

2025 FYPJ Project Documentation

AI Chatbot and Classification

Done by

Aloysius Oh 230928Q

Nicholas Wen 232343X

# Table Of Contents

[Table Of Contents 1](#_rx4lmd1cspl9)

[Project Overview 2](#_ih283f5gv2hb)

[Foreword 2](#_fueeuwie6ch6)

[Aim 2](#_edi8ubcrr0vc)

[Introspection and Roadmap 3](#_bdizrthzzkm3)

[Past Implementation Review 4](#_3j54hmw7xvnx)

[First Batch Workflow 4](#_mbzoc0r05jot)

[First Batch Analysis 4](#_ujqp1l31mw10)

[Second Batch Workflow 5](#_lbk327tvn9kg)

[Second Batch Analysis 5](#_el15pa7uo4bw)

[System Overview 7](#_pm1ooyerjpj)

[Data Processing 7](#_uoxq9kj5rn8t)

[Chat Model 8](#_oh6sd08eibvc)

[File Uploads 9](#_42k50ifk8qn2)

[Classification Model 9](#_6myhm9jho8rf)

[Installation Process 10](#_6bdz0b5fz0b)

[Changes Made 11](#_a8ba9dghwxoe)

[Miscellaneous 11](#_fszkv3cl9vwc)

[Features and Functions 11](#_x44aiko8t4g5)

[Sprint 1: Week 1 to 3 11](#_v91585atgqa)

[Hashing 11](#_635hzkx8ik8n)

[Validation 13](#_q5kk6xjc1li5)

[Fixes 15](#_e75335ai7oqv)

[Storage 16](#_zg32sgz4nc78)

[Refactorisation 17](#_19tsj4pwhz5q)

[Aesthetics 18](#_jdnxnjdkjxgk)

[Sprint 2: Week 4 to 5 20](#_h5bi26ybbaxj)

[Search 20](#_8xagn059hguv)

[Optimisation 21](#_ue980h5qrsu)

[Cross-compatibility 22](#_1fw0gykryl3u)

[Containerization 23](#_no08prg0km9i)

[API Limit 24](#_lw668r97qve5)

[Password change 25](#_osj09t1x9nju)

[Sprint 3: Week 6 to 8 26](#_4axiw5w9tmih)

[Docs 26](#_lx6fpi9dragc)

[Finetune 27](#_540zyf95qwgz)

[Lighten 28](#_7zt3ib30e5m5)

[Visuals 29](#_c1f26d9wkv0c)

[Migrate 30](#_3i2yfsqp6skv)

[Reduce 30](#_azcel19elgqw)

[Codebase Integration: Week 9 to 11 31](#_7tkqkr4zquxm)

[Audio revisited 31](#_2n5kl09dhfur)

[Suggested Improvements 31](#_1rpqdtva3bra)

# Project Overview

## Foreword

The NYP-FYP CNC Chatbot was a chatbot used to help staff identify and use the **correct sensitivity labels in their communications**.

It made use of the Python programming language, along with integrations of Streamlit, Flask, *Pandoc, Tesseract OCR and OpenAI*.

It has been refactored to use Gradio instead of Streamlit and Flask.

## Aim

The aim is to develop an AI-powered chatbot and data classification system customized for the NYP Data Classification framework. This system will assist staff in quickly **addressing CNC-related inquiries and applying the correct security classifications**, thereby minimizing the need for time-consuming manual searches through documents.

## Introspection and Roadmap

The initial two iterations of the project established a solid foundational workflow for the chatbot. However, the functionality remained quite basic: the first version included only a chatbot and file upload feature, while the second iteration introduced user sign-in and data classification capabilities.

A notable limitation in both versions is the lack of persistent memory, meaning that all conversations are lost once the program is terminated. To enhance the overall codebase, I would need to incorporate a series of incremental features aimed at improving the system's usability and operational efficiency.

As for the third batch of students working on this project, the codebase was left in a state which could be significantly improved. Passwords were stored in plain text, OpenAI changed their API definitions which broke the chatbot, data was being persisted everywhere. The code did work, but was not the most pleasant to work with or run as an end user.

In conclusion, the first three batches laid the foundation for a stable backend API. As the fourth batch, we hope to create a beta which the stakeholders can try out and give feedback on. The fifth batch can aim to make the application production ready and deploy it, building on the work of the first four batches.

# 

# Past Implementation Review

## First Batch Workflow

In the initial iteration, the project utilized a selection of libraries such as pdfminer, docx, pptx, and openpyxl to extract data from various file formats. This extracted information was consolidated into a single text file named extracted\_text.txt. The text was then segmented using **LangChain’s recursive chunking function**, followed by **embedding** and the **creation of a Chroma database**.

On the front end, files were uploaded via **Streamlit’s file uploader**, and the back end validated the file type using the filename. Once validated, the content was loaded into extracted\_text.txt, which was subsequently chunked, and the Chroma database was updated. User queries entered through Streamlit’s text input were **sanitized using regex** to remove unwanted characters before being processed by the RAG LLM model.

The query underwent **multi-query transformation** to generate multiple phrasings for broader contextual understanding. It was then passed to a **history-aware prompt** to determine if chat history was required for context. If needed, a standalone question was formulated to ensure clarity without relying on prior conversation. The final query was embedded and compared against the vector database **using a similarity search** to retrieve the top three most relevant vectors. These vectors, along with the answering prompt, were fed into the LLM to generate a response, which was displayed to the user.l

## First Batch Analysis

While the team successfully implemented functions to handle diverse file types like PowerPoint, Excel, Word, and PDFs—chosen due to their alignment with the CNC database—the system had untapped potential. For instance, incorporating image processing could have allowed users to upload screenshots of text rather than requiring them to embed it in a file.

Using a single extracted\_text.txt file posed challenges: content from one file could bleed into another, and metadata sources pointed to the text file rather than the original document. A logic flaw in the file uploader caused repeated uploads, leading to duplicates in extracted\_text.txt and, consequently, the vector store. This duplication risked biasing the model toward the duplicated data.

Additionally, the prototype lacked persistent memory, j=meaning conversations were lost once the program ended. Past questions could not be retrieved, making the system less practical for staff who might need access to previous interactions. Furthermore, the absence of a data classification feature—a key component of NYP’s CNC framework—limited the system’s utility in classifying files based on security categories, an essential aspect of NYP’s data security protocols.

