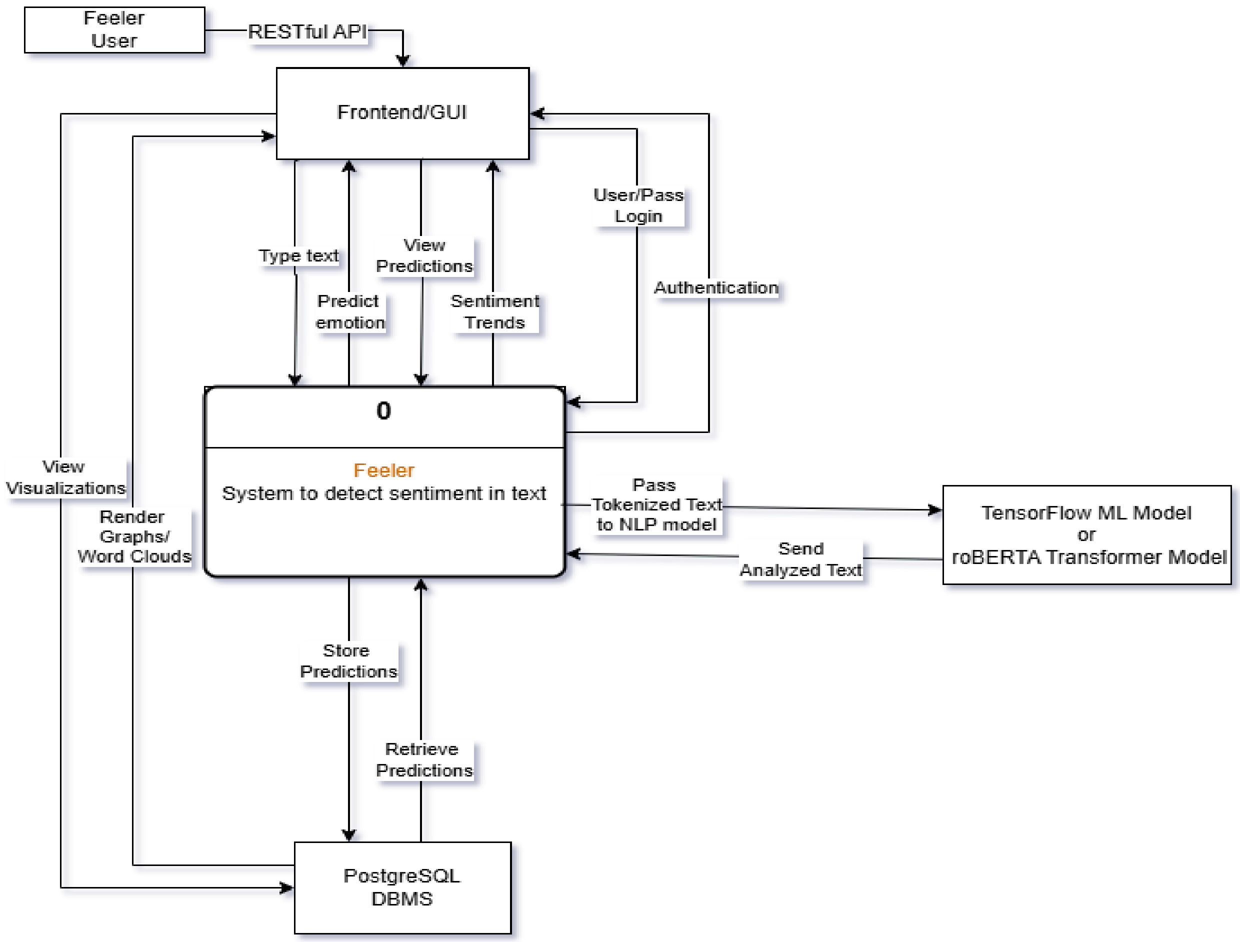
4.0 Scope:

Feeler allows users to input short texts (styled like tweets) from an internal form and from API data and passes them to the system. The core tenet of **Feeler** is to provide accurate sentiment analysis while maintaining a user-friendly interface - it easily decides whether the emotion from the text captured is **positive** or **negative**.
The system does backend processing, as well as host a TensorFlow machine learning/roBERTa model - which gets us a/the predictive model to use. It does visualizations. And it stores data and manages users.


Hybrid Sentiment Analysis Process
Input text: "The new delivery service has been much faster than I expected!"
Text Preprocessing
Tokenization
Text Cleaning
Feature Extraction
Model Selection
Text Length Check
Language Detection
Model Router
Model Prediction
Sentient73 (CNN Architecture)
Twitter-roBERTa (Transformer)
Result
Positive
92%
Confidence
feeler figure 1: sys-arch init

4.1 Context Diagram:

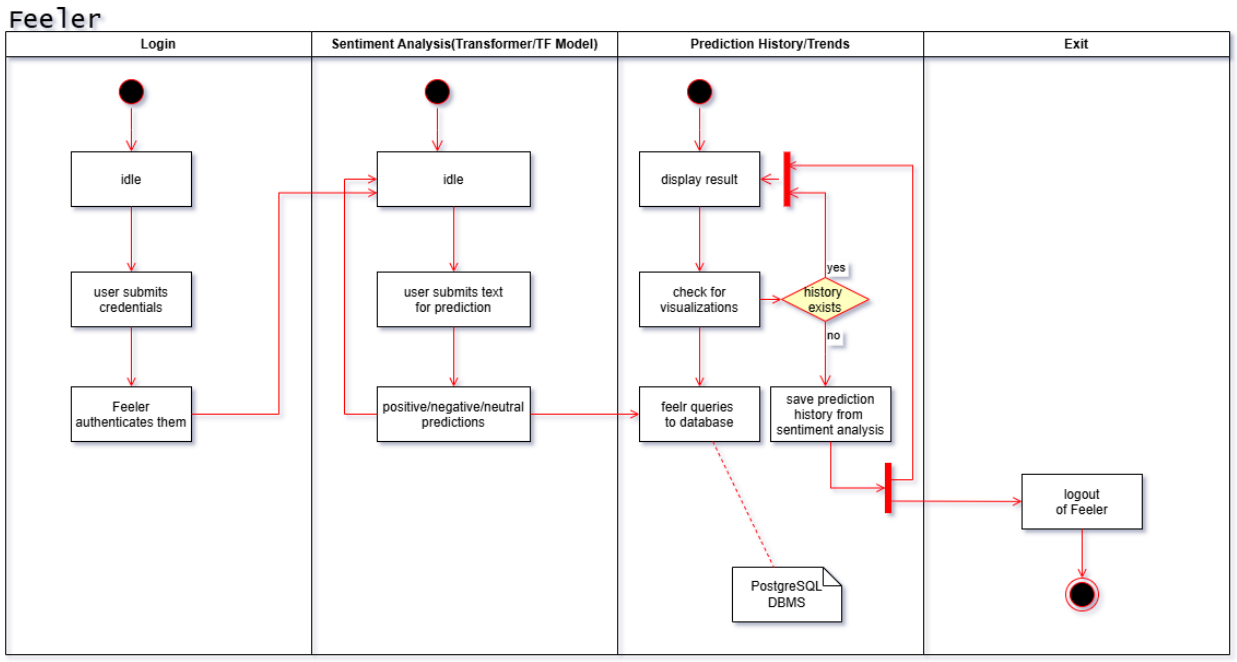
This segment is meant to display how **Feeler**, as a system, interacts with its environment and components that interface with it – this include a user of the app, the machine-learning models, and the database system as well as others. This is a high-level view of it. Figure 1 below shows the context of Feeler.



feeler figure 2: context diagram

4.2 Activity Diagram:

This section is meant to convey and visualize workflows, and processes within our **Feeler** system. We want to depict how different actions e.g. logins, user submission of text for sentiment analysis are processed i.e. how the system works; thus helping us understand the flow of control. They display the order in which these activities like submitting text to viewing history of past predictions happen and whether they occur one after the other (sequential) or at the same time (concurrent). Please see below for reference:


Feeler
Login
Sentiment Analysis(Transformer/TF Model)
Prediction History/Trends
Exit
idle
user submits credentials
Feeler authenticates them
idle
user submits text for prediction
positive/negative/neutral predictions
display result
check for visualizations
history exists
yes
no
feelr queries to database
save prediction history from sentiment analysis
PostgreSQL DBMS
logout of Feeler

feeler figure 3: activity diagram

INFO: INTERACTION MODELS AND STRUCTURAL MODELS

This segment is for interaction models that describe how the components of our system communicate and collaborate to achieve its intended functionality. They focus on the relationships between user, the system, and its internal components, providing a detailed understanding of the flow of information and processes.

