College of Computing and Data Science

NANYANG TECHNOLOGICAL UNIVERSITY

SC4052 CLOUD COMPUTING

Project Report (Topic 3: Conversational Chatbot-as-a-Service and Low-Code-No-Code Principle)

Choh Lit Han Owen
(U2221094K)

Table of contents

1. Introduction

In recent years, conversational Artificial Intelligence (AI) has significantly transformed our world, with breakthroughs in Large Language Models (LLMs) such as GPT-4 and Gemini reshaping how we interact with technology. Chatbots using LLMs, such as ChatGPT, have become useful tools for tasks such as brainstorming ideas, coding, and summarizing text [1]. Cloud computing services such as AWS enable the development of Chatbot-as-a-Service (CaaS) applications by providing a platform that developers can easily deploy their products [2]. Additionally, Low-Code/No-Code (LCNC) design principles are on the rise as intuitive graphical interfaces help increase accessibility for users to build applications without requiring underlying programming skills [3]. These principles provide the benefit of allowing people with little to no technical knowledge to develop the applications they need [4].

Despite the widespread availability and popularity of chatbot services, there may be obstacles that hinder their adoption. These include the advanced technical expertise required to develop LLMs or integrate with Application Programming Interfaces (APIs) provided by external companies. This lack of expertise creates a barrier for business users and non-technical individuals in fine-tuning chatbot personalities and contexts to suit their needs. Instead, they may need to communicate these customizations and feedback to developers, which likely results in longer feedback cycles and slower iterations.

This project aims to bridge this gap by developing a web-based chatbot platform, named SimpleChat, that allows users to create and customize their own chatbots based on LCNC principles. By reducing the need for direct programming effort and offering a user-friendly interface, the project aims to benefit users of different skill levels who require a chatbot solution in their workflow. Furthermore, this project can also serve to help organisations prototype their chatbot integration. By serving as a practical demonstration of how chatbots can be adapted into various applications, the chatbot can be fine-tuned before organizations decide to commit to a full-scale deployment.

2. Project Objectives and Scope

Project Objectives

The objectives of this project are as follows:

- **Design** a **web-based chatbot platform** that allows users to create and customize their own chatbots.
- **Enable** users to define chatbot behavior and context using natural language instructions.
- **Develop** a user-friendly interface that simplifies chatbot creation.
- **Implement** a sharing feature, allowing users to generate a link for others to interact with their chatbot.

Project Scope

This project will focus on developing a web-based chatbot creation tool with the following features:

- **User Authentication:** Users can create an account and log in to manage their chatbots. However, only a standard login with password will be included as part of the project, other solutions such as multi-factor authentication will not be included.
- **Chatbot Customization:** Users will only be able to define their chatbot's name, description, behavior, and provide background knowledge through text input. Users may also choose to upload a document for the chatbot to reference.
- **AI Model Integration:** The chatbot will generate responses using external Google Gemini APIs. Self hosted models such as those available through Ollama [5] will not be included.
- **Chatbot Sharing:** A unique generated link will allow others to use the chatbot without requiring an account.
- **Conversing with Chatbot:** Users can converse with customised chatbots without an account. However, the conversations will not be persisted once users leave the page.
- **SSL Encryption:** Certificates used for SSL encryption in order to use HTTPS will be self signed and managed using Caddy [6].
- **Deployment:** The project will be deployed on a single EC2 instance without any custom domain. This is also part of the reason behind using self signed certificates for SSL encryption.

The project will be developed using React for the frontend, Golang for the backend and SQLite for data storage. AI processing will rely on third-party APIs, which reduces development complexity while ensuring scalability and efficiency. Moreover, by leveraging external APIs, the project avoids the need to own or provision specialized hardware (such as high-performance GPUs) required for efficient AI inference [7], making the solution more cost-effective and accessible at this scale.

3. Solution Design

3.1 System Architecture

The system follows a typical client-server architecture, where the frontend (React) interacts with the backend (Golang) over a network to provide services to users [8]. The backend handles the logic of the web application such as user authentication, chatbot configuration and interactions while chatbot responses are handled by external AI APIs such as Google's AI services [9].

The figure below illustrates the deployment of the application with Caddy.

