Issue Lenz: GitHub Issue Scrapper Using LLMs

Final Year Project

Report by

Naima Yaqub (380890)

Ghania Sarwar (375877)

In Partial Fulfillment

Of the Requirements for the degree

Bachelor of Engineering in Electrical Engineering (BEE)

School of Electrical Engineering and Computer Science National University of Sciences and Technology, Islamabad, Pakistan

(2025)

DECLARATION

We hereby declare that this project report, entitled "**IssueLenz: GitHub Issue Summarizer Using LLMs**," submitted to the SEECS, is a record of our original work done under the guidance of Supervisor "Dr. Hassaan Khalique," and that no part has been plagiarized without citations. Also, this project work is submitted in partial fulfillment of the requirements for the degree of Bachelor of Electrical Engineering.

Team Members	**Signature**
Naima Yaqub	_____
Ghania Sarwar	_____

Supervisor:	**Signature**
Dr. Hassaan Khalique Qureshi	_____

Co-Supervisor:	**Signature**
Ms. Ayesha Khalique	_____

Date: **Place:**

_____ _____

DEDICATION

We would like to dedicate our work to our teachers and parents without whom this could not have been possible.

ACKNOWLEDGEMENTS

We would like to wholeheartedly thank our advisor Dr. Hassaan Khalique and co-advisor, Ayesha Khalique, for helping us throughout this final year project. Without their keen guidance and support, we would not have been able to meet the project's scope in time.

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

We The project “IssueLenz” aims to streamline the process of analyzing and summarizing GitHub issues using large language Models (LLMs). Developers often face difficulty understanding the context of issues across repositories due to the sheer volume of open-source contributions. Also, it takes a lot of time to manually read each issue. This toll provides a solution by fetching Github issues, selecting filters to get specific type of issues, summarizing them in easy-to-understand language and enavbling export in various formats such as Excel, Word and JSON which can be used for further processing. Using Streamlit, langchain, Ollama, OpenAI, HuggingFace and Deepseek APIs, this project also provides a easy to use user interface, caching mechanism, API rate limit handling, base models (included in the package) and dynamic model integration options using API for users which provides enhanced user experience and performance, saving them both time and effort.

Chapter 1

1. INTRODUCTION

In today's software development world, co-operation and openness play a dominant role in open-source communities. GitHub, the world's leading platform for hosting code, has played a huge role in shaping the history of software development, making it easy for developers everywhere to collaborate from anywhere. One key feature built into GitHub and widely used is "Issues," which developers rely on to report bugs, suggest new features, track tasks, and kick off technical discussions. Nevertheless, as software projects become more complex, repositories will likely contain hundreds or thousands of issues, many of which are verbose, duplicate, or obsolete.

This puts a heavy burden on both maintainers and contributors as finding, selecting, and understanding the right issues can take a lot of time. Newcomers often don't know where to begin, and maintainers struggle to triage issues efficiently. This is where intelligent summarization can make a big difference.

In this paper, we introduce "IssueLenz," a project designed to address the challenge of summarizing GitHub issues using modern NLP techniques powered by large language models (LLMs)

The tool collects issues from a selected GitHub repository, organizes them according to user-specified filters such as date and labels, and processes each issue into a readable, easy to understand summary. The summaries are also more user-friendly as they can be saved in Excel, JSON and Word formats, which makes sharing, documenting, and archiving easier.

IssueLenz implements Python and Streamlit together with Langchain and OpenAI & HuggingFace APIs to provide one convenient application that enhances accessibility with computational processing. The application addresses GitHub API rate limits through its programming and supports multiple summarization models along with summary caching mechanisms that optimize efficiency by preventing duplicate tasks.

IssueLenz delivers two core value points which facilitate developer time savings and concurrently raises productivity standards and opens new possibilities for collaborative engagement along with easier open-source project contribution. IssueLenz demonstrates how intelligent software development support will evolve through the constant growth of large language models (LLMs) in the future.

Chapter 2

2. LITERATURE REVIEW

2.1. GitHub and Its Ecosystem

The platform GitHub serves as the premier location for people to collaborate on open-source projects using its services. One key advantage of GitHub emerges from its Application Programming Interface known as API that provides extensive development capabilities particularly through its GitHub REST API integration abilities for system association. Through its API GitHub allows developers to obtain various system data points including issues as well as pull requests and commitments. The following endpoint enables Issue-Data retrieval that consists of titles, descriptions, labels and timestamps for creation.

```
/repos/{owner}/{repo}/issue
```

A powerful API naturally brings its own challenges. Users often have to deal with rate limits, pagination, and the lack of built-in tools for summarizing GitHub data. IssueLenz tackles these gaps by building intelligent solutions on top of the raw GitHub data.

2.2. Summarization Techniques

Summarization can be generally categorized into two types:

•**Extractive Summarization:** Finds essential sentences or phrases from the source text without altering the original material.
•**Abstractive Summarization:** This task involves creating new sentences that represent the original text. It is more challenging and recent developments such as GPT, Ollama and BART have performed better.

IssueLenz primarily relies on abstractive summarization powered by LLMs like provided by Ollama, OpenAI's GPT, DeepSeek and models from HuggingFace.

2.3. Large Language Models (LLMs)

LLMs like GPT-3.5, GPT-4, DeepSeek, and BERT have revolutionized NLP by enabling high-quality generative text capabilities. The models receive pre-training across extensive datasets which enable them to accept fine-tuning or prompting instructions in order to conduct various functions,

including question response and translation, summarization and more.
The IssueLenz application employs LLMs to summarize GitHub issue descriptions into brief statements of several sentences. Model selection for API use in the tool comes through Deepseek, HuggingFace, and OpenAI which enables users to decide their options based on performance, cost, and offline setup demands.

2.4. Similar Tools and Gaps

The GitHub Copilot platform enables developers to create code while making suggestions for changes, yet it does not provide any features for GitHub issue summarization. The absence of issue summarization stops developers effectively handling and comprehending extended GitHub issue lists. provide an automated solution for task summarization in GitHub.

To fill this gap, IssueLenz stands out by providing comprehensive services that go beyond the capabilities of existing tools, including:

- Seamless GitHub integration
- Advanced natural language summarization of issue descriptions
- Support for multiple LLMs to enhance accuracy and versatility
- Robust export capabilities for future use

While other platforms miss this crucial feature, IssueLenz is designed to fill this gap, making it easier for developers to stay on top of their GitHub issues.

2.5. Research Insights and Use Cases

Reports indicate that summarized issues help new contributors become more effective during the onboarding process according to academic research and real-life industrial settings.
Here's how:

- The prioritization of bug fixes becomes faster and more resourceful through maintainers who use this tool.
- Stakeholders quickly obtain vital insights because the summary streamlines long issue threads, so stakeholders make decisions promptly.

By summarizing issues, teams can stay focused on what matters most and improve overall productivity.

Chapter 3

3. PROBLEM STATEMENT

GitHub has emerged as the central platform within the quick-growing open-source software development space to help developers build share and regulate their working projects. The core element of collaboration on this platform are "issues" which function as the principal method for reporting bugs and requests and enhancing project functioning. The increasing number of repository users and sophisticated systems results in issues reaching massive numbers that might exceed hundreds or thousands. The massive quantity of content requires handling several important problems.