## Second Batch Workflow

The second batch retained much of the first batch’s workflow but introduced notable improvements. Text extraction from PDFs shifted from pdfminer to Google’s Tesseract OCR, while other file types continued to use similar functions, all still consolidating into a single text file. A new error-handling mechanism was added to manage file processing errors, using try blocks with logging.

Data chunking was significantly revised. Instead of relying on LangChain’s recursive chunking, the team manually split text by paragraphs and then into 512-character segments. The text was cleaned to retain only letters and spaces, and empty or duplicate chunks were removed using sets. The Chroma database was initialized at the start of the program using LangChain’s ChromaDB class, with documents added incrementally and organized into collections.

The integration framework transitioned from Flask to FastAPI, and PostgreSQL was introduced to store past conversations and implement an account system (though registration functionality was absent). Svelte was used for the front end, and Docker managed dependencies and program execution. Unlike the first batch’s single-page interface, this version featured multiple pages, including login, chatbot, file upload, and data classification.

File uploads were enhanced with a dynamic folder system that categorized files by type. Extracted content was directly pushed into the vector store instead of being stored in a text file. This logic was reused for data classification, where the first and last 40,000 characters of the text were processed by a classification model adapted from the chatbot. The input underwent multi-query transformation and history-aware prompting to generate a standalone paragraph if needed. Finally, a security category and reasoning were returned.

## Second Batch Analysis

Switching to Google’s OCR for PDF text extraction was innovative but fell prey to common OCR pitfalls, such as misinterpreting certain characters. Surprisingly, despite adopting OCR, the team did not extend its use to process additional file types like emails, which could have been valuable for data classification.

The aggressive cleaning of the dataset raised concerns. For example, instructional texts containing numbered steps (e.g., “go to question 5”) were rendered unusable when numbers were stripped, leaving the model to misinterpret the sequence as a checklist. Manual splicing also risked mangling text, though this issue was mitigated in data classification using an LLM. On the positive side, adding documents to an existing database rather than recreating it improved efficiency.

The duplication issue was resolved by separating file uploads and the chatbot into different pages, enhancing usability. The introduction of pagination provided a cleaner, more intuitive user interface.

# 

# System Overview

## Data Processing

1. The text is extracted using the unstructured open source library, which is a library that uses libmagic to detect the file type using the file’s encodings and after which extracts the text using the appropriate functions and then turned into a langchain document object.
2. The document object is then passed to langchain’s openai metadata tagger where the document is tagged with 10 new metadata labelled as keyword\_n where n is an arbitrary number from 0-9.
3. These keywords is then saved to persistent storage using shelve and called keywords databank
4. The document is then chunked into semantic chunks using the semantic chunker which are chunks that are complete and self-contained without any overlap between each chunk. Remaining logical while reducing redundancies.
5. After which it is passed onto for batching for efficiency
6. Then it is passed into one of two collections based on what was specified in the data processing function, by default it is chat for the chat model, but the other option is classification which is the documents specifically for classification.

## 

## Chat Model

1. The question, username and chat\_id is passed from the front end to the back end
2. The chat model takes in the thread id which is the chat id and pulls the conversation memory from the sqlite3 database.
3. The user query/question goes through the multi query transformation, where the single query is regenerated into multiple versions/phrasing to accommodate for multiple questions that may exist in the query.
4. The now multiple queries will get passed into the chat history prompt, which determines if the query requires the chat history to provide reference to a previous context in the chat session. If the chat history is needed, a new standalone question is generated that can be understood without the chat history.
5. The final query is then embedded and passed into the vector database.
6. The user query is entered into a LLM prompt where it is used to perform text classification/ data labelling where it retrieves the most relevant keywords from the keywords data bank based on the question asked.
7. The keywords is then used as a filter on the vector store retrieval and after which it will perform similarity search and pick a top 5 within filtered documents, to side step certain limitation of the keyword retrieval, the retriever results is bundled with a unfiltered retriever that will perform the exact same similarity search and pick the top 5 within the unfiltered documents and the results are combined using an ensemble retriever.
8. The relevant context (vectors) and the answering prompt are then entered into the LLM for the last time to return an answer to the user.
9. The responses is saved in a folder categorized by the username and thus the username is used to pull the appropriate folder and json file and then, the chat\_id, timestamp, question and answer is saved to the correct json file

## 

## File Uploads

The file uploader object has an option to take in multiple files and will return a list of the files.

1. The list is iterated through, and the files are passed from the front end to the back end and a temporary directory is created
2. The file is saved to the temporary directory.
3. Then it is processed through data processing
4. After which the file type is detected using libmagic through the function detectFileType and the appropriate folder which is categorized by file type is located or created if it doesn’t exist
5. The file is copied over with a new name that indicates the original filename, the username to track who uploaded the file, the timestamp, random characters to ensure it’s unique and the actual file extension.
6. Then the temporary folder is deleted
7. Then show the users how many files failed and how many passed

## Classification Model

1. The same logic as file uploads steps 1 and 2
2. The file contents is extracted using a function from data processing and then passed into the classification model
3. The classification model takes it in and rephrases it into 3 separate rephrasing to extract out the key topics of the text
4. After which the multi query is passed to the vector store and it performs a similarity search and retrieves the top 3 based on the highest similarity score
5. It passes back the answer which contains the security category, sensitivity of the content as well as reasoning to the user
6. The temporary folder is deleted

## Installation Process

| WhatsApp Image 2025-06-09 at 14.11.47.jpeg | The installation process that we inherited has a couple of pertinent issues. Not all of them can be fixed, but we have **simplified the installation process** to handle dependencies like pandoc automatically.  The set up script has been modified to also work on Linux. Both the front-end and back-end can be called at the same time, as the Flask routes were effectively wrapper functions calling the back-end. See (Refactor section) on more. |
| --- | --- |