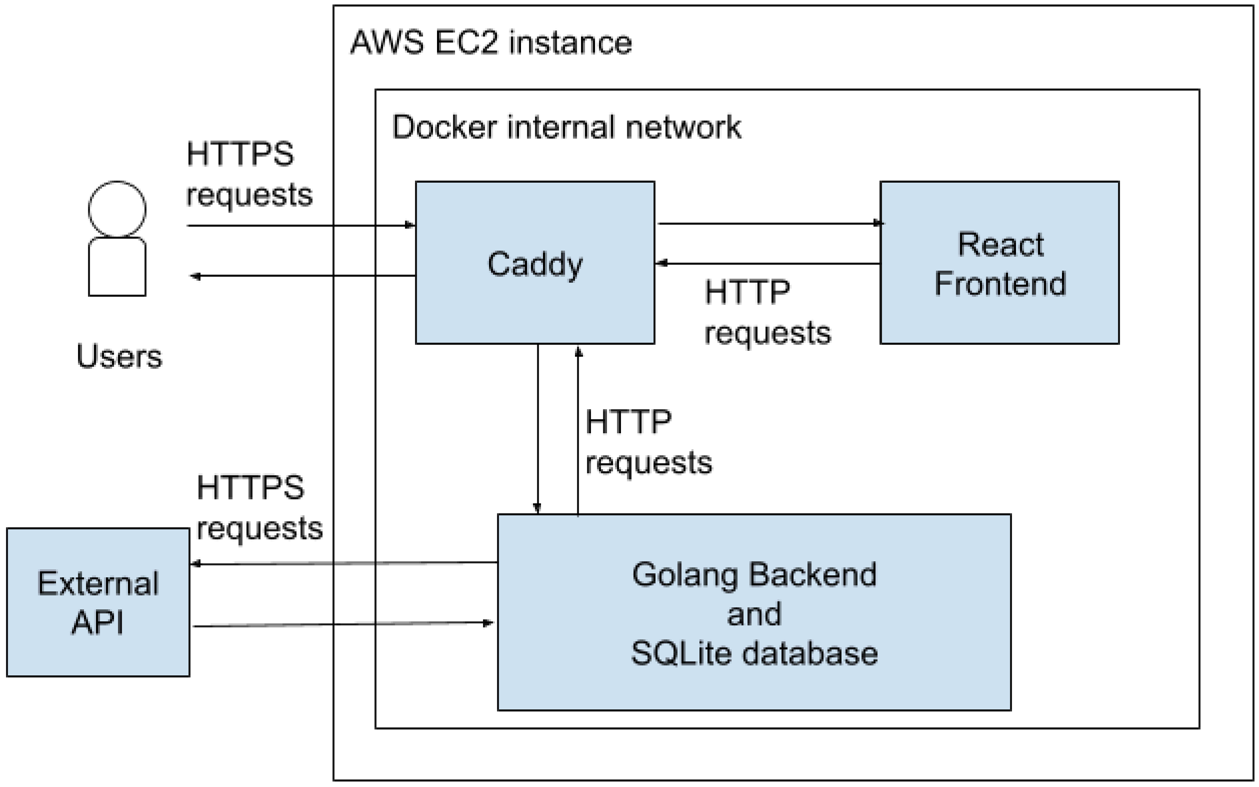


Figure 1: Overall system architecture

The key components of the system include:

- **Frontend (React + TypeScript):** Provides an interactive UI for users to create and manage chatbots.
- **Backend (Golang):** Handles business logic, API requests, database operations, and integration with AI services.
- **Database (SQLite):** Stores user profiles, chatbot configurations, and conversation history.
- **AI Processing (Third-party APIs):** Generates chatbot responses using an external AI model.

Additionally, the backend follows a monolithic architecture, where all functionalities are contained within a single service [10]. This approach simplifies development and deployment but could be modularised into a microservices architecture in the future if necessary.

3.2 Features

- **Text based customization**: Users can adjust chatbot personality, tone, and conversation flow through text inputs on a visual interface.
- **File Upload Support**: Allows users to upload files for chatbot context enrichment.
- **API Integration**: Supports external API calls for additional functionalities.
- **Secure and Scalable Deployment**: Uses HTTPS for secure communication and containerization for scalability.

3.3 Security and Scalability

- **Passwords and Authentication:**

There are many methods of verifying users that are based on the classic paradigm of “something you know”, “something you have” and “something you are” [11]. Examples include passwords, authenticator applications such as Microsoft Authenticator [12], biometrics authentication [11] and federated authentication [13].

Authenticator applications work by generating time-based one-time passwords (TOTPs) [14]. TOTPs work by generating a One-Time-Password (OTP) at specific time intervals [15]. These TOTP algorithms are able to generate temporary and secure passwords [16].

Another solution explored is federated authentication. Federated authentication is a system where users can access multiple applications or services using a single set of login credentials, typically managed by an external identity provider [17]. In this project, it would be similar to the “Login with Google” prompt that may appear as an option when trying to login or register for websites. However, this is not included in the project as it would introduce complexity, requiring additional infrastructure for securely managing authentication tokens. These tokens must be protected against unauthorized access, regularly refreshed, and correctly validated to prevent security risks such as token leakage or replay attacks. Moreover, users may not be inclined to login with their personal account just to try out a prototype.

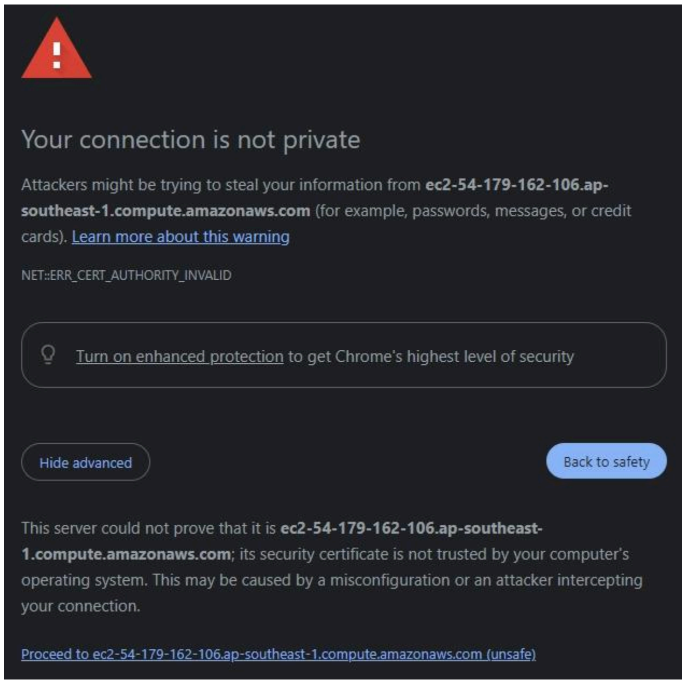
A promising solution is Keycloak, which is an open-source identity and access management (IAM) solution [18]. Keycloak supports federated authentication and user management among other things for securing applications without requiring custom authentication logic [18]. However, it is not used in the project as it would introduce additional configuration overhead and maintenance complexity beyond the scope of this implementation. Moreover, the additional authentication services mentioned earlier are not required for the scope of this project. Lastly, given the constrained infrastructure requirements for this project, there are concerns over the resource consumption that an additional Keycloak service will entail.