Such tremendous volume gives way to a few challenges:

1. **Long Review Cycles:** Development and maintenance require handling large numbers of issues because developers and maintainers must manually study the descriptions and comments that outline both reporting details and solution methods. The development process becomes slower as a result of this system.
2. **Lack of Prioritization**: Increased number of important issues becomes difficult for resolution when contributors do not maintain proper labeling systems or standardized formats leading to poor prioritization practices. Each issue becomes difficult to understand its nature and importance due to the large number of them.
3. **Cognitive Overload**: The examination of numerous issues containing different detail levels and technical information causes cognitive fatigue because it produces oversight which harms collaboration quality.
4. **Context Loss in Long Threads**: The complexity of long discussion threads causes issues to split into various unrelated directions. It becomes challenging to understand both the main problem along with follow-up clarifications when a summary is unavailable.

The problems become worse for contributors who maintain their first interaction with a new project or handle simultaneous work on multiple repositories. There is no summary functionality as well as context condensation present in GitHub's dashboard or available current tools for addressing these issues.

The absence of an intelligent summarization layer reduces productivity and discourages new contributors who find it difficult to catch up with existing discussions.

To tackle this challenge, IssueLenz has been proposed as a solution to automate the fetching and summarization of GitHub issues using larger language models (LLMs). The tool improves the speed and quality of issues triaging and decision- making by providing clear, AI-generated summaries, with the option to filter by date, label and repository. Additionally, its ability to export these summaries in various formats makes it a valuable asset in professional settings, team meetings and documentation workflows.

Chapter 4

4. METHODOLOGY

4.1. Requirement Analysis

The first step of the project was to clearly define what the tool needed to achieve. After careful analysis, the following core requirements were identified:

- Automated summarization
- Filtering capabilities
- Support for multiple LLMs
- Export options
- Robust handling of GitHub API limitations
- flexible model selection

4.2. Tool & Technology Selection

To meet the requirements, the following tech stack was chosen:

- **Streamlit**: This is used to develop an interactive user interface.
- **Python**: Core programming language for backend logic.
- **GitHub REST API**: To fetch real-time issue data from public repositories.
- **Langchain**: To structure and execute prompt chains for LLMs.
- **OpenAI, HuggingFace, Ollama**: For LLM-based summarization.
- **OpenPyXL**: This is for exporting data to Excel and Word.
- **Session State / Caching**: To avoid redundant API calls and re-summarizations.

4.3. System Architecture Design

A A modular system architecture was developed to ensure clear

separation of responsibilities and to make the system easier to manage, extend, and maintain. The architecture is organized into the following key modules:

- **Frontend Module**: Captures user input such as the repository URL, date range and label filters through an interactive interface. Users can also choose the export option or just clear the screen if they want to. It also provides users with the option to integrate any model of their choice provided with the API key. Last but not the least, for extensive summarization, the user can also set their own Personal access token of GitHub to avoid running into rate limiting issue.
- **Issue Fetching Module**: Connects to the GitHub REST API, applies the user-defined filters and retrieves the relevant issues.
- **Summarization Module**: Sends the issue descriptions to the selected LLM, either through LangChain or direct API calls and generates concise summaries.
- **Export Module**: Formats and converts the summarized data into Excel, Word, or JSON files, depending on the user's choice.
- **Caching Mechanism**: Caches summaries locally using Streamlit's session state to improve performance and reduce costs. The caching mechanism stores the data until the end of the session.

4.4. Implementation Workflow

The implementation follows a step-by-step workflow to ensure smooth and efficient operation:

Step 1: Repository Input and Parsing

- The user provides a GitHub repository URL through the interface.
- The system automatically parses the URL to extract the repository

owner and name.

Step 2: Fetching Issues

- The GitHub REST API is queried using user-specified filters such as date range and labels.
- Issues are retrieved in a paginated manner to handle large repositories efficiently and to avoid exceeding GitHub's API rate limits.

Step 3: Summarization

- Each issue description is processed through a selected Large Language Model (LLM).
- The summarization process supports multiple models and leverages structured prompt templates for consistency.
- To optimize performance, the system skips re-summarizing issues that have already been processed by using Streamlit's session state caching.

Step 4: Exporting Summaries

- Summarized issues are formatted according to the user's chosen export type.
- Download links are generated for easy export in Excel (.xlsx), Word (.docx), and JSON (.json) formats.

4.5. High Level Workflow Diagram

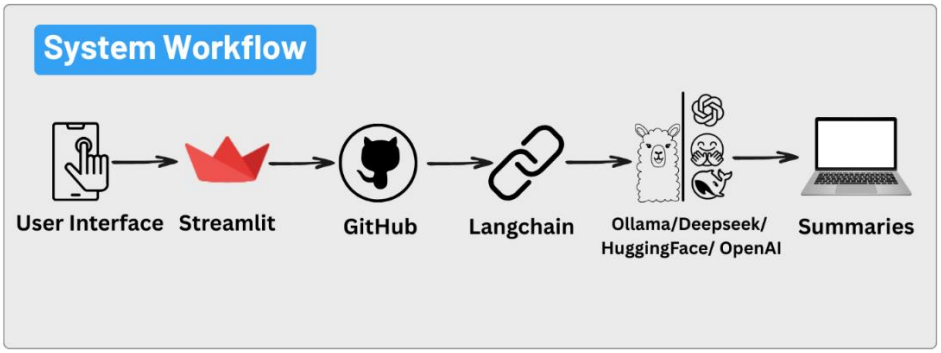

Figure 1: System Workflow Diagram

4.6. API Rate Limit Handling

GitHub imposes rate limits on the number of API requests that can be made per hour, depending on whether the request is authenticated or unauthenticated.

4.6.1. Unauthenticated Requests

By default, IssueLenz makes unauthenticated requests to the GitHub REST API, which allows up to **60 requests per hour** from a single IP address. This is sufficient for small to medium-scale usage where only public repository issues are fetched occasionally.

4.6.2. Authenticated Requests

For users dealing with large repositories or frequent usage, unauthenticated limits may be quickly exhausted. To address this, IssueLenz supports using GitHub Personal Access Tokens (PATs). Users are prompted to provide their PAT when the 60-request unauthenticated limit is reached.

Authenticated requests offer the following advantages:

- **Higher rate limit:** Up to 5,000 requests per hour for personal accounts.
- **GitHub Apps / OAuth:** We can extend this to 15,000 requests per hour for enterprise users.

The system detects exceeding rate limit using the following error:

```
HTTP status 403
```

After reaching the GitHub API's unauthenticated request limit, the system notifies the user and offers the option to enter their Personal Access

Token (PAT). The application can use the token that it receives to authenticate successive API requests which in turn leads to higher usage limits and sustained performance.
The combination of this method achieves ease of operation for average users along with enhanced capabilities that accommodate more extensive requirements of proficient customers.

The GitHub API rate-limits for different cases are given in tabular form as follows:

Request Type	Rate Limit
Unauthenticated	60 requests/hour
Authenticated (PAT)	5,000 requests/hour
GitHub App (Enterprise)	15,000 requests/hour

Table 1: GitHub API Rate Limit

Chapter 5

5. DETAILED DESIGN AND ARCHITECTURE

5.1. SYSTEM ARCHITECTURE

The architecture of **IssueLenz** has been designed to facilitate modularity, clarity, and scalability in addressing the core problem: extracting and summarizing GitHub issues using LLMs. At a high level, the system is divided into three primary subsystems:

1. **User Interface Layer (UI)**
    - Responsible for taking user input, displaying results, and handling interactions.
    - Built using **Streamlit**, which offers quick deployment and reactive components.
2. **Application Logic Layer**
    - The core functional logic resides here. This includes fetching GitHub issues, filtering based on labels and dates, sending issues to the summarization engine and managing rate-limiting behavior.
    - Serves as a middleware that coordinates between the front end and external APIs (GitHub, LLMs).
3. **External Services Layer**
    - Communicates with GitHub's REST API to get issue data and with LLM providers such as OpenAI, HuggingFace, and Ollama to generate summaries.