# 

# Changes Made

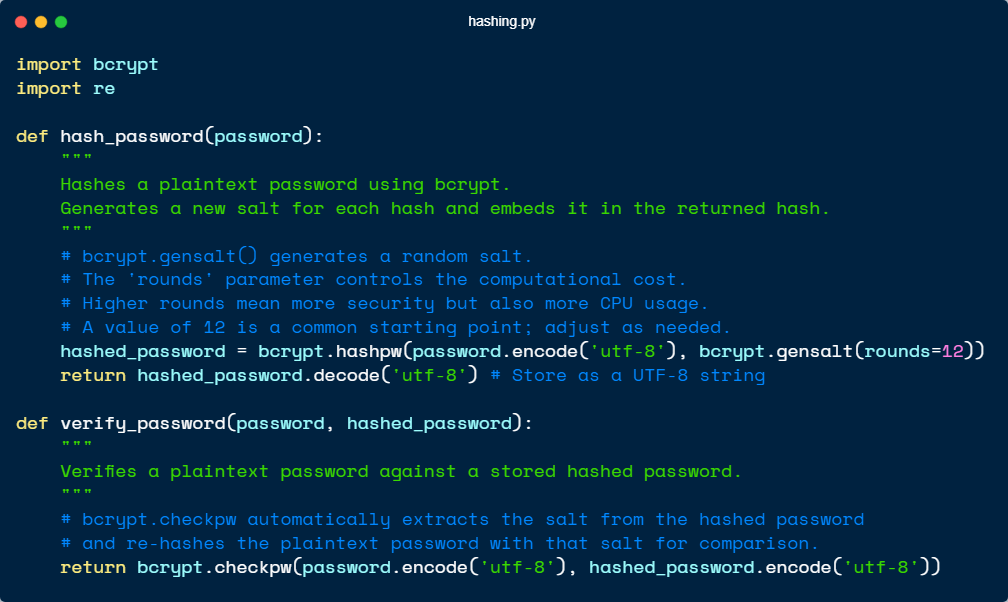
## Miscellaneous

1. Use of standard filetype Python library instead of libmagic. The dependency is now implicitly bundled by another dependency.
2. Change where all application data is saved to ~/.nypai-chatbot and relevant sub-directories. This remains true even after containerisation of the application with Docker.

## Features and Functions

### Sprint 1: Week 1 to 3

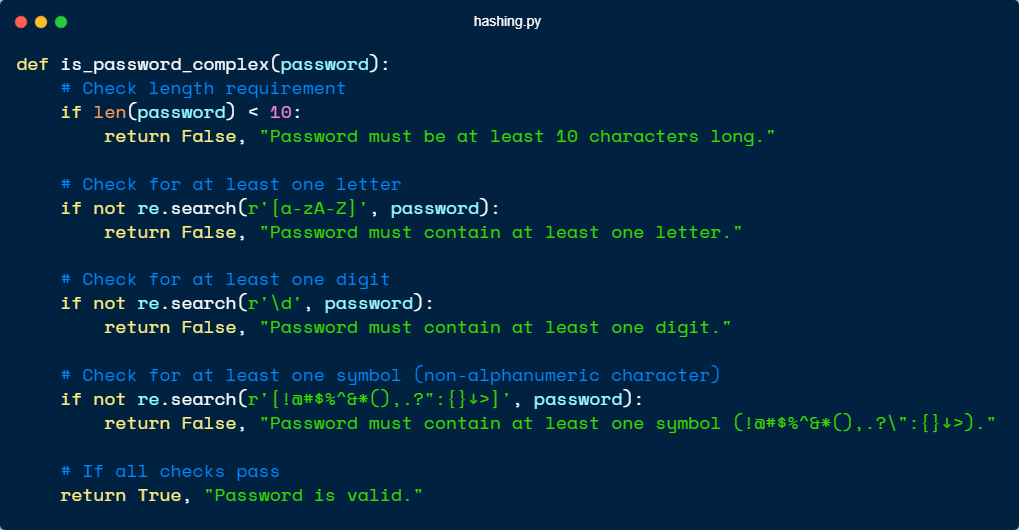
#### Hashing



Password hashing was defined in a hashing.py file, where functions were implemented using the bcrypt library for encryption. These functions were then called to hash user passwords during registration and verify them before login. Hashing is crucial as it ensures that even if the database is compromised, *the actual passwords are not exposed*, thereby maintaining **basic software security standards**.

By employing bcrypt, which is specifically designed for password hashing, the system benefits from its built-in salting mechanism and computationally intensive nature, *making brute-force attacks significantly more difficult*. Furthermore, this approach underscores the importance of secure password management practices, **protecting both users and the integrity of the application** itself.

#### Validation



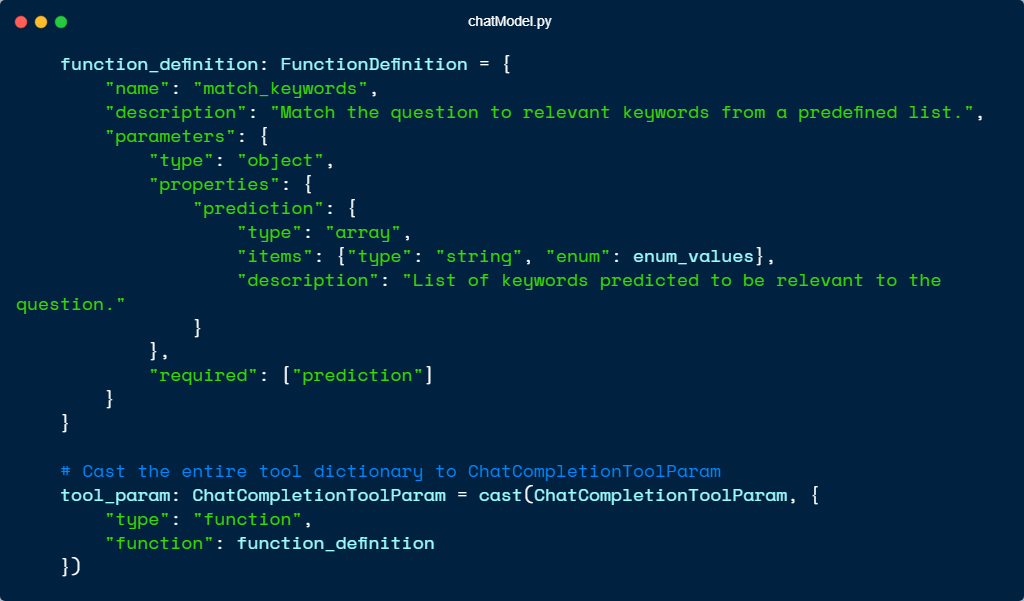
Password complexity checks were defined in a hashing.py file, where functions were implemented using the re library for **regular expression pattern matching**. This function is then called to *check if the password meets the required complexity standards* before proceeding with user registration. By leveraging regular expressions, the system ensures that passwords **adhere to specific criteria**, such as including *a mix of uppercase and lowercase letters, numbers, special characters, and meeting a minimum length requirement*.