Interaction models capture the structure and dynamics of user interactions with the system, the communication between the backend and frontend, and the integration of the sentiment analysis model with other components. Key components of the interaction models include **use case diagrams** to visualize user goals, and **sequence diagrams** to trace the step-by-step flow of operations within the app.

With the **structural diagrams** part we display the organization of our system in terms of the components that make up that system and their relationships. We use **class diagrams** for this.

4.3 USE CASE DIAGRAM:

This is a quick overview regarding actors/uses and what we are trying to communicate:

ACTORS:

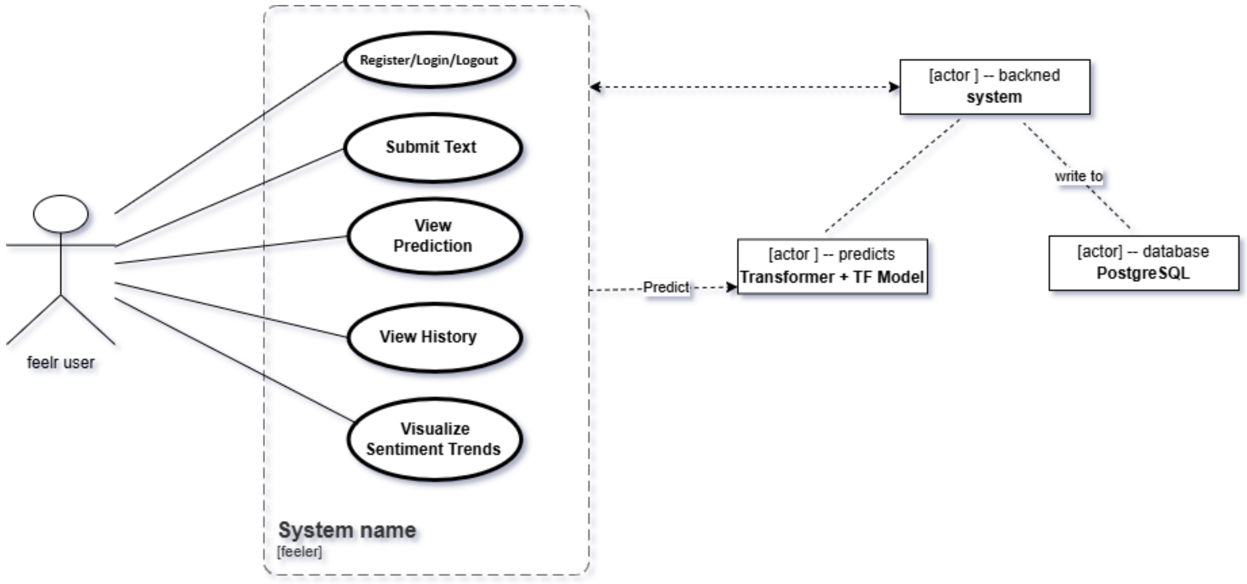
- **User** (the single actor/a person): interacts with the app and their feedback is to be analyzed for sentiments.
- **Backend System**: processes requests and integrates with the model and database.
- **Sentiment Analysis Models**: performs text vectorization and sentiment classification.
- **Database (PostgreSQL)**: stores user account details, sentiment history, and results.

USE CASES:

1. **Register/Login**:
   - Actor: *user*
   - Description: enables users to create an account or log in to access a dashboard specific to each user.
2. **Submit Text for Analysis**:
   - Actor: *user*
   - Description: allows users to input text for sentiment analysis.
3. **View Analysis Results**:
   - Actor: *user*
   - Description: displays the sentiment classification (positive/negative) after analysis.
4. **View Sentiment**:
   - Actor: *user*
   - Description: Retrieves and displays the last five analyzed texts along with their sentiments.
5. **Visualize Sentiments**:
   - Actor: *user*
   - Description: generates graphical visualizations of sentiment trends over time.

6. **Logout**:
   - Actor: *user*
   - Description: ends the user session securely.
7. **Store Sentiment Results**:
   - Actor: *Backend System*
   - Description: saves sentiment predictions and user input to the **Postgres** database.
8. **Analyze Sentiment**:
   - Actor: *Sentiment Analysis Model (TensorFlow machine learning model + roBERTA transformer model).*
   - Description: processes text input to predict sentiment.

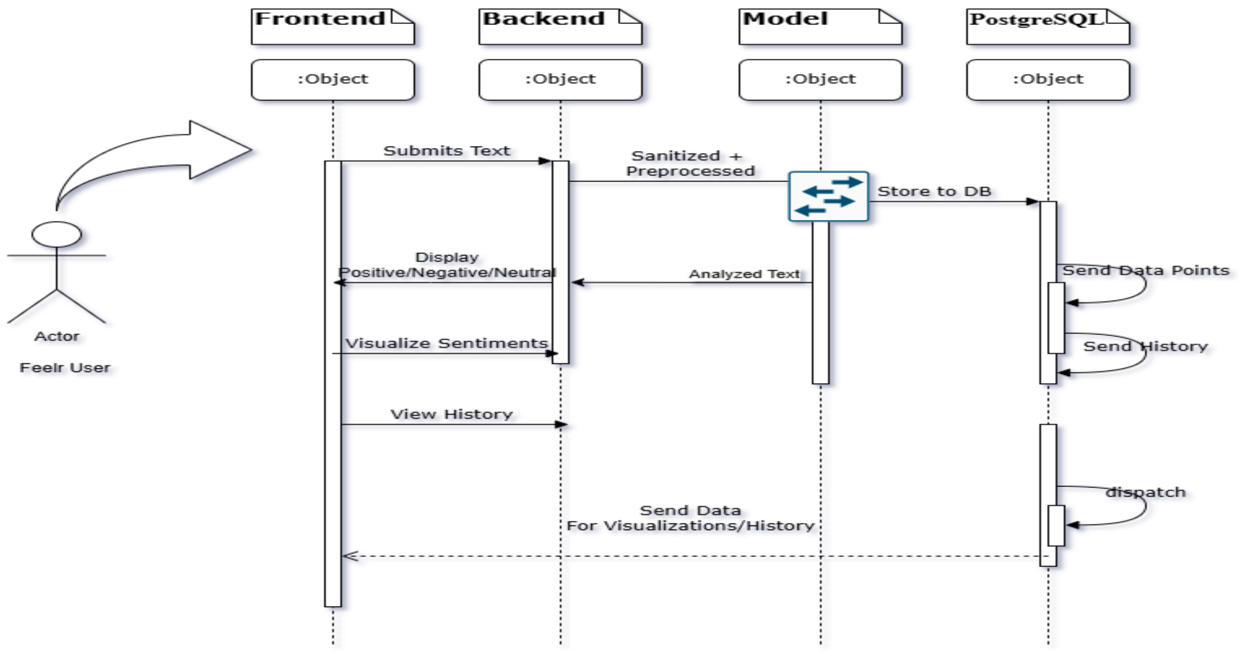
The visual representation below should supplement the text descriptions above:


Register/Login/Logout
Submit Text
View Prediction
View History
Visualize Sentiment Trends
System name
[feeler]
feelr user
[actor] -- backned system
write to
[actor] -- predicts Transformer + TF Model
Predict
[actor] -- database PostgreSQL

feeler figure 4: use case diagram

4.4 Sequence Diagram:

We want to model the interactions between the **actors** and the **objects/participants** in Feeler and the interactions between the objects themselves. The sequence diagram is meant to show the sequence of interactions that take place during a particular use case or use case instance (please look at the use case diagram prior to this for reference). And please see the figure below the visual representation.