Therefore, for the purposes of this project, passwords will be used for authentication given the simplicity and scope of the project. To enhance security, these passwords are hashed and salted before storage, mitigating the risk of unauthorized access in case of a data breach.

- **SSL Encryption**:

SSL encryption, also known as Hypertext transfer protocol secure (HTTPS), is the secure version of HTTP [19]. This is implemented as part of the project to ensure security of the data that passes between the user's browser and the server so that sensitive information such as passwords are not exposed while being sent to the server.

SSL encryption is implemented using self-signed certificates which are managed by Caddy for secure HTTPS connections [6]. This ensures data transmitted between the client and server remains encrypted, mitigating risks of interception and unauthorized access. However, self-signed certificates are not inherently trusted by browsers which only trust Certificate Authority (CA) by default [20]. This may cause security warnings when accessing the application as shown in the figure below.



Your connection is not private

Attackers might be trying to steal your information from **ec2-54-179-162-106.ap-southeast-1.compute.amazonaws.com** (for example, passwords, messages, or credit cards). Learn more about this warning

NET::ERR_CERT_AUTHORITY_INVALID

Turn on enhanced protection to get Chrome's highest level of security

This server could not prove that it is **ec2-54-179-162-106.ap-southeast-1.compute.amazonaws.com**; its security certificate is not trusted by your computer's operating system. This may be caused by a misconfiguration or an attacker intercepting your connection.

Proceed to ec2-54-179-162-106.ap-southeast-1.compute.amazonaws.com (unsafe)

Figure 2: "Your connection is not private" error on google chrome

- **Secrets Management**:

In 2024, there are 39 million secrets that are exposed in Github [21]. As secrets are critical and can easily to lead to security breaches [21], this project has taken some steps to ensure that they are stored and managed securely.

API keys and sensitive data are securely managed using environment variables and Docker secrets [22], minimizing exposure risks. Additionally, secrets are also excluded from the version control system git using a gitignore file [23], while also being kept separate from the files meant for system configuration. This approach ensures credentials are not hardcoded in source code and reduces the likelihood of accidental leaks.

- **Containerization**:

Docker enables seamless deployment of the project and allows for scaling up the project if required [24]. By encapsulating the application within containers, dependencies and environment consistency are maintained across different deployment environments [24]. Furthermore, using Docker Compose simplifies the orchestration of multiple services, such as the backend, frontend, and reverse proxy (Caddy) [24]. However, containerization also introduces additional considerations, such as ensuring proper network configurations between services and management of secrets to avoid leaking them in the built images.

3.4 Data Flow and Interaction

The following is the user flow that the web application is designed around.

1. **User logs in** → Backend verifies credentials.
2. **User creates chatbot** → Sends chatbot configuration to backend.
3. **Creator tests the custom chatbot** → Request is sent to backend, then forwarded to AI API.
4. **External API processes request** → Response is sent back to backend and displayed in the frontend.
5. **Custom chatbot is then shared to the intended audience** → Others can access it via a generated link.

3.5 Breakdown of key project components

3.5.1 Frontend (React)

The frontend is responsible for user interaction and provides the LCNC interface for chatbot creation. It includes:

- **User Authentication:** Users can log in to create and manage chatbots.
- **Chatbot Configuration:** Users define chatbot behavior, personality, and context.
- **Chat Interface:** Enables users to interact with the chatbot.
- **Sharing Mechanism:** Generating and navigating to the link to share the chatbot with others.

React was chosen for its following advantages in building interactive web applications [25]:

- **Component-based architecture**: making UI development modular.
- **Virtual DOM**: ensuring fast rendering and smooth chatbot interactions.
- **State management features**: simplifying parts of the frontend development.

3.5.2 Backend (Golang)

The backend serves as the core of the system, supporting the frontend and handling:

- User Authentication and Session Management
- Chatbot Storage and Configuration Management
- API Endpoints for Frontend Requests
- Integration with External AI APIs for Response Generation

Golang was chosen due to its advantages in backend development [26]:

- **High performance**: Offering fast execution for backend services.
- **Efficient concurrency**: Uses goroutines to handle multiple chatbot requests simultaneously.
- **Strong standard library**: Simplifies API and database interactions without requiring many external libraries.

3.5.3 Database (SQLite)

The database is responsible for storing persistent data, including:

- **User Accounts** for authentication and chatbot ownership
- **Chatbot Configurations** such as prompt settings and context information

SQLite was selected because of its advantages below [27]:

- **Lightweight and self-contained**: Reduces the complexity of managing a separate database server, making deployment simpler.
- **Requires minimal setup**: Making it easy to integrate into the project.
- **Provides fast read performance**: Suitable for chatbot configurations and small-scale user data.

There may be a potential limitation as SQLite locks the database while data is being written [28]. However, SQLite also has a feature that allows for concurrent non-conflicting writes to be executed [28] which is applicable for the project. However, since SQLite has fast performance [29], this is unlikely to be an issue especially since the expected user load is low for this project.

3.5.4 AI Processing (External APIs)

Instead of hosting AI models, the system uses external AI APIs for response generation. Using external AI services removes the need to own or provision expensive GPU-based infrastructure.

Additionally, cloud-based AI providers ensure continuous model improvements, keeping the chatbot up-to-date without requiring additional development effort.

Google’s AI services were chosen due to their comparable performance to other AI models [30] and the availability of a free tier [31], making them a cost-effective solution for this project. The project will make use of the “gemini-2.0-flash-thinking-exp-01-21” model as it has low latency, multimodal input support and a 1 million token context window [32]. This choice helps users to iterate through fine tuning of their chatbots, allows for the support of file uploads for context, and the large context window allows for edge use cases such as enquiring on a large codebase.