These checks are critical for enhancing the security of user accounts by *preventing the use of weak or easily guessable passwords*. Additionally, enforcing password complexity **reduces the risk of unauthorized access** due to common attack vectors like brute-force or dictionary attacks, thereby safeguarding both individual users and the application as a whole.

## 

#### Fixes

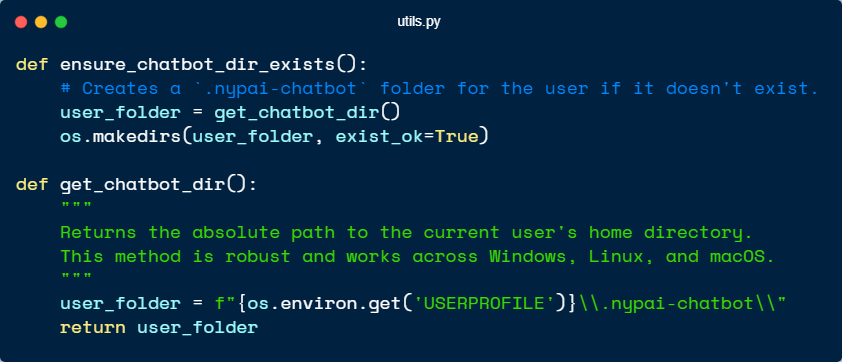


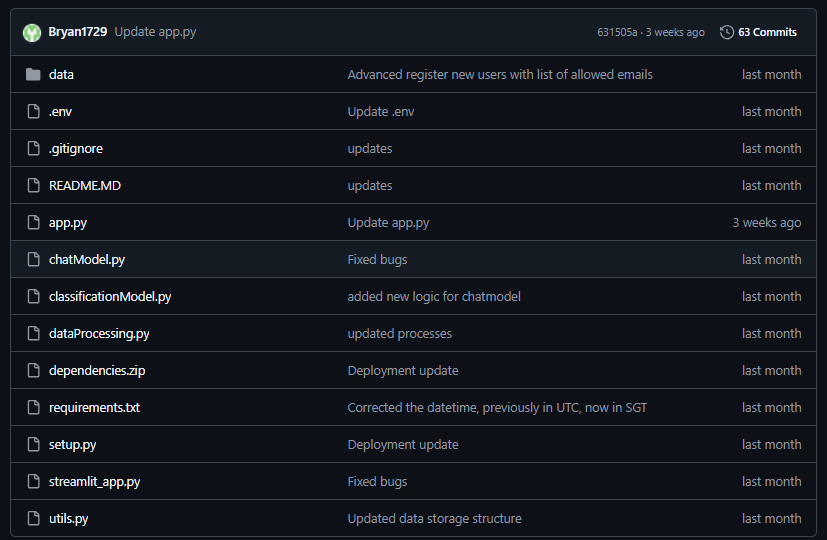
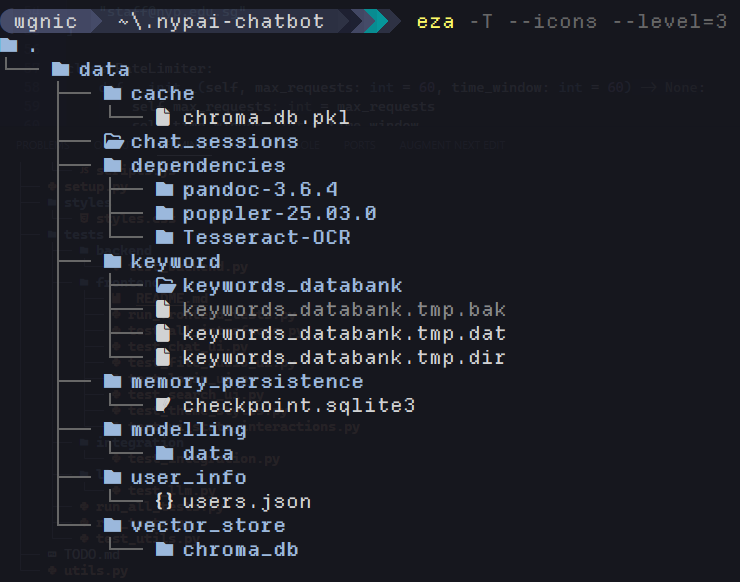
The codebase was updated to align with OpenAI's latest API definitions, replacing the older syntax to **ensure compatibility and access to new features.** This change not only improves the chatbot's functionality by leveraging the most recent advancements in OpenAI's models but also ensures long-term maintainability by adhering to current standards. By adopting the new API, the chatbot benefits from **enhanced performance, better error handling**, and streamlined integration with other services, providing a more robust foundation for future development.

#### 

#### 

#### Storage

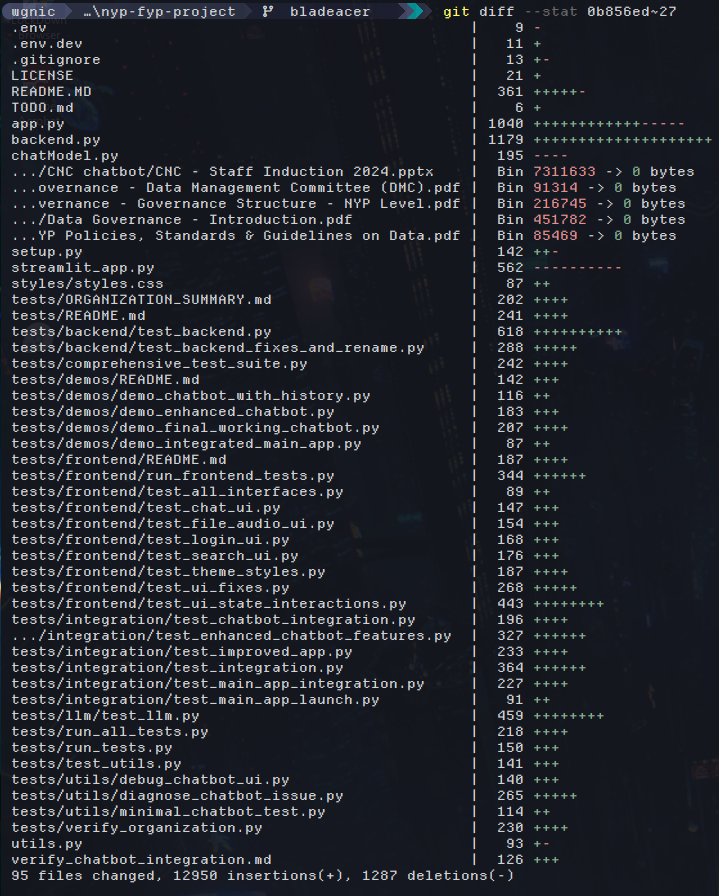




The application's default data storage location has been updated from data/ under the project root to a **custom sub-directory**, ~/.nypai-chatbot, under the user’s home directory to prevent “path pollution”, which refers to the *undesirable accumulation of files* in shared locations like the project root that can clutter the workspace, cause version control conflicts, pose security risks, and reduce portability. After the Docker containerisation, the application data is now stored within the Docker container.