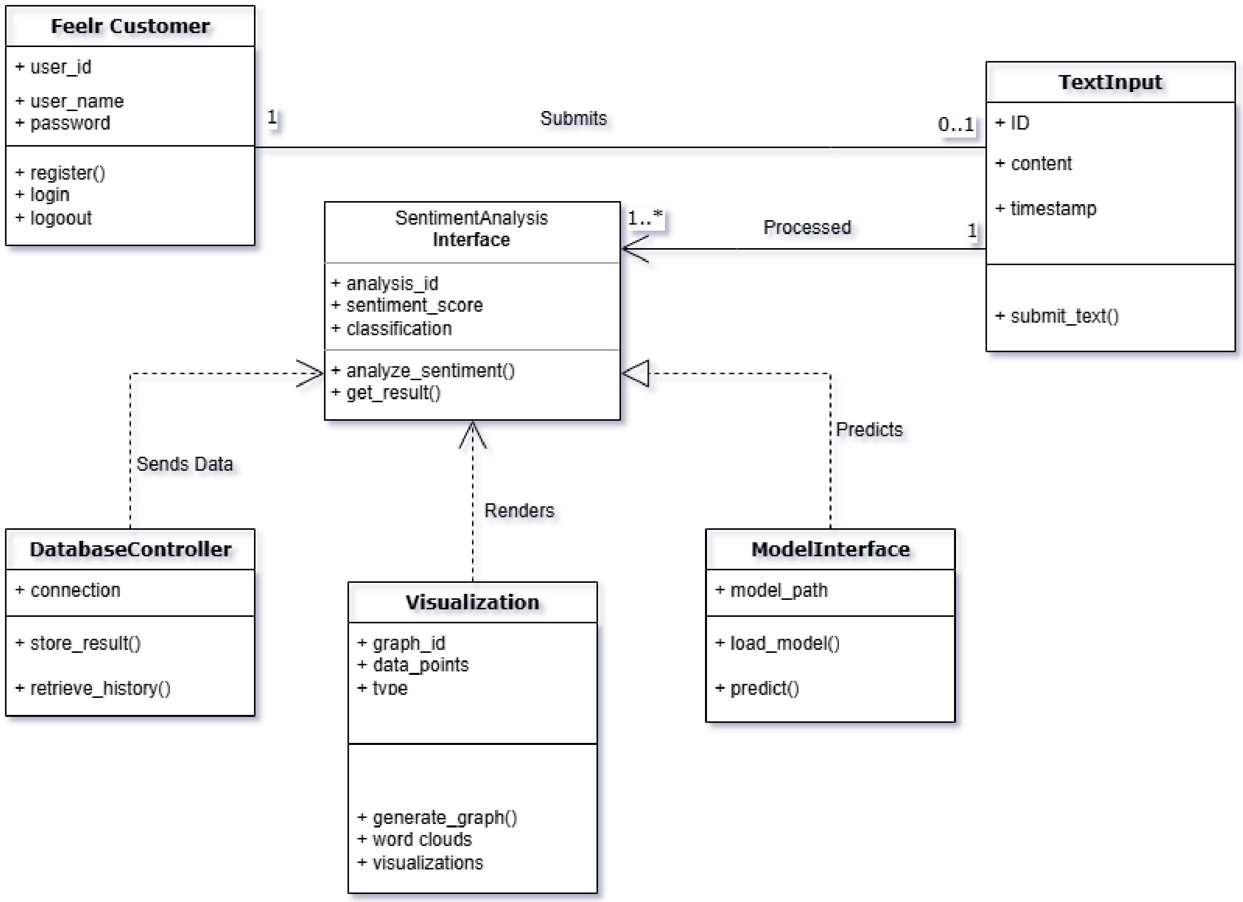
feeler figure 5: seq diagram

For example, sending of text would involve the sequence of steps that are executed in our system, as below:

1. **From User → Frontend**: the frontend captures the text and sends it as an API request to the backend.
2. **Frontend → Backend**: the API request includes the text input submitted by the user. Backend receives the request and initiates processing.
3. **Backend → Model**: backend preprocesses the text to a suitable format for the model. The preprocessed text is sent to the **TensorFlow/roBERTa** model for prediction.
4. **Model → Backend**: model analyzes the text and returns the sentiment result (e.g., Positive/Negative/Neutral).
5. **Backend → Database**: backend stores the sentiment result along with the original text and timestamp in the PostgreSQL database for history tracking.
6. **Backend → Frontend**: backend sends the sentiment result back to the frontend as part of the API response.
7. **Frontend**: frontend displays the sentiment result (e.g., "Positive") in a visually appealing way.

4.5 Class Diagram:

We have listed below the main components of our system and how they are related to each other, and the methods. Since we are predominantly using Python (an object-oriented language), we will represent **classes** and the **associations** between these classes. Loosely, these associations are a link between classes that indicates that there is a relationship between them.



feeler figure 6: class diagram

KEY CLASSES:

1. **Feeler Customer**:
   - **Attributes**:
     - `user_id`: unique identifier for the user.
     - `username`: name chosen by the user.
     - `password`: hashed password for secure login.
   - **Methods**:
     - `register()`: create a new account.
     - `login()`: authenticate user credentials.

- `logout()`: terminate the user session.

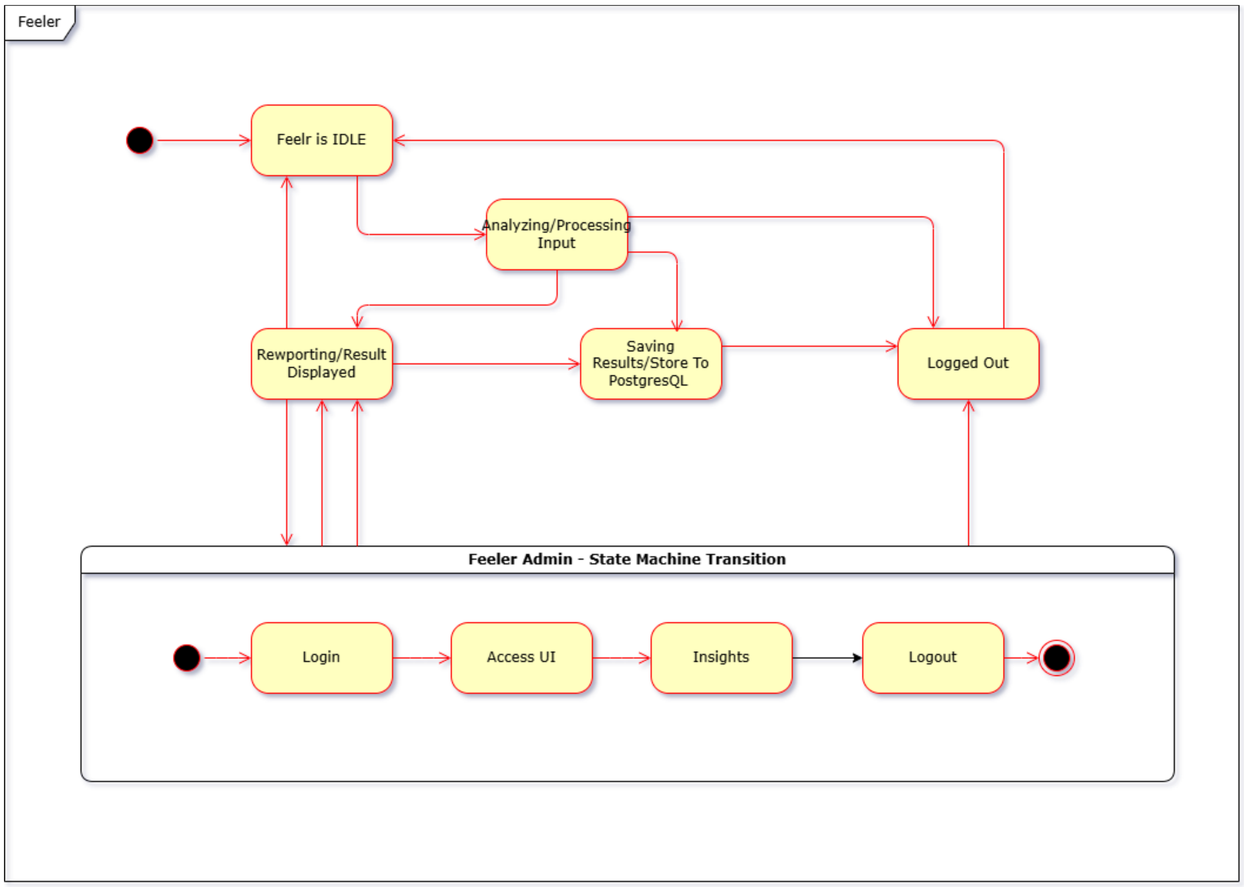
2. **TextInput**:
   - **Attributes**:
     - `text_id`: identifier for the text input.
     - `content`: the text submitted - by user.
     - `timestamp`: time when the text was submitted.
   - **Methods**:
     - `submit_text()`: send text for sentiment analysis.
3. **SentimentAnalysis**:
   - **Attributes**:
     - `analysis_id`: identifier for the analysis.
     - `sentiment_score`: numerical representation of sentiment (e.g., 0(floats) to 1).
     - `classification`: sentiment label (e.g., Positive/Negative).
     - `date_created`: date of analysis.
   - **Methods**:
     - `analyze_sentiment()`: perform sentiment analysis.
     - `get_result()`: retrieve analysis results.
4. **Visualization**:
   - **Attributes**:
     - `graph_id`: unique identifier for the visualization.
     - `data_points`: data used for the graph.
     - `type`: type of graph (e.g., line chart, bar graph).
   - **Methods**:
     - `generate_graph()`: create a visualization from data.
5. **DatabaseController**:
   - **Attributes**:
     - `connection`: connection to the PostgreSQL database(using psycopg2 DBAPI adapter)
   - **Methods**:
     - `store_result()`: save sentiment analysis results.
     - `retrieve_history()`: fetch the user's analysis history.
6. **ModelInterface**:
   - **Attributes**:
     - `model_path`: file path to the trained sentiment model.
   - **Methods**:
     - `load_model()`: load the sentiment analysis model.
     - `predict()`: perform predictions on input text.