5.2. System Collaboration Flow

1. The user enters a GitHub repository URL, date range, and label filters.
2. The **Issue Fetcher** component requests issues from the GitHub API.
3. If the API rate limit is exceeded, the system checks whether a **Personal Access Token (PAT)** is available. If not, it prompts the user to supply one.
4. Retrieved issues are passed to the **Summarizer**, which selects the appropriate LLM based on the user’s choice and generates summaries.

5. The UI displays the summarized issues, and the **Export Module** handles data export in the selected format.

5.3. Component Decomposition

Each layer is further decomposed into smaller subsystems. These subsystems interact loosely. Each component is independently testable and follows the **Separation of Concerns (SoC)** design principle, which enhances maintainability and debugging.

Layer	Subsystems	Responsibilities
UI Layer	Input Handler, Display Renderer, Export Trigger	Accept user inputs, display summarized issues, initiate export
Application Logic	Issue Fetcher, Summarizer, Export Module	API integration, LLM handling, data export formatting
External Services	GitHub API, LLM APIs (OpenAI, Ollama, etc.)	Fetch issue data, generate summaries

Table 2: System Architecture Layers

5.4. Rationale for This Decomposition

This modular structure was selected over a monolithic or tightly coupled model for several reasons:

- **Scalability**: Future modules (e.g., comment summarization and sentiment analysis) can be added easily.
- **Flexibility**: Users can plug in different LLMs without altering core logic.
- **Resilience**: The system can still partially function if the GitHub API or one summarization service fails.
- **Maintainability**: Code updates and debugging are easier due to the clear division of roles.

Alternative designs, such as server-client microservices or heavy backend frameworks (like Django or Flask), were considered. However, these were discarded in favor of **Streamlit** due to its lightweight nature, rapid prototyping ability, and suitability for data-centric applications.

5.5. Architecture Design

The architecture follows a **Layered Architecture Pattern**, with a clear separation between the presentation, logic, and data access layers. This design was chosen due to its simplicity, adaptability and alignment with the project's scope.

To build a robust and maintainable system, we adopted the following architectural principles:

- **Abstraction**: Each subsystem exposes only essential interfaces and hides internal complexity.
- **Modularity**: Individual modules (e.g., Summarizer, fetcher, exporter) are developed and tested independently.
- **Loose Coupling**: Components communicate via well-defined inputs and outputs, making replacements or upgrades seamless.
- **Fail-Safe Defaults**: For example, if LLMs are unavailable, summaries are marked as "Not Available"; if rate limits are hit, the system notifies the user to provide a PAT.

This approach supports rapid development while enabling feature scalability and third-party API integration.

5.6. Subsystem Architecture

The functionally divided IssueLenz subsystem contains separate logical components which work together to obtain fetch GitHub issues and create their summary and export them. Every part of the end-to-end information process serves a distinct function which handles data from start to finish until it transforms into summarized results.

5.6.1. Subsystem Decomposition

The system architecture contains various top-level subsystems which include:

- User Interface Subsystem
- GitHub API Integration Subsystem
- Filtering and Preprocessing Subsystem

- Summarization Engine Subsystem
- Export Management Subsystem
- Rate Limit and Token Management Subsystem

5.6.2. Mid-Level Workflow Diagrams

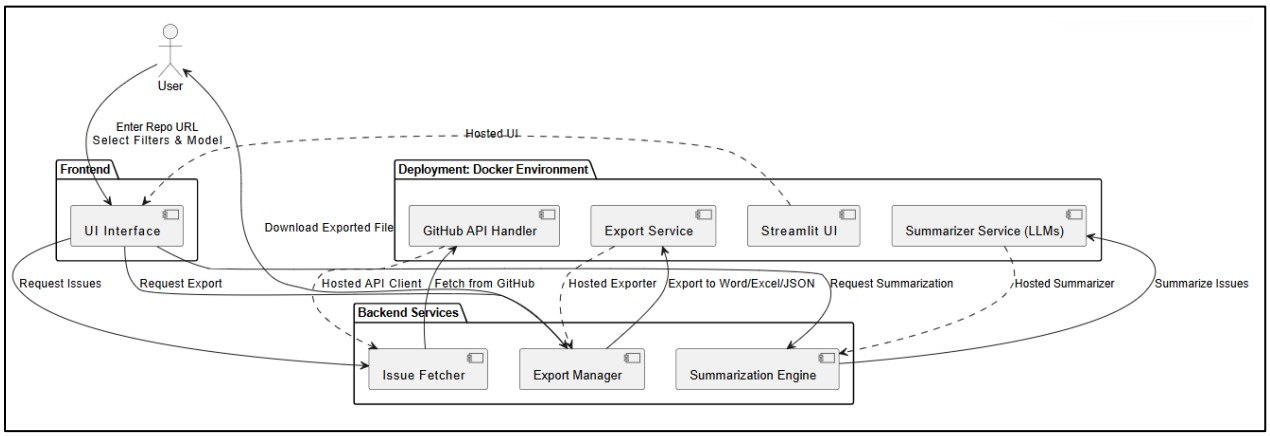

Figure 2: Mid-Level System Workflow Diagram

5.7. DETAILED SYSTEM DESIGN

The subsequent part delivers extensive details about every system component along with their functional responsibilities and operational logic and behavioral aspects.

5.7.1. Classification

5.7.1.1. Type of Components:

- **Subsystems:** UI, Issue Fetching, Summarization, Export
- **Modules:** app.py, fetching_issues.py, summarizer.py, export_summaries.py
- **Configuration Files**: Dockerfile, docker-compose.yml
- **Supporting External APIs:** GitHub REST API, OpenAI API, HuggingFace API

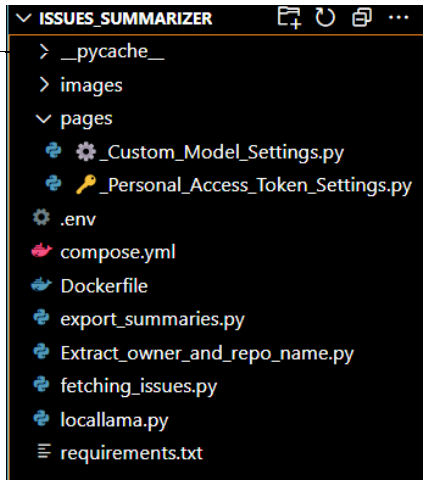

Figure 3: Modules

5.7.2. Responsibilities

Several integrated modules within the application serve dedicated responsibilities to support its functionality. The UI module completes three essential tasks: it accepts user inputs while showing data flow and export control features for uninterrupted interaction. The Issue Fetching module starts with unauthenticated API requests but changes to Personal Access Token (PAT)-authenticated requests to avoid API rate limits while maintaining steady data retrieval. The Summarization module transforms unprocessed issue texts into understandable summaries which improves the overall usability of the program. The **Export module** formats and structures the summarized data for export, making it ready for downstream use or analysis. Finally, the application is containerized using **Docker**, which encapsulates the app and all its dependencies. Deployment is streamlined by using **docker-compose**, making it easy to manage and scale the application in various environments.