By isolating user-specific data in ~/.nypai-chatbot, the change ensures a cleaner separation of concerns. It also **improves maintainability, enhances scalability**, and *reduces the risk of accidental deletion* while adhering to modern best practices for application data management.

#### Refactorisation



The backend.py file has been refactored to remove its reliance on Flask, significantly reducing the number of dependencies required for the application. By replacing the existing Flask app routes with asynchronous function calls, the codebase becomes more lightweight and efficient, while also improving performance through better utilization of system resources. This change not only enhances the readability and maintainability of the code but also simplifies the logic by eliminating the need for Flask-specific constructs. As a result, the backend is now easier to understand, extend, and debug, making it more robust and adaptable for future development needs. Additionally, the shift to asynchronous functions aligns with modern best practices, ensuring the application is better equipped to handle concurrent requests and scale effectively in production environments.

#### 

#### Aesthetics

| vivaldi 20 June 2025 147.png | Refactoring the UI to leverage **Gradio instead of Streamlit** has brought about several key improvements, particularly in terms of *performance and user experience*.  One of the most notable benefits is Gradio's **superior support for asynchronous operations**, which allows the application to handle concurrent tasks more efficiently, leading to a *smoother and more responsive interface*. |
| --- | --- |

| vivaldi 20 June 2025 148.png | The updated interface also features a dropdown which lets users select from their past chats, allowing them to rename and search through them. |
| --- | --- |

# 

| vivaldi 20 June 2025 146.png | Additionally, the Langchain compilation process, which was previously a bottleneck, has seen dramatic improvements, reducing the app’s startup time from approximately **30 seconds to under 20 seconds**.  This optimization not only *enhances developer productivity* by minimizing wait times but also **improves the end-user experience** by enabling faster access to the application.  Overall, the shift to Gradio streamlines both development and runtime processes, making the application more **efficient and scalable** while maintaining an *intuitive and interactive user interface*. |
| --- | --- |

### Sprint 2: Week 4 to 5

#### Search

| vivaldi 20 June 2025 148.png | The search function is designed to help users efficiently retrieve specific conversations from their chat history. It begins by retrieving the loaded chat history from memory, ensuring that all past interactions are accessible for searching in real time.  Once the chat history is retrieved, the function parses and reads through each message to prepare it for querying. The search feature **leverages fuzzy finding via the difflib library**, which allows for flexible matching. This means users can search using approximate keywords or phrases, and the system will intelligently identify the most relevant results.  By combining precise data retrieval with robust fuzzy matching, the search functionality **ensures that users can quickly locate the information they need, even if they don’t remember the exact wording of past messages.** This approach not only improves usability but also makes the chatbot more intuitive and responsive to user needs. |
| --- | --- |