3.5.5 Deployment with reverse proxy (Caddy)

Aside from using Docker containers, Caddy is used as the reverse proxy for the chatbot service. It manages incoming requests, routes traffic to the backend services, and provides automatic HTTPS configuration. Caddy is selected due to its ease of setup, automated certificate management, and lightweight footprint [6]. Compared to other reverse proxy solutions such as Nginx or Traefik, Caddy requires minimal configuration and comes with built-in HTTPS management, reducing deployment complexity and maintenance overhead.

The key responsibilities of Caddy in this project include:

- **SSL Termination**: Caddy handles HTTPS encryption, ensuring secure communication between clients and the server. The SSL termination at Caddy means that the project can make use of the security of HTTPS while the backend services communicate via HTTP [33] which simplifies the local development of the project.
- **Reverse Proxying**: It forwards requests from users to the appropriate backend service. In addition to the SSL termination mentioned previously, reverse proxying allows for future load balancing and protection from attacks among many other benefits [34].
- **Automatic Certificate Management**: Although self-signed certificates are currently used, they require periodic renewal as they expire over time. Moreover Caddy can also help to obtain certificates from a trusted certificate authority if configured. Caddy is able to automate both processes, thereby reducing maintenance effort and ensuring continuous secure communication.

By using Caddy, the project benefits from a simplified deployment process while maintaining secure and efficient internet traffic routing. However, the current use of self-signed certificates results in browser security warnings unless users manually trust the certificate. This approach was chosen due to the requirement for a domain name or the cost associated with obtaining SSL certificates from recognized certificate authorities. While this solution ensures encrypted communication, it introduces minor usability concerns that could be mitigated in future iterations. Since Caddy already includes built-in support for Let's Encrypt, transitioning to an automated certificate issuance process would be straightforward. This separation of concerns ensures that SSL certificate management remains decoupled from application logic, making future enhancements or domain integration straightforward without significant architectural changes.

4. Implementation

This section details the primary aspects of the implementation for both the frontend and backend. Please refer to the full source code in the project repository to view all functions, modules, and implementation details.

4.1 Frontend Development

The user interface allows users to create and customize chatbots using a simple dashboard. Users can specify chatbot behavior, upload reference files, and generate unique links for chatbot sharing. The frontend is implemented in React with TypeScript.

4.1.1 Login and Register UI

In order to keep track of the user's configurations, these configurations are tied to the user's account. As such, a method to register and login is implemented in the figures below.