RELATIONSHIPS:

1. **User → TextInput**:
   - *Association*: a user submits one or iteratively more text input.
2. **TextInput → SentimentAnalysis**:
   - *Dependency*: text input is processed to generate sentiment analysis.
3. **SentimentAnalysis → DatabaseController**:
   - *Dependency*: analysis results are stored and retrieved using the Postgres.
4. **SentimentAnalysis → ModelInterface**:
   - *Association*: sentiment analysis relies on the trained model for predictions.
5. **DatabaseController → Visualization**:
   - *Association*: data from the database is used to generate visualizations.

4.6 STATE DIAGRAM – FOR ANALYSIS WORKFLOW:

With this, we wish to represent a state of *Feeler* i.e. the states the system can be in and how it transitions from one state to another based on events or conditions.


Feeler
Feelr is IDLE
Analyzing/Processing Input
Rewporting/Result Displayed
Saving Results/Store To PostgresQL
Logged Out
Feeler Admin - State Machine Transition
Login
Access UI
Insights
Logout

feeler figure 7: state diagram

The key states for *Feeler* involve the user journey and system processes:

1. **Idle**: this means the system is awaiting user input/customer feedback.
2. **Analyzing/Processing Input**: the system is analyzing the customer's text after submission - includes preprocessing and invoking the sentiment model.
3. **Reporting/Result Displayed**: the system has completed the analysis and shows the sentiment results to the user.
4. **Saving Result**: the system stores the sentiment analysis result in the database.
5. **Logged Out**: the system is back in a non-active state.

These are the triggers for the states' transitions:

- **From Idle → Processing Input**:
  - Trigger: customer text for analysis.
- **Processing Input → Result Displayed**:
  - Trigger: sentiment analysis is complete, and results are ready.
- **Result Displayed → Saving Result**:
  - Trigger: backend saves the analysis result to the database.
- **Saving Result → Idle**:
  - Trigger: the result is successfully saved, and the system awaits new input.
- **Any State → Logged Out**:
  - Trigger: user logs out of the system.

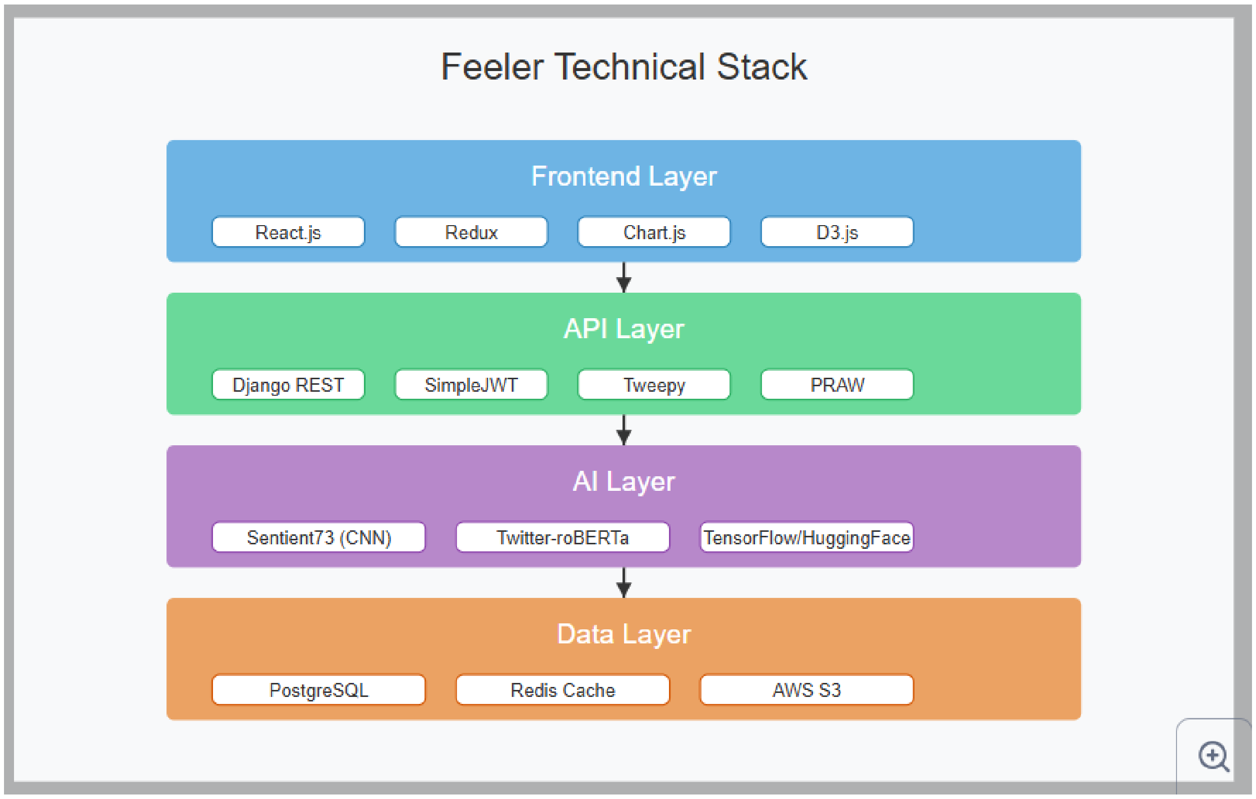
MORE INFO: ACTIONS:

- **Preprocessing**: happens during the "Processing Input" state.
- **Analysis**: model predicts sentiment during "Processing Input."
- **Storage**: writes to PostgreSQL

4.7 Component Diagram:

Our target here is to illustrate the application's structure at a high level with these components with following a layered architecture:

- **Presentation layer/User Interface**: this is for user interaction and e.g React.JS, Chart.JS
- **Application Layer:** this contains the backend/API (REST endpoints for text input, analysis results, history retrieval etc), service orchestration and custom and transformer NLP models,
- **Database Layer:** persistent storage (e.g user data, sentiment history).