#### 

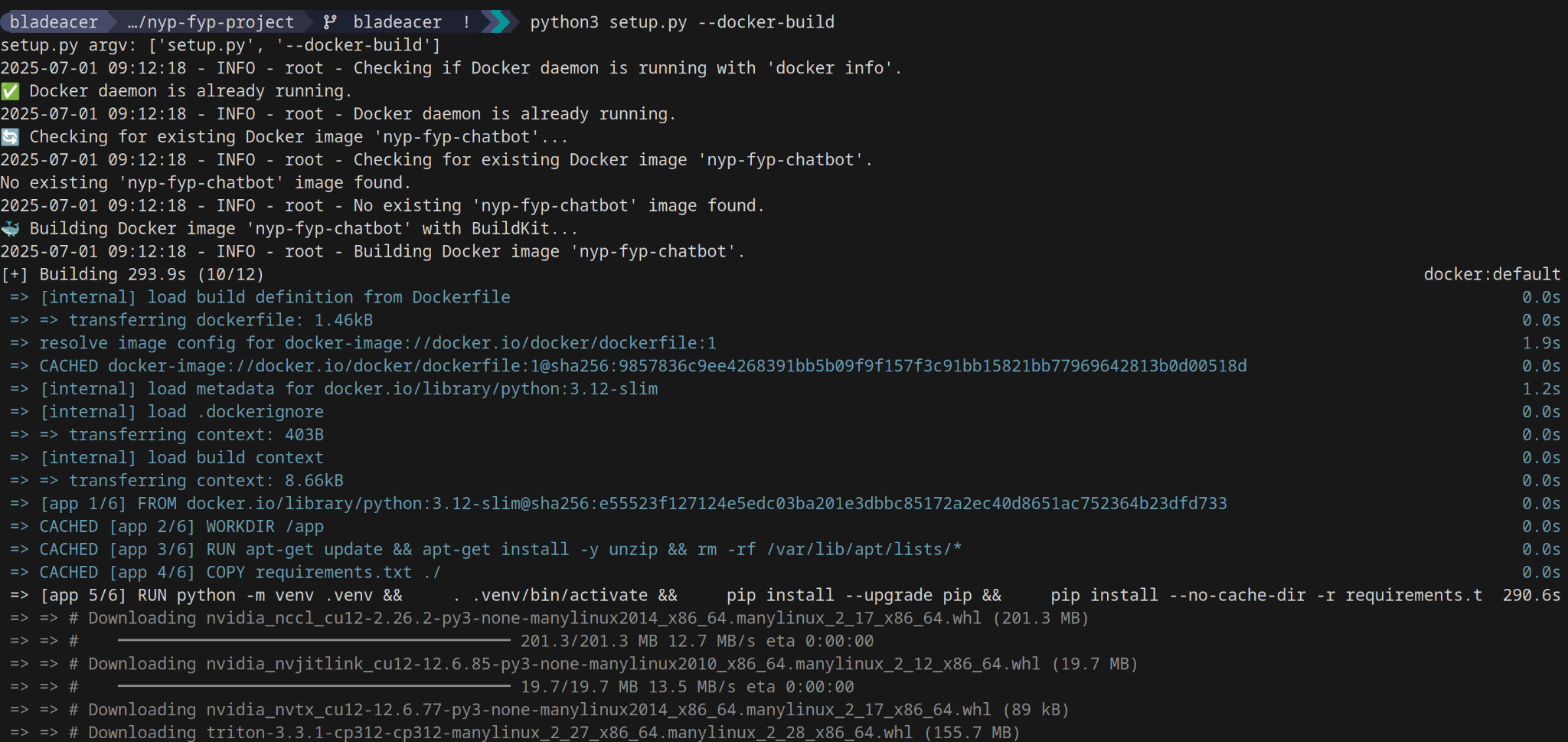
#### Optimisation

| carbon (2).png  alacritty 24 June 2025 14d.png | Further optimizations have been implemented in the Large Language Model (LLM) codebase, with a **specific focus on enhancing data processing efficiency.** These improvements are anticipated to **reduce application startup time**, yielding a more responsive and streamlined user experience.  While the performance gains achieved during the previous sprint’s refactor were relatively modest, the current set of enhancements has been strategically **designed to deliver more substantial and measurable improvements.**  By refining critical areas of the codebase, these changes not only aim to optimize overall system performance but also provide clearer and more quantifiable metrics for evaluating the impact of these modifications. This approach **ensures that future iterations can build upon a more robust and well-documented foundation**, further supporting sustained advancements in efficiency and scalability. |
| --- | --- |

#### Cross-compatibility

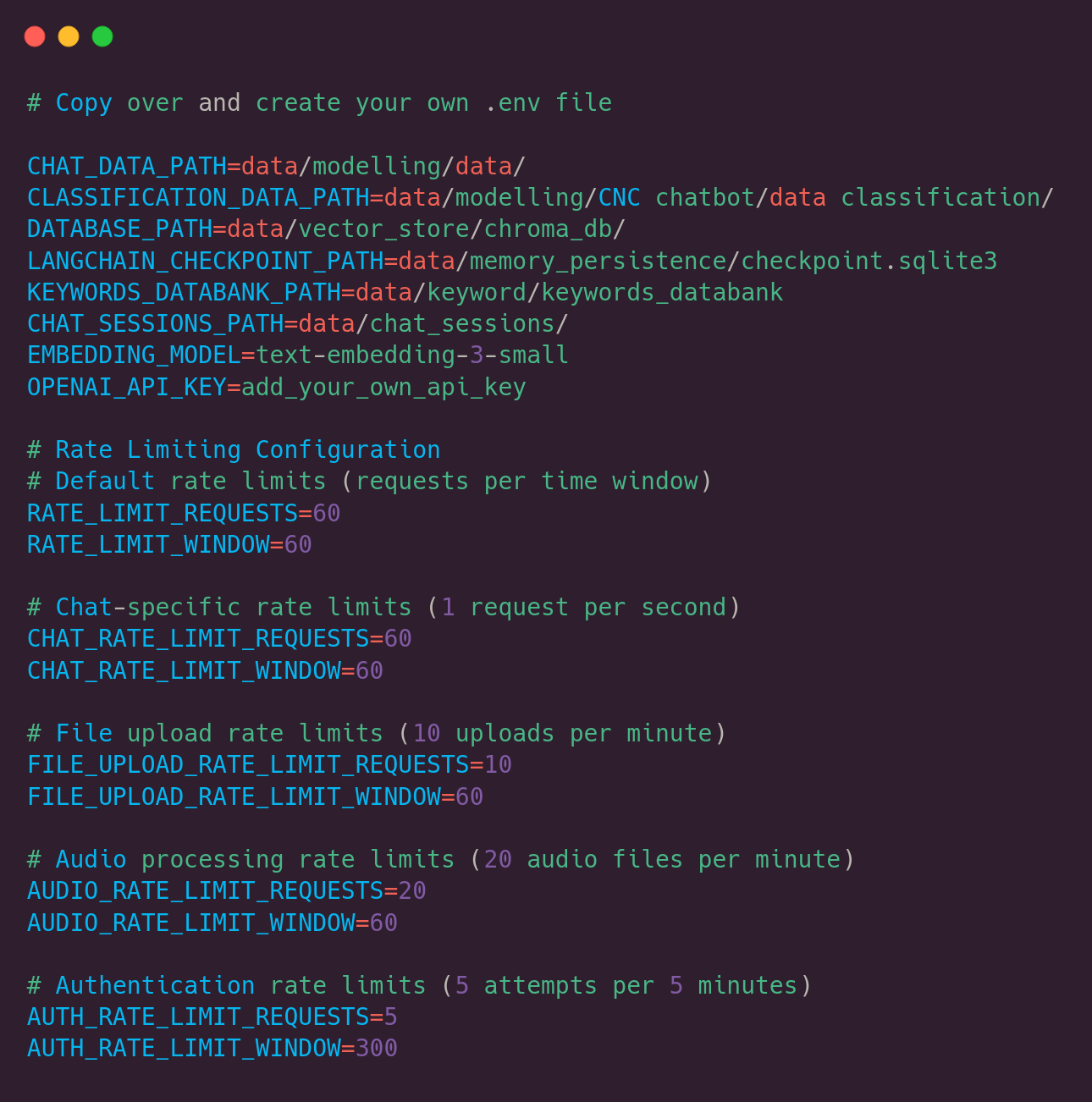
| Screenshot from 2025-06-26 11-19-22.png | In the prior batch's project, the chatbot was limited to Windows, restricting its use across other platforms. To address this, **the code has been updated to support Linux compatibility.**  Key components were refactored to handle platform-specific dependencies and file system differences, ensuring smooth operation on Linux distributions.  **Cross-platform testing confirmed consistent behavior across both Windows and Linux environments.** This improvement broadens the chatbot’s accessibility, aligns with best practices for platform independence, and enables deployment in a wider range of scenarios, including cloud and containerized setups. |
| --- | --- |

#### Containerization



The codebase has been enhanced to install its dependencies and execute **within a Docker container**, which **significantly boosts reproducibility** and ensures consistent behavior across different operating systems. By encapsulating the application and its dependencies in a container, this approach eliminates environment-specific issues, making it easier to develop, test, and deploy the application consistently. This shift not only streamlines the development process but also marks a crucial step toward making the application production-ready, as it simplifies deployment and enhances reliability in various environments.