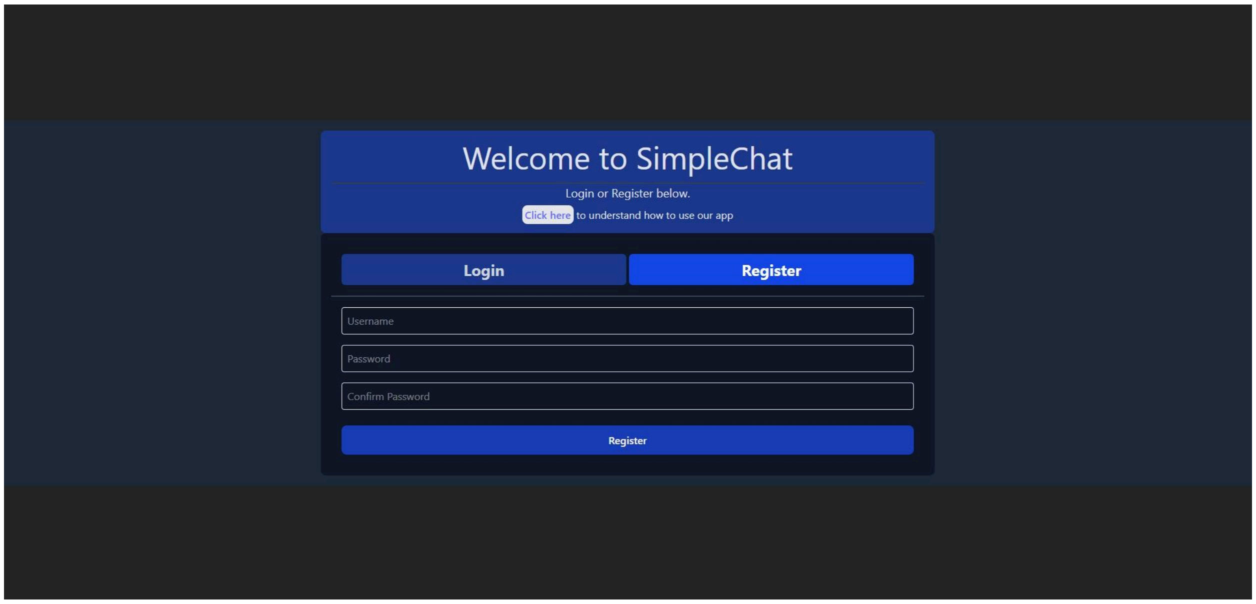


Figure 3: Register UI on welcome page

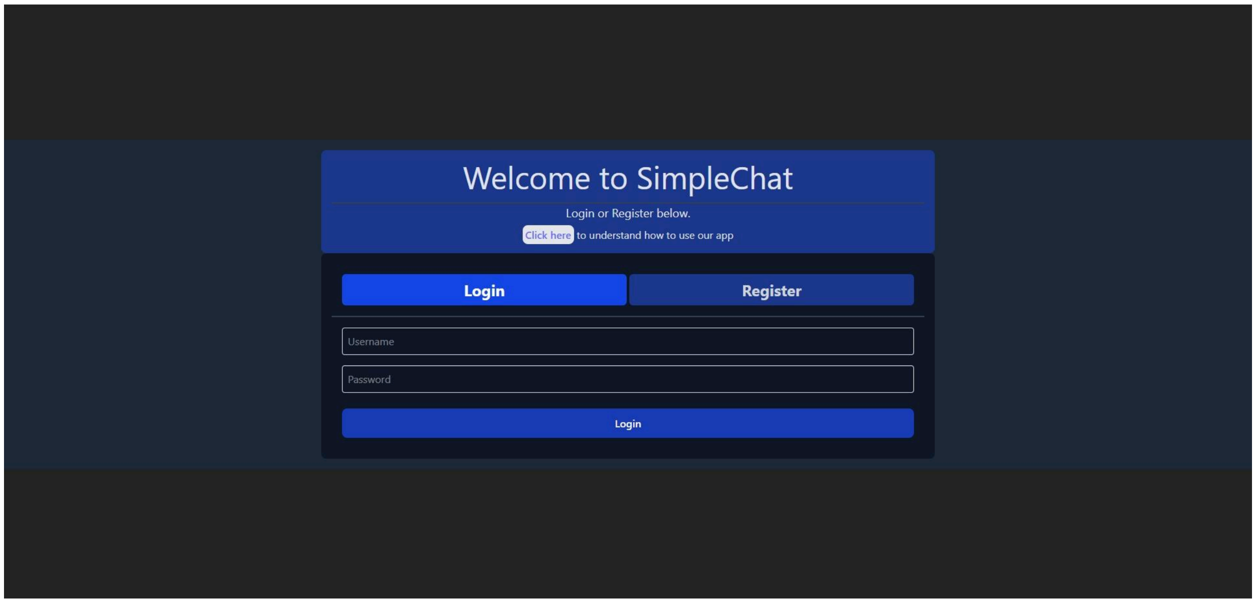
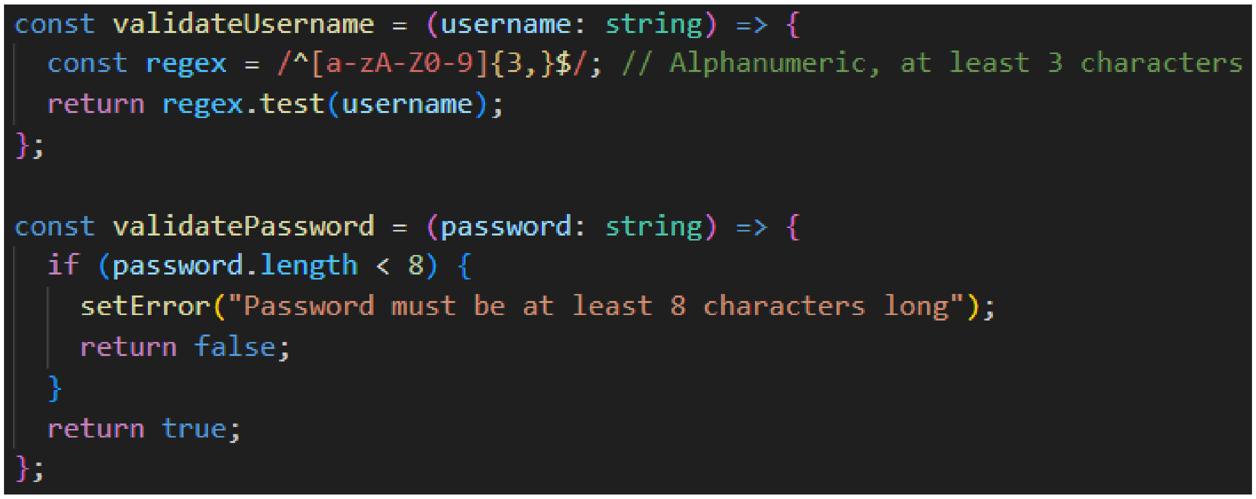


Figure 4: Login UI on welcome page

Some checks are also implemented in this page and the corresponding error messages will appear to notify users if the validation does not pass:

- When registering for an account, the application will check if the two passwords entered are the same in case users type their password wrongly.
- The username must not contain any spaces or special characters as the username forms part of the customised chatbot URL and upload file directory and these characters may cause security issues or potentially conflicting with system paths (refer to figure 5 for the validation code snippet).
- A minimum of three characters is introduced to ensure that usernames remain meaningful and distinguishable. Since the username forms part of the chatbot's custom URL, very short usernames (e.g., one or two characters) may result in URLs that are ambiguous, difficult to read, or potentially conflicting with system paths. By enforcing this constraint, it helps to improve usability and ensure a clear structure for chatbot links.
- Passwords must be more than 8 characters long to ensure some complexity in the password for security (refer to figure 5 for the validation code snippet).



```
const validateUsername = (username: string) => {
  const regex = /^[a-zA-Z0-9]{3,}$/; // Alphanumeric, at least 3 characters
  return regex.test(username);
};

const validatePassword = (password: string) => {
  if (password.length < 8) {
    setError("Password must be at least 8 characters long");
    return false;
  }
  return true;
};
```