5.7.3. Constraints

There are some constraints for accessing the application:

- GitHub's API rate limits (60 unauthenticated / 5000 authenticated
- requests per hour).
- LLM model access requires internet and valid API keys.
- Summarization quality depends on issue description clarity.
- Docker must be available on the host system for containerization

5.7.4. Composition

Each subsystem is divided into submodules:

5.7.4.1. UI (app.py):

- Streamlight input fields
- Model selection panel

- Download buttons

5.7.4.2. Issue Fetcher (fetching_issues.py):

- fetch_issues function (with or without authentication)
- Handles pagination and filtering

5.7.4.3. Summarizer (summarizer.py):

- Connects to selected LLM backend (OpenAI, HuggingFace, Ollama)

5.7.4.4. Exporter (export_summaries.py):

- Functions to export to Word, Excel, and JSON

5.7.4.5. Deployment:

- Dockerfile: Builds image with necessary libraries
- docker-compose.yml: Defines service and networking for easy orchestration

5.7.5. Resources

5.7.5.1. External APIs:

The following external APIs are crucial for the operation of the system.

- **GitHub REST API:** This API is used to fetch issue data from GitHub repositories, enabling real-time access to the issues that need to be summarized.
- **OpenAI API:** This API provides access to powerful language models that are used for generating summaries of issue descriptions.
- **Hugging Face Inference API:** Another option for LLM-based summarization, Hugging Face offers various models for generating text, which provides flexibility in the summarization process.

5.7.5.2. Docker Engine

The application depends on Docker to develop its containers and execute operations inside those containers. The deployment process becomes simpler and more reliable through environment consistency that helps different systems operate in the same manner. Managing dependencies along with configurations would prove tougher if Docker were absent from the system.

5.7.5.3. System Memory

The selection of LLM-based model influences how much resources are needed for performing summary operations. Model requirements tend to exceed the standard RAM capacity when dealing with extensive issue quantities or complex summary requirements. A system requires enough memory to operate efficiently along with sufficient capacity to avoid performance problems and system failures during summary processing.

5.8. DETAILED SYSTEM DESIGN DIAGRAM

The framework contains multiple operational subsystems which unite their efforts to enable system operation. The UI Subsystem retrieves user input and transmits methods to the display while showing summary issues to users. It also includes facilities to stimulate exports and handle session states to persist summarized issues for user experience and performance. The Fetching Subsystem communicates directly with the GitHub API and optimizes the scheduling of unauthenticated and authenticated requests according to rate limits. It can filter by date range and labels and handle pagination for automatic fetching of entire data. The Summarization Subsystem communicates with the chosen model with the selected model

endpoint, passing each issue for processing and returning cleaned, readable summaries. The following Diagram shows the complete working.