#### API Limit



Backend functions and API calls to OpenAI have been updated to include an explicit rate limit, which can be configured via the .env file. This addition ensures that API calls are made in a controlled and measured manner, preventing excessive usage that could lead to unexpectedly high costs. By allowing developers to set and adjust the rate limit according to their needs, this feature provides greater control over resource consumption and budget management. Additionally, it promotes responsible usage of the OpenAI API, reducing the risk of hitting rate limits imposed by the service itself and ensuring the application remains cost-effective and sustainable in production environments.

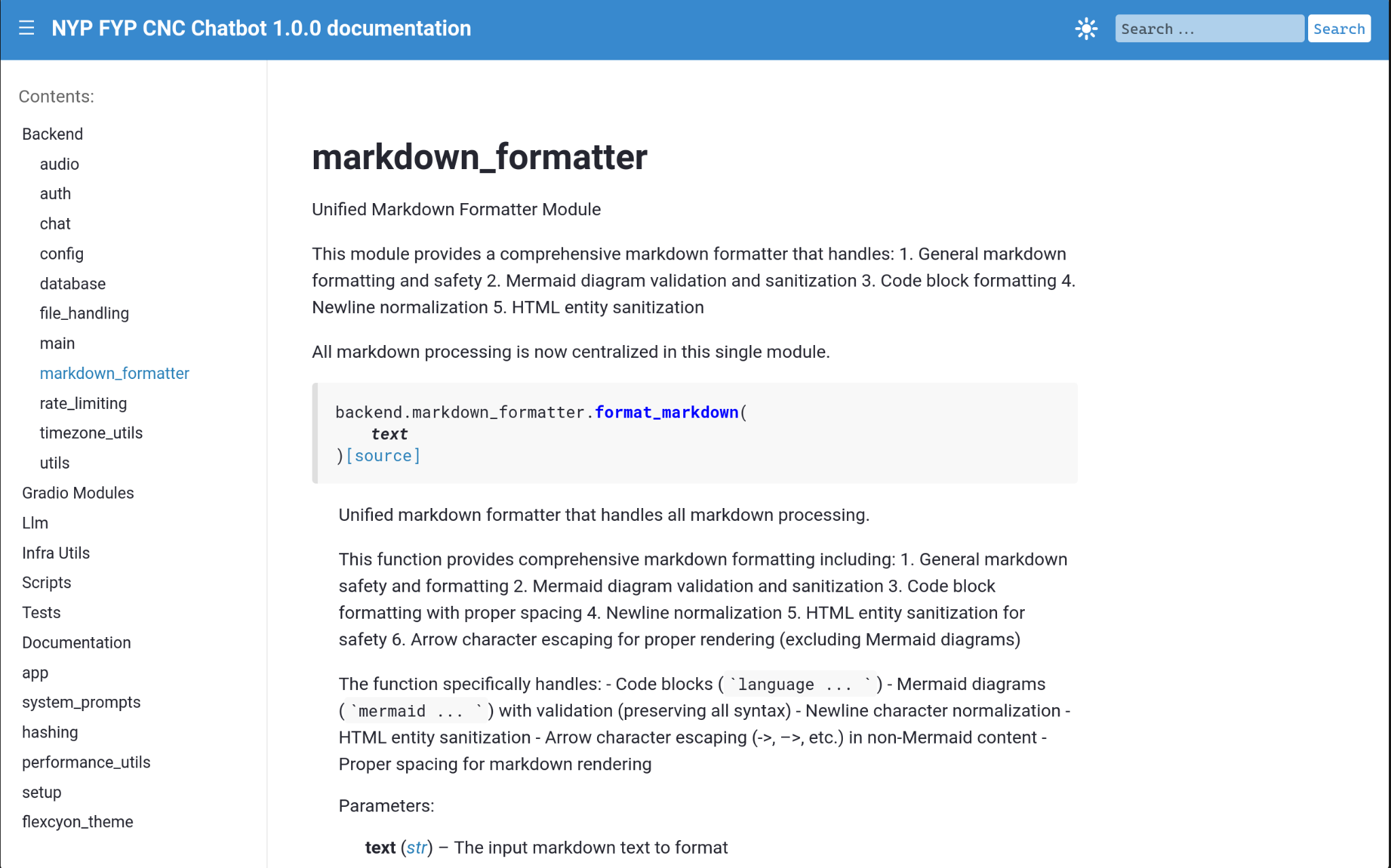
#### Password change

| Screenshot from 2025-07-02 14-32-15.png | The application has been enhanced with a change password functionality, addressing the realistic and essential need for users to update their credentials securely.  This addition significantly improves the robustness of the application by providing users with greater control over their account security and ensuring compliance with best practices for credential management.  By incorporating this feature, the application not only meets user expectations but also strengthens its overall security posture, making it more reliable and user-friendly. Furthermore, this functionality reinforces trust in the application, as users can now easily adapt to changing security requirements or respond to potential compromises of their accounts. |
| --- | --- |