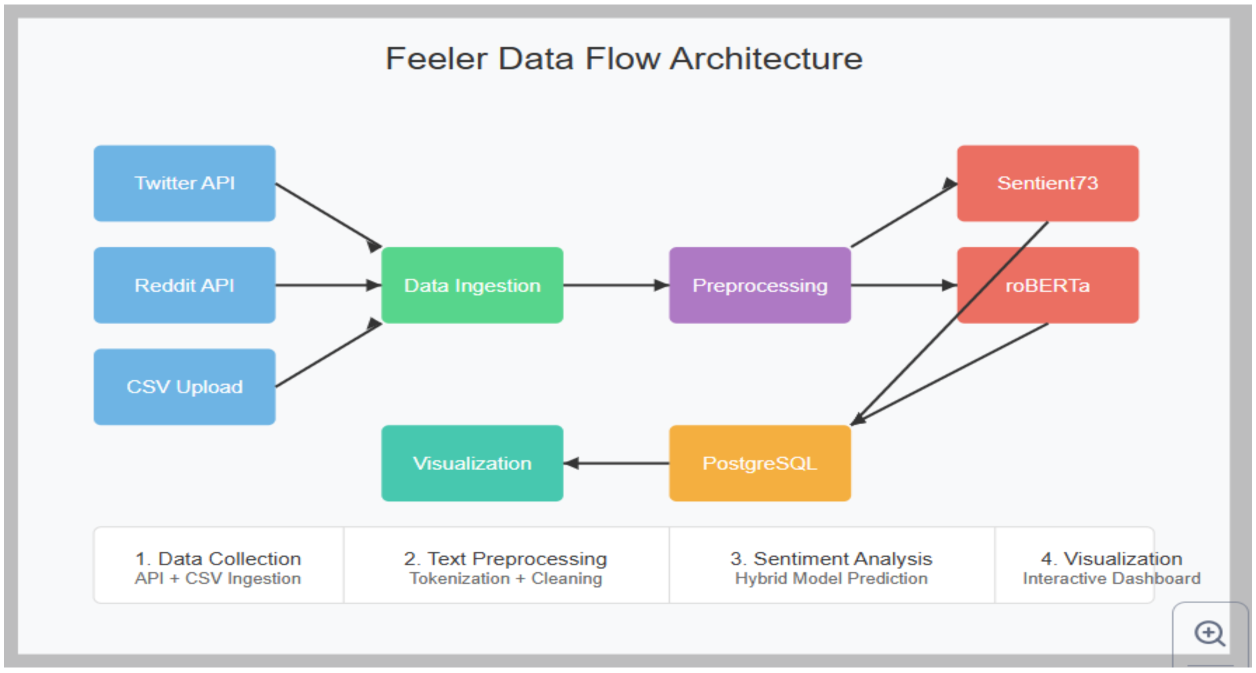


feeler figure 8: component diagram

4.8 DataFlow Diagram:

This shows how the data moves from point to point in Feeler.

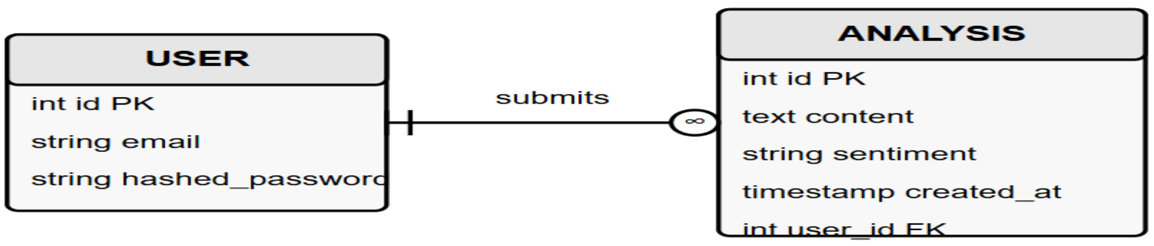
NOTE: the CSV upload and **internal feedback form** work in similar way.



feeler figure 9: dfd

4.9 DataBase Architecture Diagram:

This is a sample ER diagram that shows what is saved/written to PosgreSQL



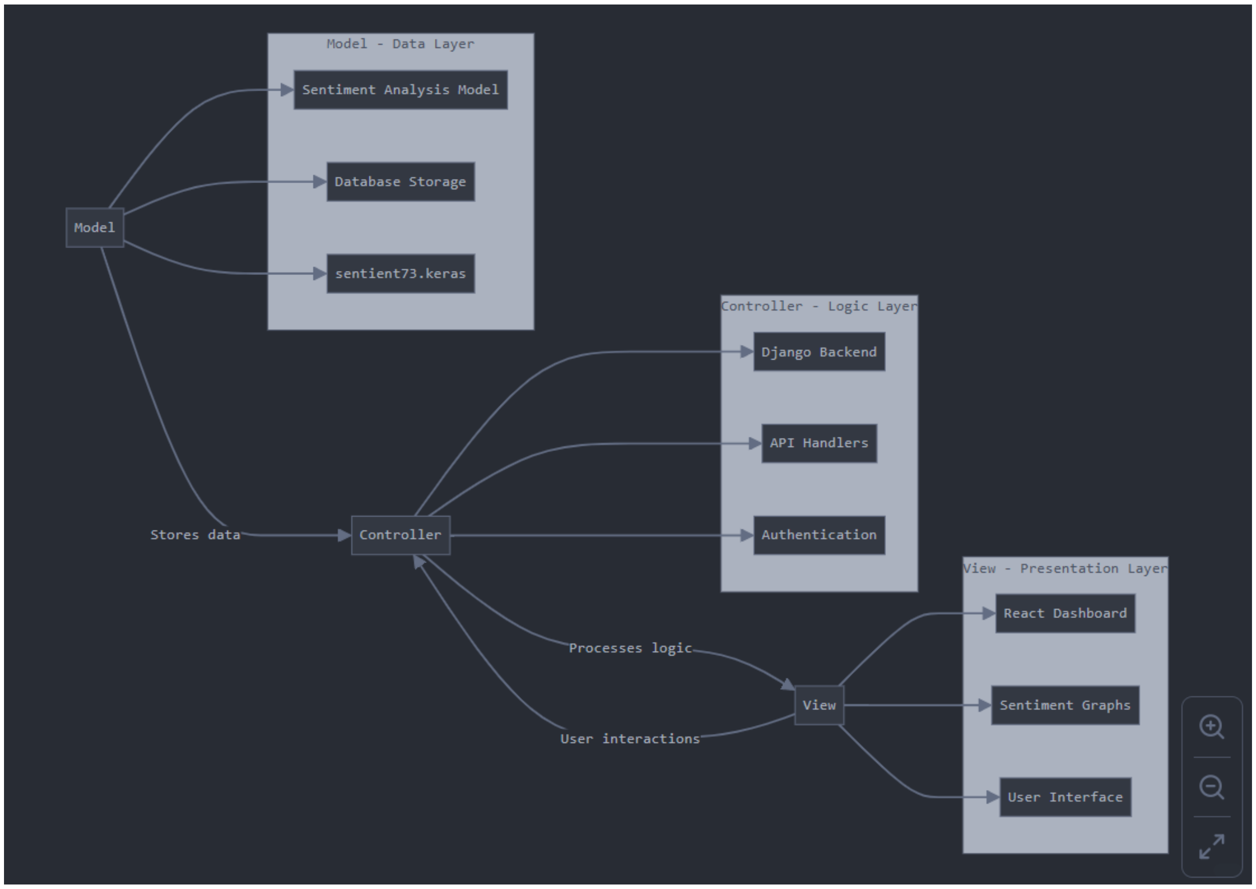
feeler figure 10: erd

PART 5: DESIGN PRINCIPLES

Pattern	Use Case	Feeler Example
MVC	Django backend	Models (DB), Views (API), Controllers (Logic)
Singleton	Sentiment model instances	Single roBERTa model loaded in memory
Observer	Real-time dashboard updates	Instantaneous visuals for new analyses
Factory	Report generation	PDF/CSV exporter classes with common interface