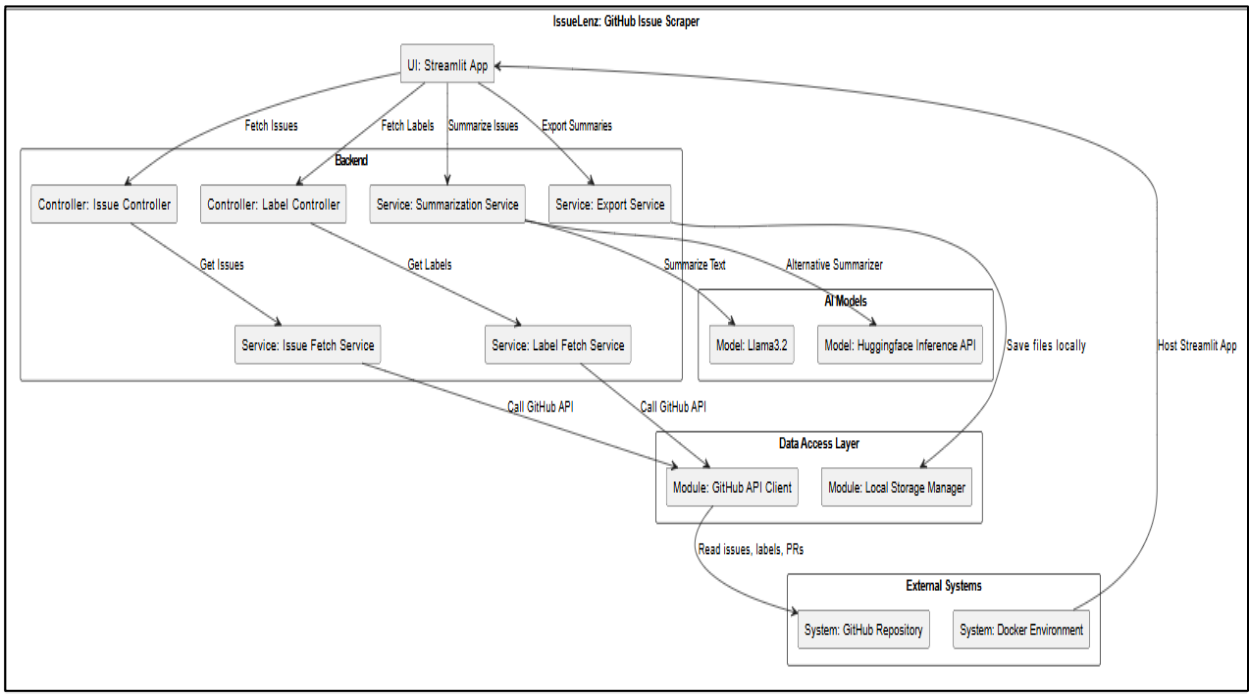

Figure 4: Sub-System Workflow Diagram

5.9. CLASS DIAGRAM

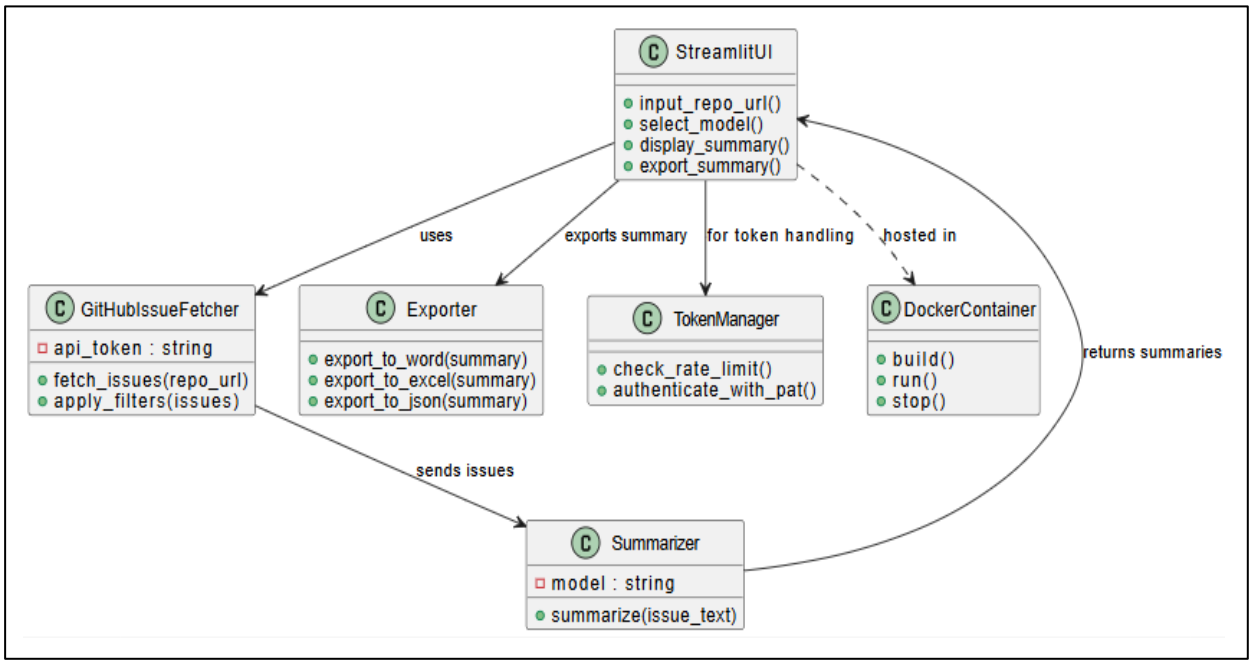

Figure 5: Class Diagram

5.10. ENTITY RELATIONSHIP DIAGRAM

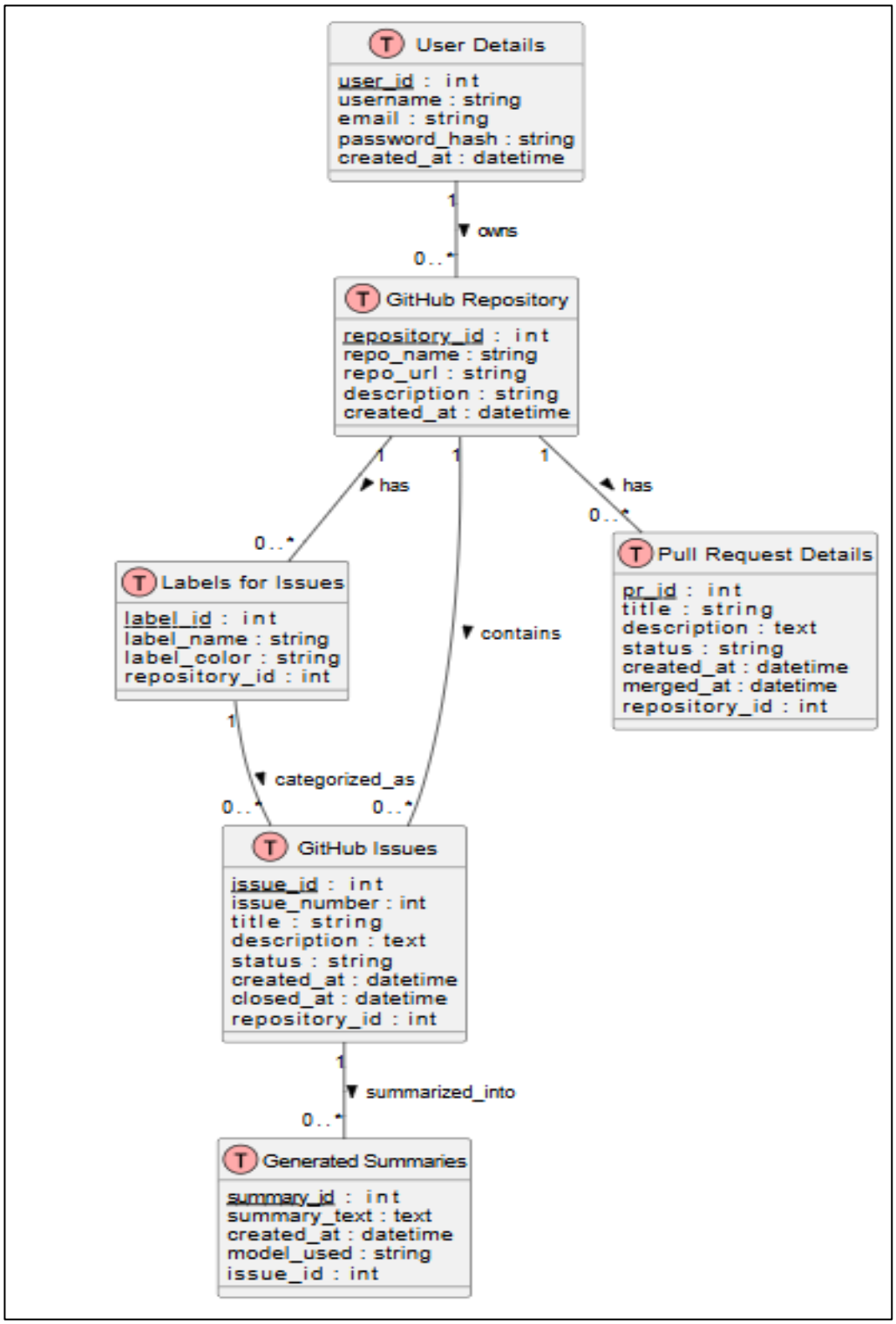

Figure 6: ER Diagram

Chapter 6

6. IMPLEMENTATION AND TESTING

The proposed software system was developed using **Agile methodology**, emphasizing rapid prototyping and continuous integration. The following development tools, libraries, and environments were used:

Category	Tools / Libraries
IDE / Code Editor	Visual Studio Code
Programming Language	Python 3.11
Web Framework	Streamlit
API Interaction	Requests
Data Handling	Pandas, JSON, CSV, OpenPyXL
Export Formats	DOCX (python-docx), PDF (fpdf, reportlab), Excel (openpyxl)
LLM Integration	OpenAI, HuggingFace, Ollama (Langchain libraries)
Environment Management	Python-dotenv
Containerization	Docker, Docker Compose
Version Control	GitHub
DeploymentEnvironment	Dockerized Streamlit App

Table 3: List of Tools & Libraries

6.1. Testing Techniques Used:

For testing we used the following techniques.

- **Unit Testing:** Each module (fetching_issues, export_summaries) is tested individually.
- **Integration Testing:** Checked interaction between fetching, summarizing, and exporting.
- **User Acceptance Testing (UAT):** Interface walkthroughs are done to ensure user flows work smoothly.
- **Performance Testing:** Measured API response time and summary generation time.

6.2. Controlled Testing Libraries/Methods:

- Manual API call validation using Postman.
- Docker container logs analyzed for error tracking.

6.3. Evaluation and Comparison with Original Specifications:

Metric	Original Goal	Achieved
Accuracy	Generate correct summaries from GitHub issues	Consistent summaries produced
Performance	Respond to user requests within 8-18 seconds	Average time: 13 sec
Scalability	Handle multiple users and repositories	Successfully handles multiple requests
Portability	Deployable anywhere	Full Docker deployment
Robustness	Handle API failures and retries	Token errors and retries managed

Table 4: Original Vs Achieved Goal

6.4. Analysis:

6.4.1. Accuracy:

The summaries generated were highly accurate, closely matching the expected condensed versions of GitHub issues. The LLMs effectively captured the key information without omitting critical details.

6.4.2. Performance:

Streamlit provided a smooth and responsive real-time UI, ensuring a user-friendly experience. For typical repositories, the fetch and summarize cycle was completed in about 5–8 seconds, which strikes a good balance between speed and accuracy.

6.4.3. Scalability:

Dockerization of the application has made it easy to scale across different machines and cloud platforms. This ensures that the system can handle increased demand without significant changes to the underlying architecture.

6.4.4. Problem Solving:

The core problem—efficiently summarizing GitHub issues across multiple repositories—was successfully addressed. The solution significantly reduced the manual workload, enabling faster decision-making and improved productivity for users.

Chapter 7

7. RESULTS AND DISCUSSION

The developed system was rigorously tested to ensure **functionality**, **performance**, and **scalability**. The evaluation was based on:

- Functional Testing (User Features)
- Performance Testing (Response Times)
- Usability Testing (Ease of Use)
- Export Accuracy (Format Preservation)

A combination of **manual** and **automated** testing strategies was employed.