# 

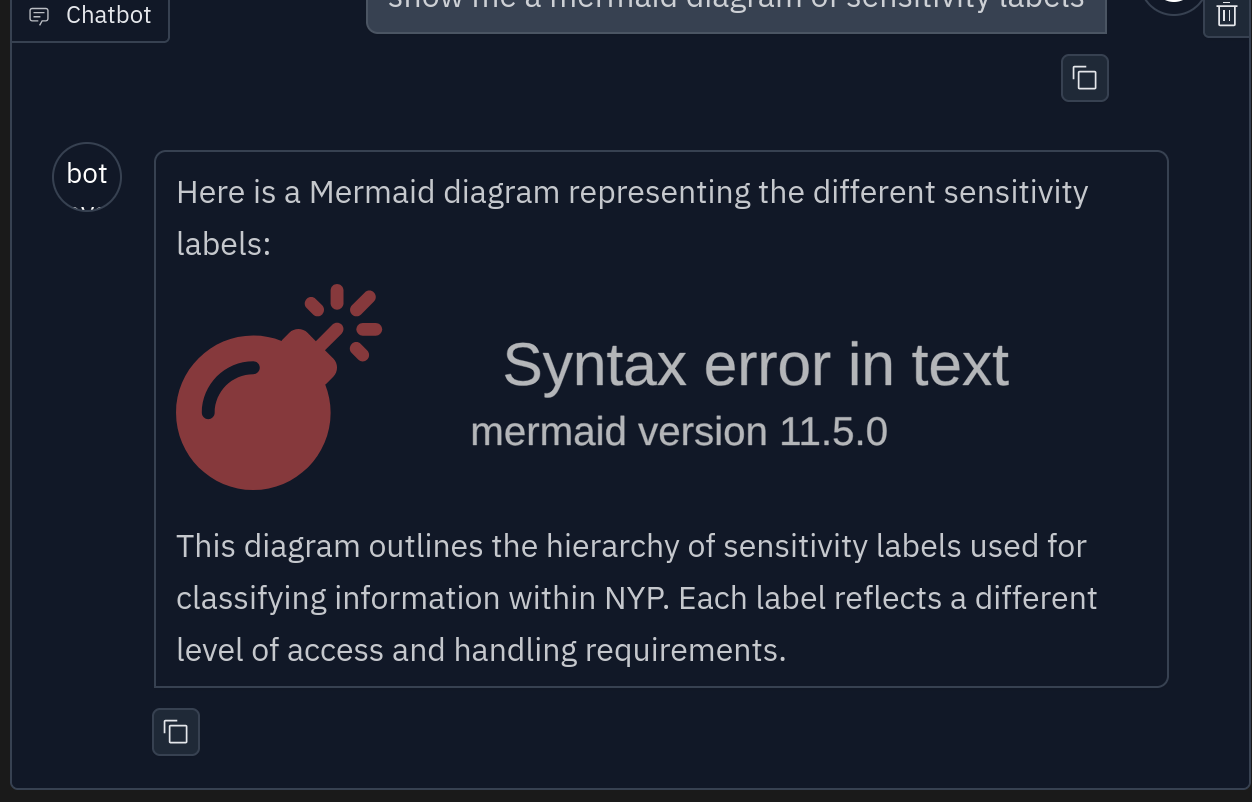
### Sprint 3: Week 6 to 8

#### Docs



Sphinx, a powerful documentation generator, was utilized to automatically create comprehensive software documentation pages within a Docker container. This approach ensures that the documentation remains consistent with the codebase's docstrings, thereby enhancing the long-term maintainability of the project. By leveraging this setup, the generated documentation can be effortlessly configured for deployment to platforms such as GitHub Pages or Read the Docs, enabling seamless access and distribution. The integration of Sphinx within a Dockerized environment not only streamlines the documentation process but also reinforces best practices in software development and project management, making it particularly suitable for collaborative academic and professional settings.

#### Finetune



A markdown formatter was implemented to address and prevent syntax errors in Mermaid diagrams, which are commonly caused by improper handling of special characters. This involved systematically escaping certain characters that could interfere with the rendering process while ensuring that others were appropriately included to preserve the intended functionality of the diagrams. By refining the formatting logic, the solution ensures that Mermaid diagrams are rendered accurately and consistently, improving the overall reliability of the documentation.

In addition to this, optimizations were made to the file classification process by reducing the maximum number of characters and keywords analyzed during the initial scanning phase. This adjustment significantly improves performance, particularly when processing large files or complex directory structures, without compromising the accuracy of the classification results. These enhancements not only streamline the workflow but also contribute to a more efficient and robust system for generating and organizing documentation.

#### Lighten

|  | Switching to Alpine Linux has enabled us to significantly reduce Docker image size and build times, thanks to its minimalistic design and small footprint. As a lightweight Linux distribution, Alpine eliminates unnecessary bloat, resulting in faster builds and more efficient deployments.  Additionally, we implemented further optimizations, such as leveraging uv to run pip install commands in parallel, which accelerates dependency installation and enhances overall efficiency. |
| --- | --- |

#### 

#### Visuals

#### 

By leveraging [Hyperfine](https://github.com/sharkdp/hyperfine)[[1]](#footnote-0), we established a robust benchmarking process to evaluate and export detailed performance metrics for our application, enabling us to pinpoint potential backend bottlenecks and optimize critical workflows. This involved volume mounting the Docker container to the host filesystem with Docker Compose.

#### 

#### Migrate

#### Screenshot from 2025-07-18 12-15-21.png

In addition to tracking application performance, we systematically recorded Docker build times, storing this data in both SQLite and JSON formats for comprehensive analysis and easy integration into reporting tools.

By leveraging database migration helpers to efficiently transfer application data from JSON to SQLite, the process not only minimizes manual intervention and associated errors but also ensures robust database integrity. This approach takes full advantage of SQLite's ACID compliance, guaranteeing that transactions are processed reliably and consistently across iterations. As a result, the migration enhances data durability and accuracy, providing a seamless transition while maintaining the application's performance and scalability.

#### Reduce

Audio pre-processing is used because it significantly enhances transcription accuracy by refining the quality of audio inputs before they are transcribed. Techniques like volume normalization are applied to balance audio levels, addressing inconsistencies caused by uneven recording conditions.

These pre-processing methods ensure that the audio is cleaner and more distinct, reducing issues caused by ambient noise or low-quality recordings. By isolating and amplifying the speaker’s voice, pre-processing creates optimal input for transcription systems, enabling them to deliver more accurate and reliable results.

### Codebase Integration: Week 9 to 11

The last weeks were heavily focused on finishing up and polishing features in the codebase. We had to ensure that our code was properly integrated and working seamlessly together. There were minor issues with the markdown formatter for Mermaid diagrams returned by the chatbot. We had to ensure that the Markdown codeblock syntax was preserved. We also had to rewire up certain interfaces to their correct backend APIs, as during the major codebase refactors some interfaces were affected. We have fixed their issues.

#### Audio revisited

During integration, we had the idea of making use of ffmpeg to trim audio to a maximum of 30 seconds, and apply normalisation to at least 6 decibels. This ensures that we do not overload the OpenAI Whisper API with excessive bandwidth, incurring huge costs for the one provisioning the API key.

This also allows us to standardise the audio format that gets sent to the API as .wav, meaning users can input other audio formats and the transcription will convert to the correct file format before sending the audio object to OpenAI. There is also validation so that non-audio files are rejected.

# 

# Suggested Improvements

1. Bundling application as an executable file (Gradio has a Progressive Web App flag)
2. Refactoring the code to work with self-hosted ollama (via server set up by the school, likely to involve Docker and nginx or Caddy)
   1. Further step would be fitting that and the backend functions to a dedicated HTTP service that can be called
   2. Self-hosted server means one is only limited by server hardware and not API pricing, model training and using custom NYP specific data is also much easier
   3. Only self-hosting an alternative OpenAI might be tricky
3. Using an enterprise database solution like MySQL, MS SQL, Postgres

1. Peter, D. (2023). hyperfine (Version 1.16.1) [Computer software]. <https://github.com/sharkdp/hyperfine> [↑](#footnote-ref-0)