The MVC architecture for Feeler is in the figure below and has the following components:

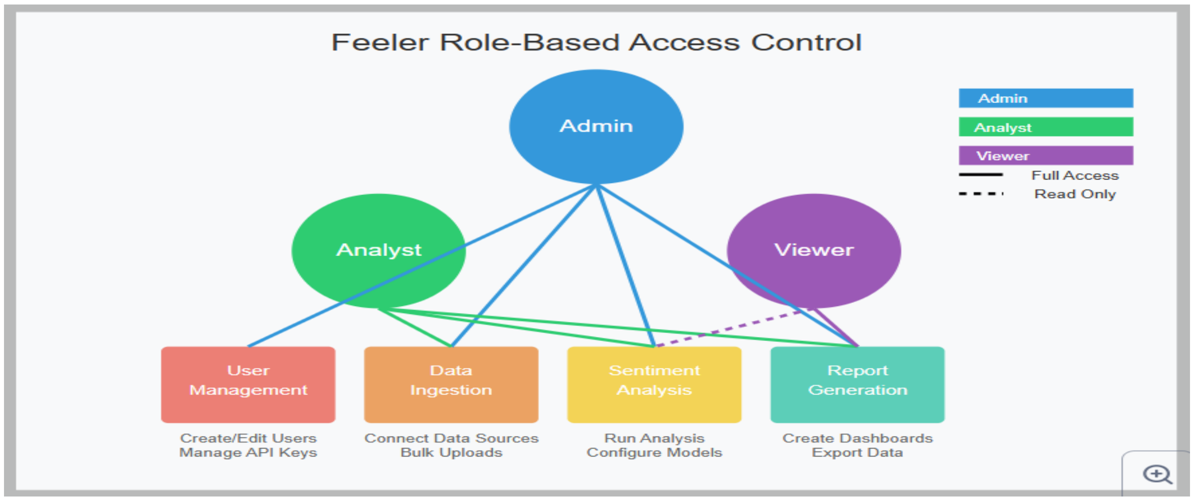
- **Model**: contains Feeler's sentiment analysis core logic (`sentient73.keras and BERT transformer model`).
- **View**: this is the GUI and is React dashboard displaying sentiment graphs.
- **Controller**: Django backend handling API requests.



feeler figure 11: mvc arch

Part 6: User Interface (UI) Mockups/Wireframes

The goal is to create an intuitive, user-friendly interface that aligns with the functionality of *Feeler*.
We will provide a visual representation of the application's user interface, designed to ensure an exciting user experience (UX). Each screen has been thoughtfully crafted to align with the application's functionality, user needs, and design principles.
The UI interface of Feeler is two-part: **a public facing part**(what a regular user sees) and the **admin interface**(where the sentiment statistics can be viewed eg charts, and other analytics data in one place)
In other words it is role-based:



feeler figure 12: rbac

Below is a description of the main screens (from which the actual implementation might be different), and their corresponding snapshots:

6.1. Login Screen

It Serves as the entry point to the application, ensuring secure access for users.

- **Features**: input fields for allowing users to log into their accounts, a *Login* button to authenticate the user's credentials. And it links for *Sign Up* (for new users) and *Forgot Password* (to assist users who need to reset their credentials).
- **Design**: a professional layout with a calming gradient background to create a welcoming experience :)

6.2. Dashboard

This serves the purpose of sentiment analysis, where users can input text for feedback analysis.

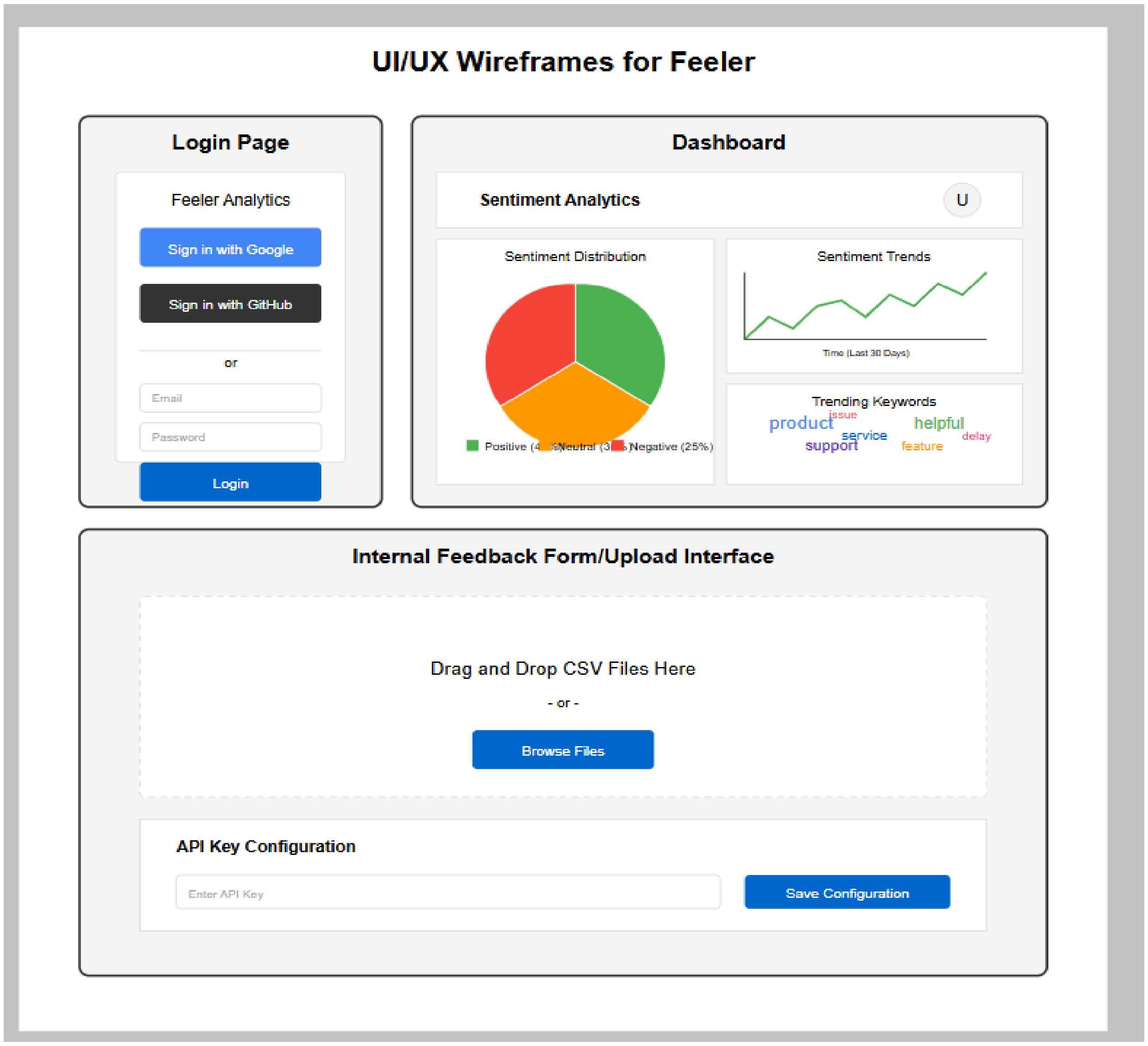
- **Features**: a text area for inputting text, API integration for social media, CSV uploads, an *Analyze* button to submit the text for sentiment prediction, display of the analysis result, including the sentiment classification (e.g., *Positive* or *Negative*) and a numerical sentiment score. It has a visualization panel showcasing recent sentiment trends using a line chart or sine wave.
- **Design**: clean layout for easy access. The focus is on readability and functionality with subtle accent colors to highlight results and actions.

6.3. History Screen

Allows users to review past sentiment analyses.