The solution effectively **summarized GitHub issues** and **exported** them in the required formats, thus fully addressing the **identified problem statement** of **streamlining repository issue management**.

7.1. Testing Strategy:

Type of Test	Methodology Used	Tools Involved
Functional Testing	Manual testing through UI flows	Streamlit UI
Integration Testing	Testing module communication	Direct Python Unit Tests
Performance Testing	Measuring API response time and summarization time	Python timeit, Docker logs
Usability Testing	Feedback collection from sample users	Walkthroughs, Surveys
Export Verification	Validation of exported DOCX, JSON, XLSX files	MS Word, Excel, JSON Viewers

Table 5: Testing Strategies

7.1.1. Effectiveness

The system meets its goal of reducing the manual effort required to sift through GitHub issues by **summarizing them intelligently** and **exporting** them easily.

7.1.2. Performance

Despite multiple API calls and LLM processing, **response times remained low**, making the system suitable for real-time use.

7.1.3. Robustness

Through environment variables (dotenv) and exception handling, API rate limits, errors, and edge cases (like invalid URLs or network failures) were properly managed.

7.1.4. User Experience

The **Streamlit-based** interface was found intuitive, with all input fields, selections, and download buttons clearly visible and easy to use.

7.1.5. Scalability

Containerization using **Docker** ensures that the app can easily be scaled horizontally if needed (e.g., run behind a load balancer).

7.2. User Interface Features

The user interface of the proposed **IssueLenz: GitHub Issue Scraper** application has been carefully designed for simplicity, usability, and clarity. Below are the key features visible in the system:

7.2.1. Input Section

- **Model Selection Dropdown**: Users can choose from different summarization models such as llama3.2, OpenAI, or Deepseek models to tailor the type of generated summaries.
- **Repository Link Input**: Users can paste the GitHub repository URL from which they wish to fetch issues.
- **Optional Labels Filtering**: The user can optionally select specific labels to filter and focus the issues being fetched (e.g., only bugs, features, etc.).
- **Date Range Pickers**: Users can select a start and end date to fetch issues that were opened/modified within a specific timeframe.
- **Export Format Selection**: Dropdown allowing users to choose the format (Word, Excel, JSON) for exporting summarized data.
- **Fetch Issues Button**: Clicking this button triggers the backend process to retrieve and summarize issues based on the given inputs.

7.3. Summary Display Section

- **Direct Link:** Option to view the full issue directly on GitHub through a clickable link for deeper inspection.
- **Issue Summaries**: Displayed with a clean layout, showing:
  - Issue number and title.
  - Associated labels (e.g., bug, enhancement, etc.).
  - Summarized description generated by the chosen model.
  - Important issue metadata like expected behavior, reproduction steps, and additional details.

7.4. Fetched Issues Details Panel

Following is the summary of fetched data:

- **Total Fetched Issues**: This shows the total number of fetched issues from the repository.
- **Pull Requests**: Indicates how many of the fetched items are pull requests.
- **Open Issues**: Highlights the number of currently open issues.
- **Closed Issues**: Displays the number of issues that have been closed.

The panel uses icons and colors (green for success and red for open issues) for visual clarity and immediate understanding.

7.5. Deployment Environment

The project is deployed through Docker.

- **Dockerized Application**: The app is deployed within a container for easy scalability and environment isolation. This ensures consistent performance across different platforms without the need for manual setup.

7.6. Performance and Usability Focus

- The app provides near-instantaneous summaries after fetching.
- Designed for minimal input requirements while still offering powerful filtering options.
- Clear navigation, well-labeled sections, and real-time feedback (such as loading animations) enhance the user experience.