Figure 5: Validation for username and password inputs on frontend

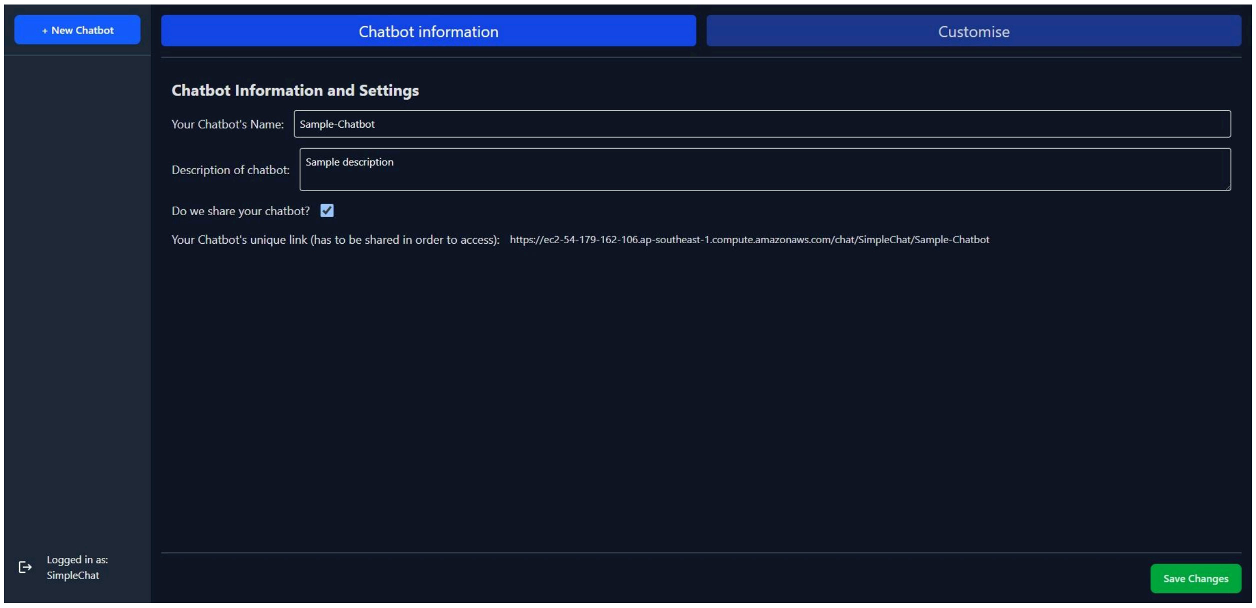
4.1.2 Chatbot Customization

The frontend provides a user-friendly interface for configuring and customizing individual chatbots. This allows users to tailor the chatbot's behavior and knowledge base to specific needs. The customization options are structured across two main tabs: "Chatbot Information" and "Customize."

4.1.2.1 Chatbot Information Tab:

This tab focuses on foundational information about the chatbot (Refer to figure 6 below):

- **Your Chatbot's Name**: A required field for specifying a unique name for the chatbot. This name will be used to identify the chatbot within the SimpleChat platform.
- **Description of chatbot**: An optional field that allows users to provide a brief description of the chatbot's purpose or functionality. This can help other users (if the chatbot is shared) understand its intended use.
- **Do we share your chatbot?**: A checkbox allowing users to determine whether the chatbot is intended to be shared.
- **Your Chatbot's unique link**: This readonly text displays the sharing link for users to share with other users so that other users can assess it



Chatbot Information and Settings

Your Chatbot's Name: Sample-Chatbot

Description of chatbot: Sample description

Do we share your chatbot? ✓

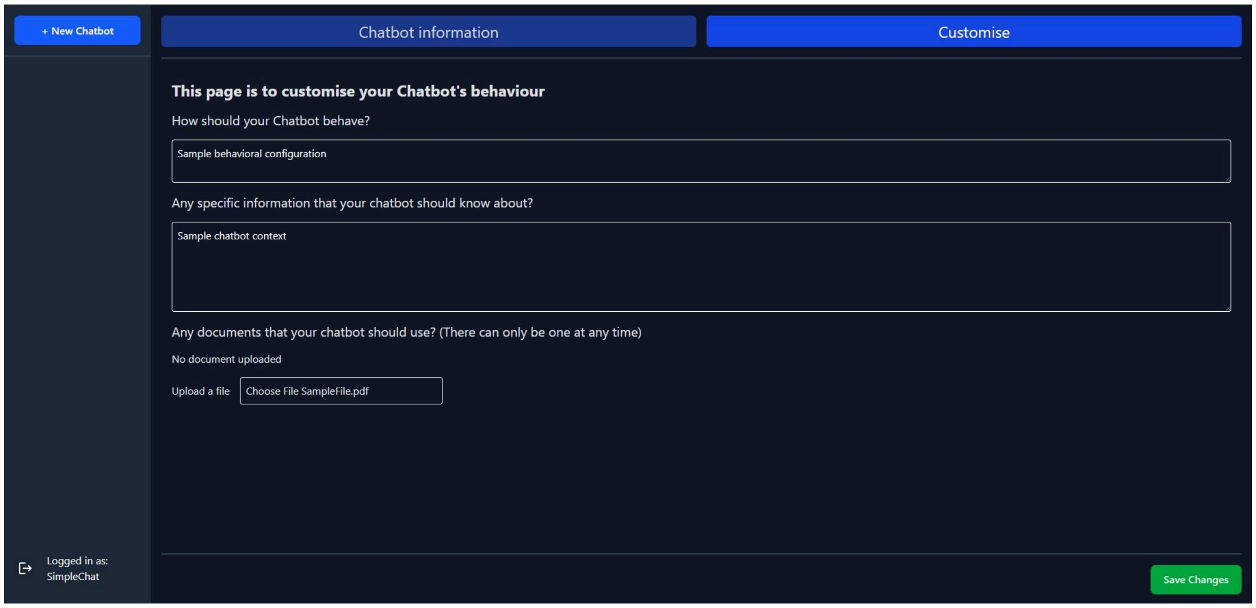
Your Chatbot's unique link (has to be shared in order to access): https://ec2-54-179-162-106.ap-southeast-1.compute.amazonaws.com/chat/SimpleChat/Sample-Chatbot

Figure 6: “Chatbot information” tab to customise chatbot

4.1.2.2 Customize Tab:

This tab allows users to fine-tune the chatbot's behavior and knowledge (Refer to figure 7 below):

- **How should your Chatbot behave?**: A text input field where users can provide instructions on the chatbot's desired personality, tone, and overall behavior (e.g., "Be friendly and helpful," "Respond in a professional and concise manner," "Act as a customer service representative").
- **Any specific information that your chatbot should know about?**: A text input field where users can provide specific facts, background information, or context that the chatbot should be aware of. This allows users to prime the chatbot with relevant knowledge.
- **Any documents that your chatbot should use?**: This section allows users to upload a file that the chatbot can use as a knowledge base. The system supports uploading of files, indicated by the "Choose File" button. Once a file is uploaded, it is displayed, with a button to include the original file that can be toggled to choose to remove the file. This feature is currently limited to only allow one file at a time to limit the complexity of file storage and tracking of user files.