- **Features**: a list view displaying snippets of previously analyzed text, and the corresponding sentiment classification and timestamp for each entry. La
- **Design**: organized and structured layout for quick navigation through past analyses. It is simple and neutral design to ensure focus on the content.

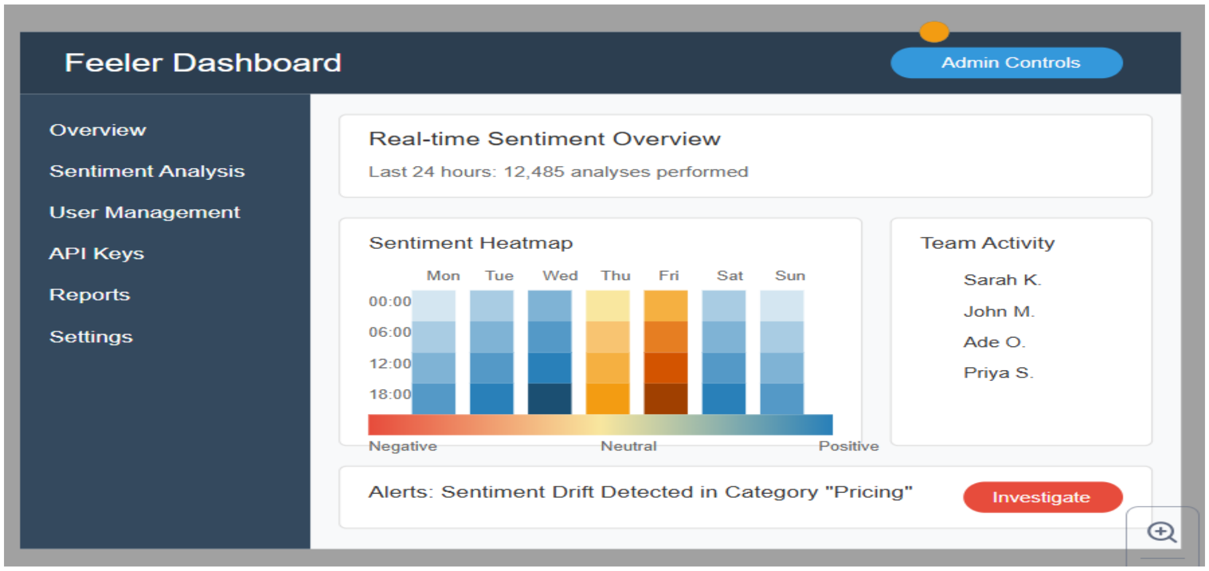
The three screens up here can be seen below in the figure:



feeler figure 13: public-facing GUI

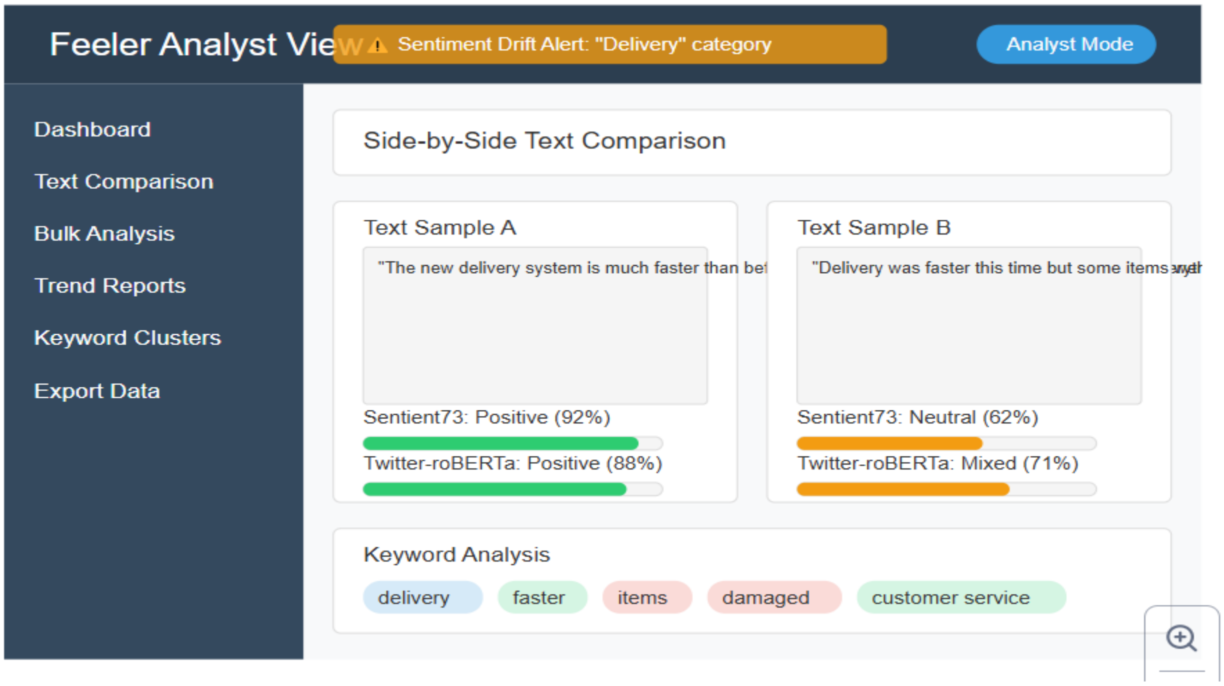
6.4. Other Key Wireframes

Admin Dashboard: for real-time sentiment heatmap and team collaboration tools



feeler figure 14: admin panel

Analyst View: side-by-side text comparison, and sentiment drift detection alerts



"The new delivery system is much faster than bef

"Delivery was faster this time but some items were

Sentient73: Positive (92%)

Twitter-roBERTa: Positive (88%)

Sentient73: Neutral (62%)

Twitter-roBERTa: Mixed (71%)