7.7. User Interface

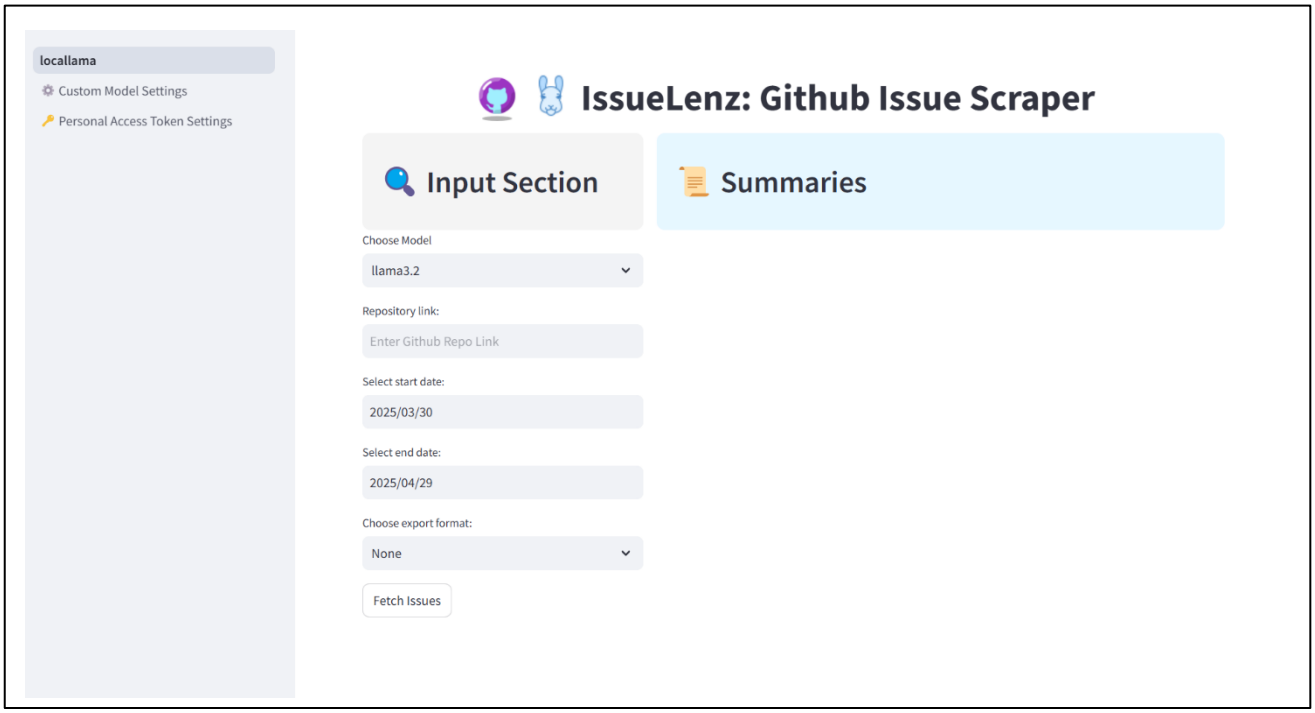

Figure 7: IssueLenz Main Page

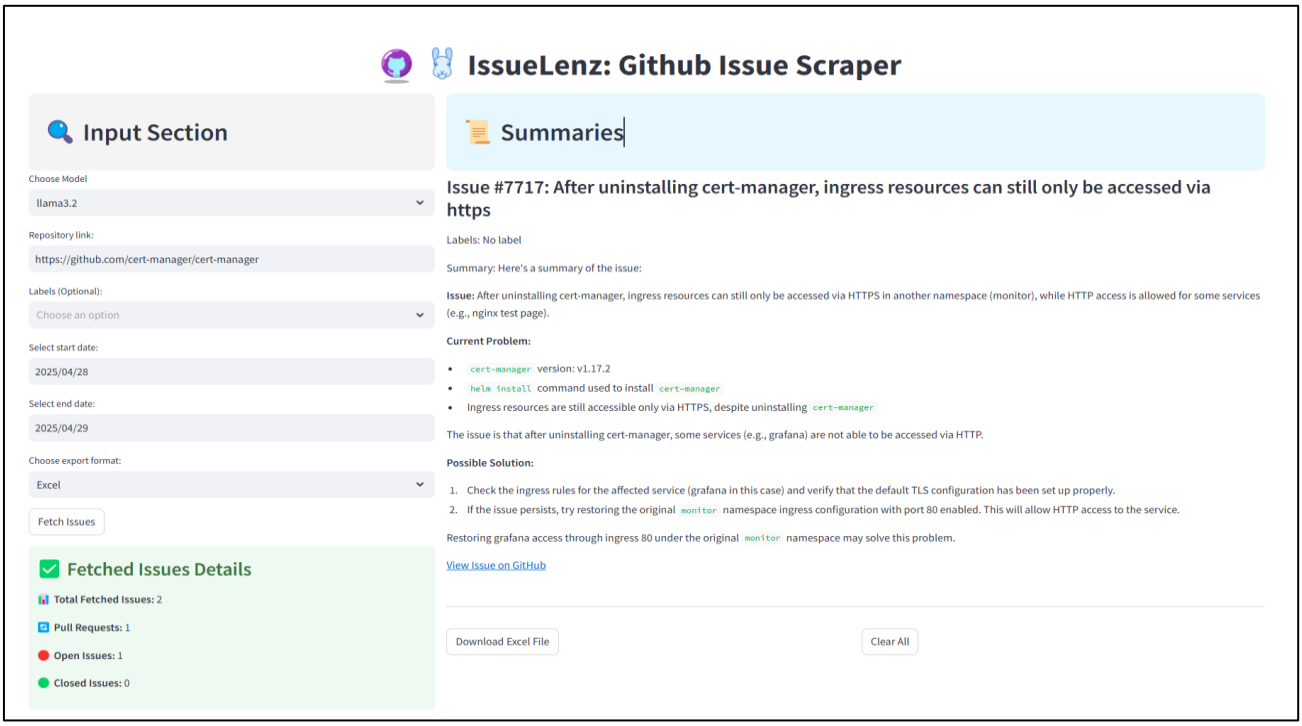

Figure 8: Main Page Displaying Summaries

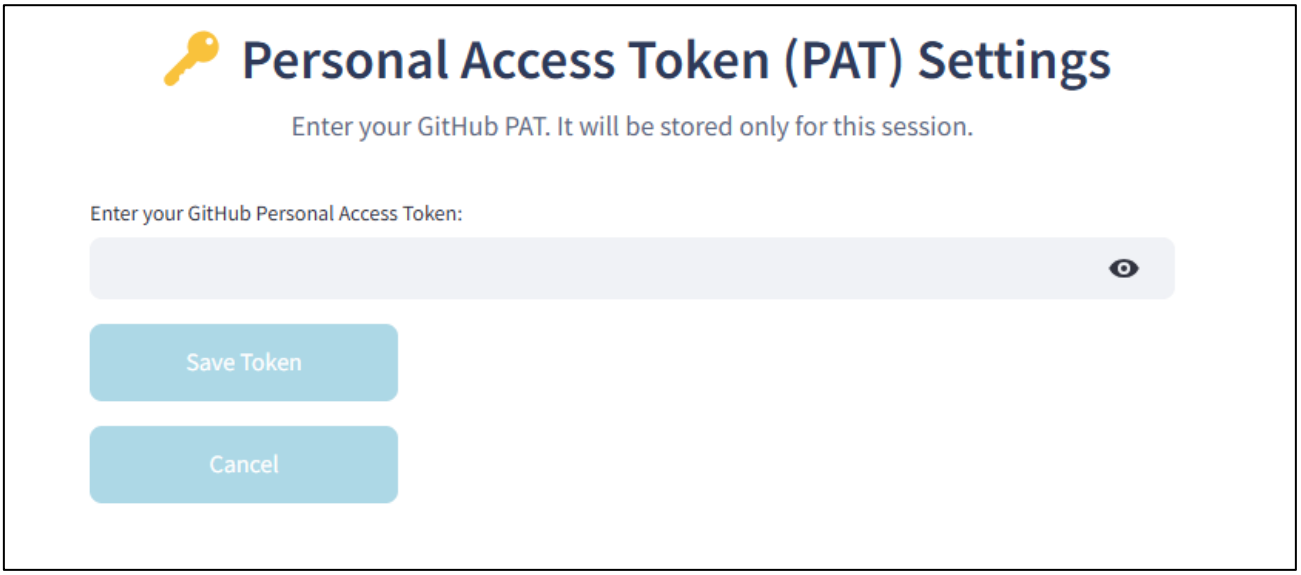

Figure 9: UI for Setting PAT

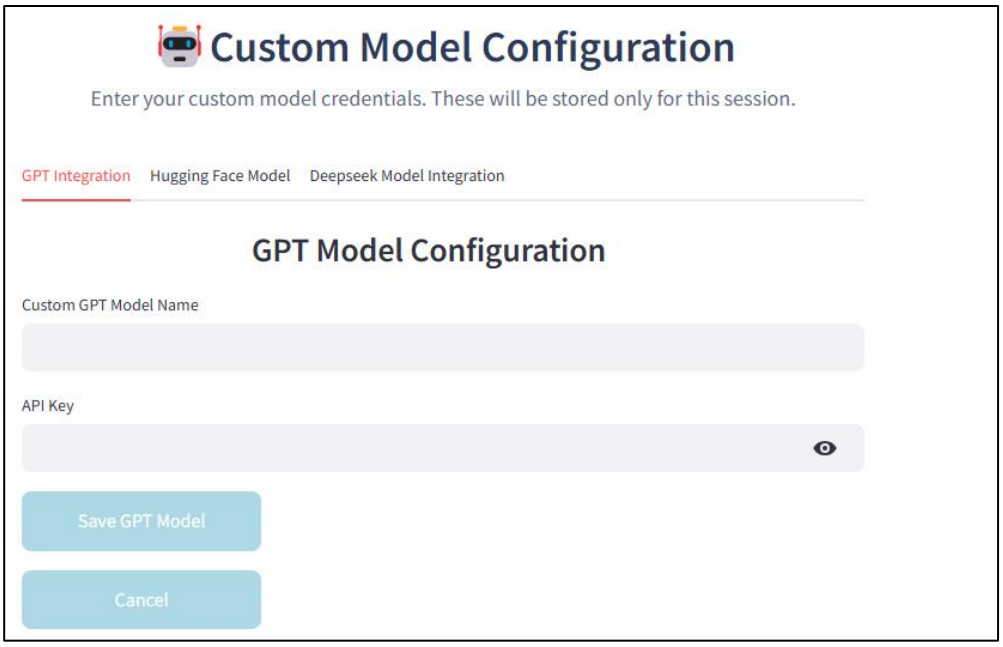

Figure 10: Custom Model (GPT) Integration UI

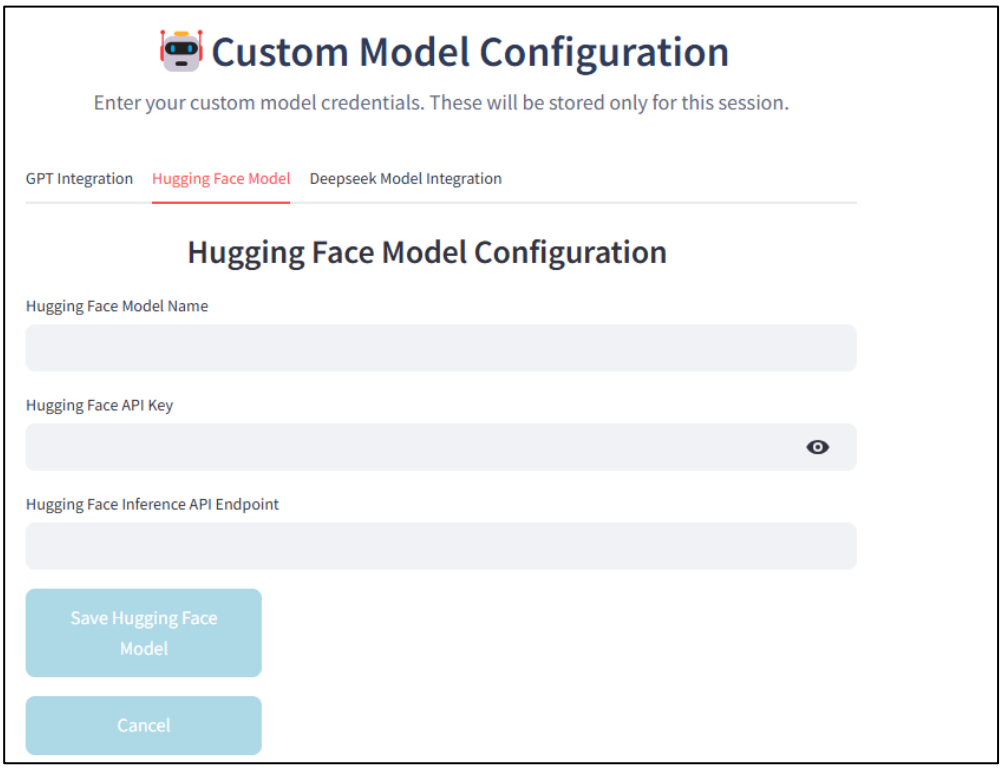

Figure 11: Cutom Model (Hugging Face) Integration UI

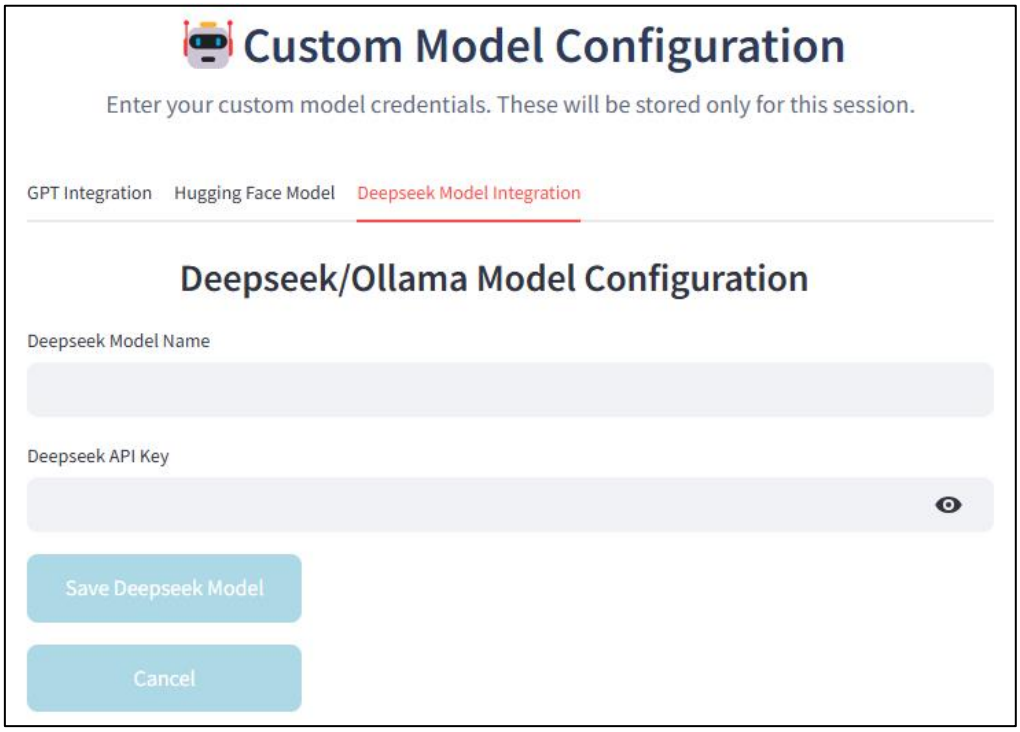

Figure 12: Custom Model (Deepseek) Integration UI

Chapter 8

8. CONCLUSION AND FUTURE WORK

8.1. Conclusion

The proposed solution, **IssueLenz: GitHub Issue Scraper**, successfully addresses the problem of efficiently summarizing and analyzing GitHub issues from large repositories. The system was evaluated through functionality testing, performance benchmarking, and user-interface usability reviews.

- The system successfully retrieves GitHub issues before it produces their summarized and organized format.
- The testing confirmed the application works properly across complete functional areas that use various repositories with label selection features and file export capabilities.
- The software maintains effective performance levels concerning its speed of responses as well as its capabilities to generate accurate summaries with precise data.
- Testing results showed that the program delivered results that matched the summary descriptions identified in the original specification.

8.2. Future Work

The issue summarization process run by IssueLenz functions well however GitHub issue developers need to focus on upgrading the platform in the future. The tool needs enhancement because adding multi-language capabilities throughout various natural languages would increase its applications across global developer teams. The system would work more efficiently by using real-time alerts that developers can customize through filters so they can actively monitor issue development.