This page is to customise your Chatbot's behaviour

How should your Chatbot behave?

Sample behavioral configuration

Any specific information that your chatbot should know about?

Sample chatbot context

Any documents that your chatbot should use? (There can only be one at any time)

No document uploaded

Upload a file: Choose File SampleFile.pdf

Figure 7: Chatbot customisation “Customise” tab to customise chatbot

4.1.2.3 Input Validation

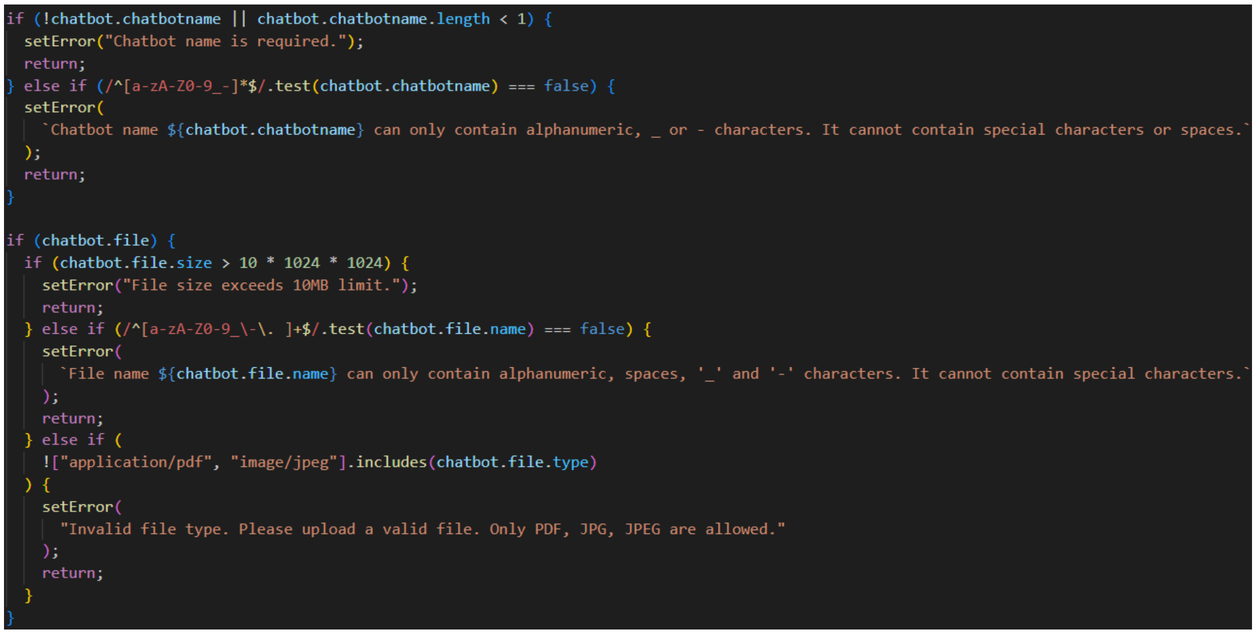
There is input validation for the chatbot name and uploaded file only, as the other fields are intended for users to provide flexible textual descriptions and information. Strict validation on those fields would unduly restrict user creativity and the ability to provide nuanced instructions to the chatbot. However, this does introduce a slight tradeoff and the backend would need to apply additional validations or sanitization to prevent potential vulnerabilities such as cross-site scripting (XSS). More details will be included in the backend section.

The explicit validation implemented for the chatbot name and file upload fields is critical for several reasons (refer to figure 8 for the validation code snippet):

- **Chatbot Name**: The chatbot name validation ensures that a name is provided, as it is a required identifier for the chatbot within the system. Furthermore, the regex validation restricts the allowed characters to alphanumeric characters, underscores, and hyphens. This helps prevent injection attacks, ensures compatibility with backend systems (e.g., database queries or filename generation), and maintains a consistent naming convention across the platform.

- **File Upload**: The file upload validation addresses several potential security and usability concerns.
    - The file size is limited to 10MB, preventing potential denial-of-service attacks and ensuring efficient resource utilization on the server. Processing extremely large files can strain system resources and negatively impact performance especially since these files are reused for chatbot conversations.

- Similar to the chatbot name, the filename is restricted to only alphanumeric, spaces, underscore and dash characters. Special characters have been excluded to prevent potentially malicious filenames or filenames that could cause issues with file storage or retrieval on the server.
- The accepted file types are restricted to PDF, JPG and JPEG images. This prevents users from uploading arbitrary files, which could pose a security risk or be incompatible with the intended use of the file within the chatbot system. Moreover, there are also the input limitations of the external API [32] and these file types have been tested to be working.