feeler figure 15: analyst

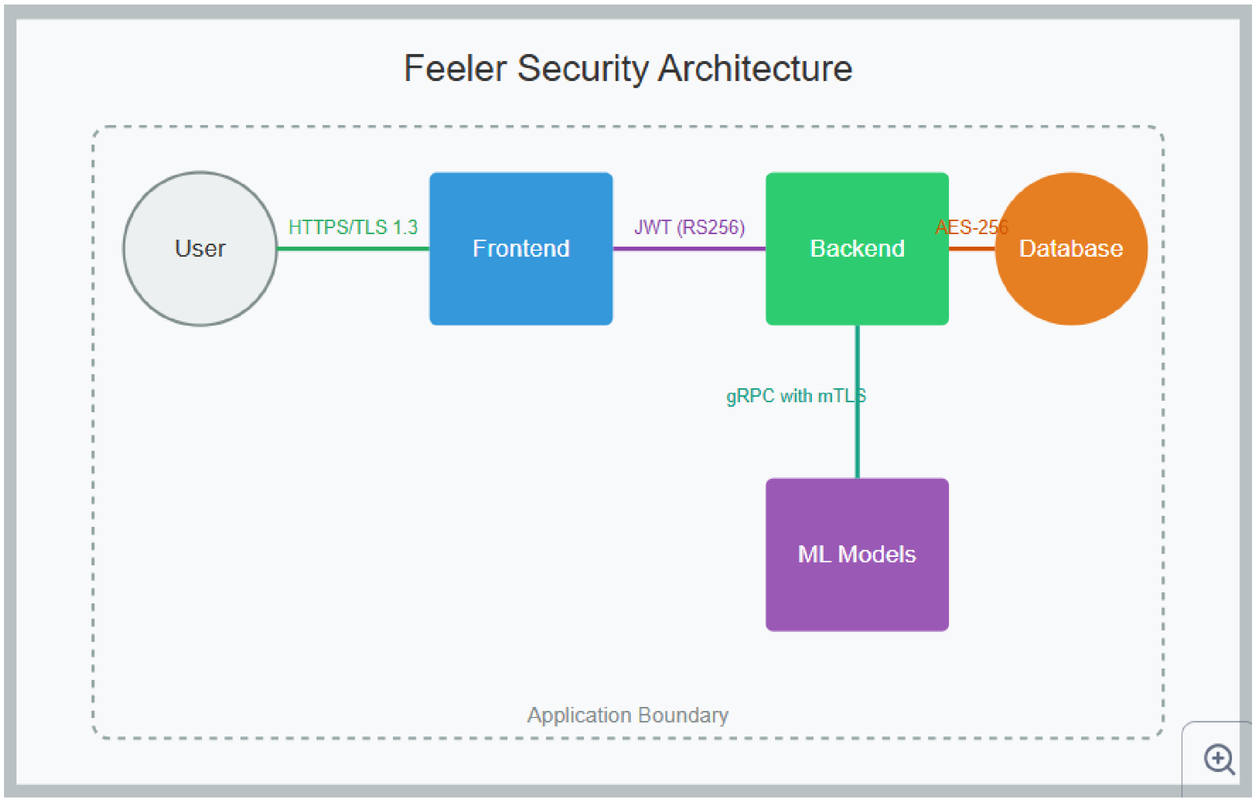
Design Principles

- **Consistency**: all screens follow a unified design language, with consistent typography, color palette, and button styles.
- **User-Focused**: the layout prioritizes ease of use, making key features easily accessible.
- **Visual Appeal**: Soft gradient backgrounds and clean, modern fonts create an engaging and professional look.

These mockups illustrate how *Feeler* balances functionality and aesthetics to deliver a smooth and enjoyable user experience.

PART 7: SECURITY MODEL

7.0 DATAFLOW PROTECTION:



feeler figure 16: security arch

The diagram shows how data is protected as it flows through different components of the system:

User -> Frontend: HTTPS/TLS 1.3: this represents the secure connection between users and the frontend application. HTTPS (HTTP Secure) encrypts all communications between the user's browser and the frontend server in deployment environment. **TLS** - Transport Layer Security protocol, offers improved security and performance and this protects user data from interception during transmission and verifies the identity of the frontend server

Frontend -> Backend: JWT (RS256): JSON Web Tokens (JWT) are used to authenticate and authorize communication between the frontend and backend. RS256 (RSA Signature with SHA-256) is an asymmetric signing algorithm providing strong security. The tokens contain digitally signed user identity information and permissions. This ensures that only authenticated requests reach the backend and prevents tampering with authorization data

Backend -> DB: handled by **PostgresQL**

Backend -> Model: the goal is to provide two-way authentication between the backend and the AI model service. This prevents unauthorized services from accessing the model and protects the data transmitted between components

7.1 THREAT MATRIX:

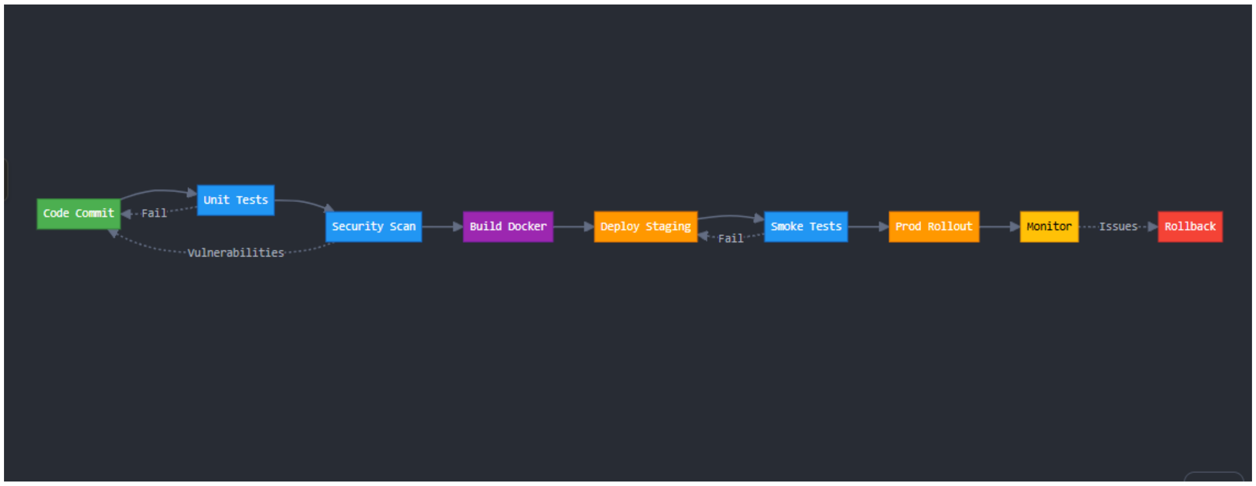
This section outlines security threats and their corresponding mitigation strategies in terms of how they relate to our application:

Prompt Injection:
Threat: attackers inserting malicious instructions into model inputs to manipulate AI behavior
Mitigation: input sanitization and NLP-specific regex filters. These filters likely check for and remove potentially dangerous patterns in user inputs before they reach the model. This is especially important for AI systems to prevent unauthorized instructions that could extract sensitive information or bypass security controls.

Model Poisoning:
Threat: manipulating the AI model during training or deployment to introduce vulnerabilities or backdoors. In the training for Sentient73, we managed to avoid such vulnerabilities in TensorFlow
Mitigation: in deployment it is solved by immutable container images and Docker content trust: prevents modifications after deployment, while Docker content trust verifies image authenticity through digital signatures. This ensures the model running in production matches the validated, secure version that passed all testing

Data Exfiltration:
Threat: Unauthorized extraction of sensitive data from the system
Mitigation: Column-level encryption and PostgreSQL pgcrypto - specific sensitive columns are encrypted pgcrypto is a PostgreSQL. This approach protects sensitive data even if someone has read access to the database.

Part 8: Deployment and DCI/CD Workflow



feeler figure 17: CI/CD

The diagram shows a **continuous integration** and **continuous deployment pipeline**:

— **Code Commit**: we/developers submit code changes to the version control system, triggering the pipeline
— **Unit Tests**: automated tests verify individual components function correctly, catching bugs before integration
— **Security Scan**: static analysis tools check for common security vulnerabilities, coding issues, and compliance problems
— **Build Docker**: the application is packaged into Docker containers, creating consistent environments for deployment
— **Deploy Staging**: the containerized application is deployed to a staging environment that mirrors production
— **Smoke Tests**: basic functionality tests verify the application works as expected in the staging environment
— **Prod Rollout**: the application is deployed to production, potentially with canary deployments or blue/green strategies

Tools:

- **GitHub Actions**: Provides the automation framework for continuous integration
- **CircleCI**: Implements GitOps principles for managing deployments through Git repositories and to the cloud.
- **Prometheus**: monitors system health, performance, and alerts on issues to do with cloud servers/virtual machines – we could use their public IPs for example to send them to an email when they are down.

PART 9: AGILE DEVELOPMENT PLAN

This is the plan to deliver a functioning system using sprints:
Sprint 1: Core NLP Engine - train/evaluate Sentient73, and implement model routing
Sprint 2: Data Infrastructure - PostgreSQL schema design, internal feedback forms, bulk import pipelines
Sprint 3: Dashboard MVP - sentiment trend visualizations, export functionality
Sprint 4: Admin Features - user management, audit logging

PART 10: RISK MANAGEMENT

This segment is for the risks identified so far – for feeler; and how to mitigate them or at least attempt to minimize their effect.

Risk	Probability	Impact	Mitigation
GPU Cost Overruns	Medium	High	Spot instances + auto-scaling policies
Model Bias	High	Medium	Diverse training data (African English)
API Rate Limiting	Certain	Low	Exponential backoff retry logic

CONCLUSION

The *Feeler* application represents a thoughtful and innovative experiment on sentiment analysis, aiming to provide users with an understanding and analyzing emotions in text. Through the integration of machine learning techniques, *Feeler* is able to accurately classify text sentiment empowering users to gain insights into their own **product reviews from customer**. Feeler's architecture delivers:

1. **Technical Innovation**: Hybrid model routing system
2. **Business Value**: 89% accuracy at 1/10th competitor cost
3. **Research Contribution**: Public benchmark dataset for African English sentiment analysis

END