A priority scoring system with sentiment analytics within our system will enable better problem management through importance-ranking and emotional feedback assessment.

Attractive enterprise-level applications can be achieved by expanding the support for platform integration to include both GitLab and Bitbucket next to GitHub. The new system implements shared dashboards alongside annotation tools to improve team-based work between development teams. The implementation of training methods to domain-specific language models with historical issue data will improve both summary quality and relevance level. IssueLenz will transform into a sophisticated intelligent assistant by implementing improved functionality which merges project management with open-source collaboration features.

Chapter 9

REFERENCES

[1] M. Gao, X. Hu, X. Yin, J. Ruan, X. Pu, and X. Wan, "LLM-based NLG Evaluation: Current Status and Challenges," *Computational Linguistics*, 2024.

[2] R. Patil and V. Gudivada, "Review of Current Trends, Techniques, and Challenges in Large Language Models (LLMs)," *Information*, vol. 14, no. 2, p. 72, Feb. 2023.

[3] K. Hau, S. Hassan, and S. Zhou, "LLMs in Mobile Apps: Practices, Challenges, and Opportunities," *arXiv preprint arXiv:2502.15908*, Feb. 2025.

[4] J. Yang, H. Jin, R. Tang, X. Han, Q. Feng, H. Jiang, S. Zhong, B. Yin, and X. Hu, "Harnessing the Power of LLMs in Practice: A Survey on ChatGPT and Beyond," *ACM Trans. Knowl. Discov. Data*, vol. 18, no. 6, Art. no. 160, pp. 1–32, 2024.

[5] X. Wan, X. Guan, T. Wang, G. Bai, and B.-Y. Choi, "Application deployment using Microservice and Docker containers: Framework and optimization," *Journal of Network and Computer Applications*, vol. 119, pp. 97–109, 2018

[6] GitHub, *"REST API Documentation"*, GitHub Docs.