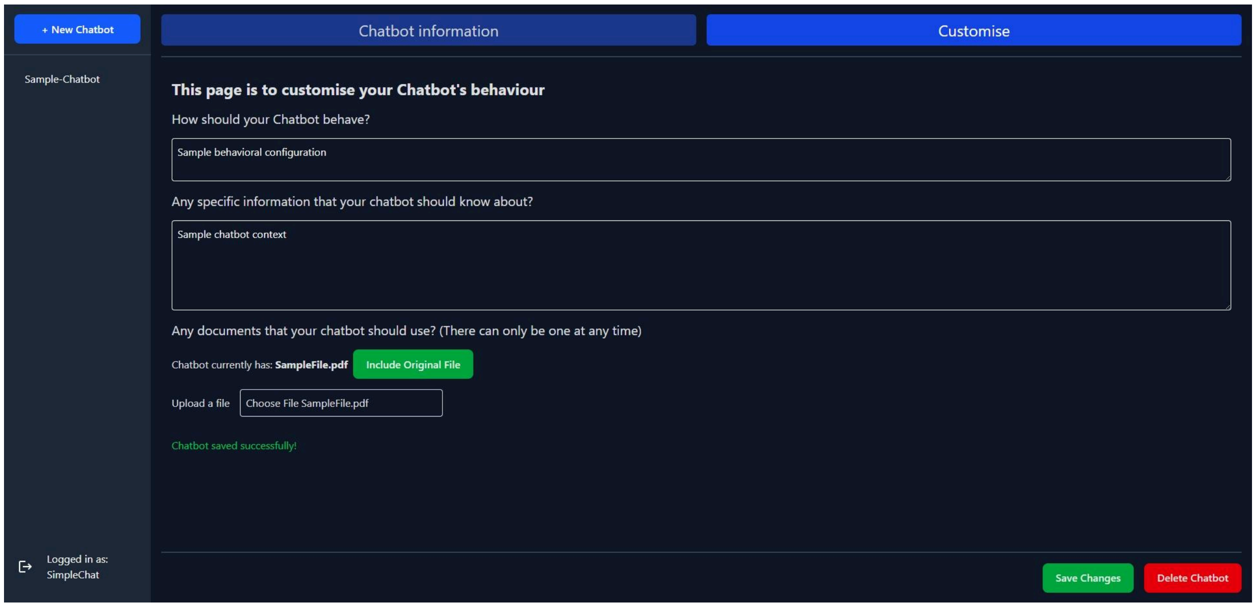
```
if (!chatbot.chatbotname || chatbot.chatbotname.length < 1) {
  setError("Chatbot name is required.");
  return;
} else if (/^[a-zA-Z0-9_-]*$/.test(chatbot.chatbotname) === false) {
  setError(
    `Chatbot name ${chatbot.chatbotname} can only contain alphanumeric, _ or - characters. It cannot contain special characters or spaces.`
  );
  return;
}

if (chatbot.file) {
  if (chatbot.file.size > 10 * 1024 * 1024) {
    setError("File size exceeds 10MB limit.");
    return;
  } else if (/^[a-zA-Z0-9_\-\. ]+$/.test(chatbot.file.name) === false) {
    setError(
      `File name ${chatbot.file.name} can only contain alphanumeric, spaces, '_' and '-' characters. It cannot contain special characters.`
    );
    return;
  } else if (
    !["application/pdf", "image/jpeg"].includes(chatbot.file.type)
  ) {
    setError(
      "Invalid file type. Please upload a valid file. Only PDF, JPG, JPEG are allowed."
    );
    return;
  }
}
```

Figure 8: Chatbot name and file validation code snippet

4.1.2.4 Additional Notes

There are also a few other features related to the frontend chatbot customisation as follows:

- **Save Changes**: A button to save the configurations made to the chatbot to the database, allowing users to create a chatbot or save the updated configuration. Figure 9 and 10 shows the success message and the code that sends the user inputs to the backend.



This page is to customise your Chatbot's behaviour

How should your Chatbot behave?

Sample behavioral configuration

Any specific information that your chatbot should know about?

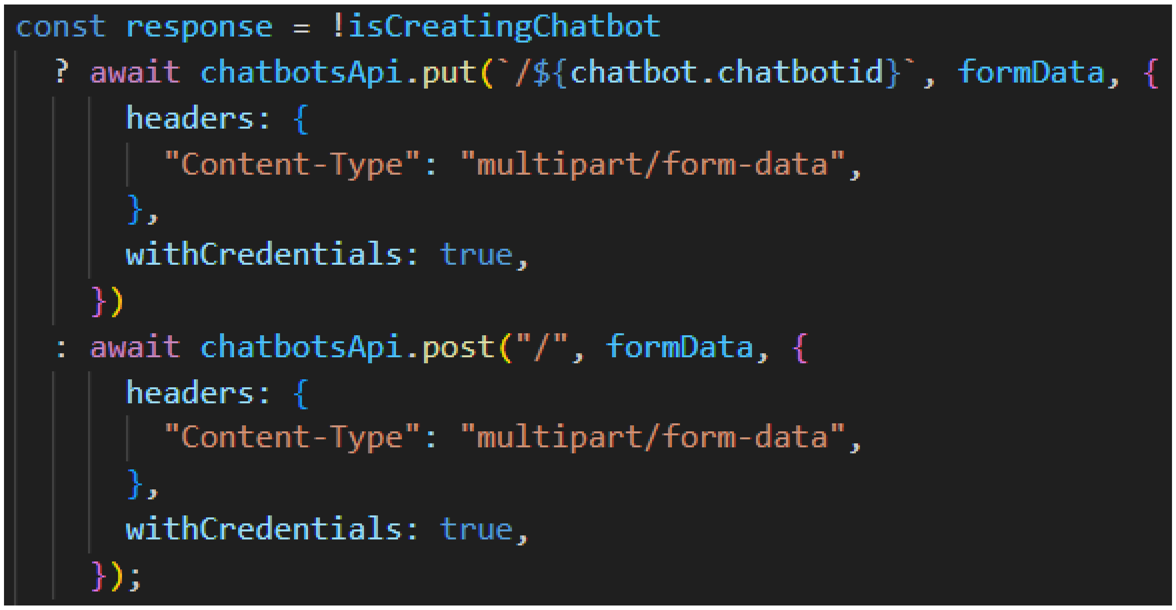
Sample chatbot context

Any documents that your chatbot should use? (There can only be one at any time)

Chatbot currently has: **SampleFile.pdf**

Chatbot saved successfully!

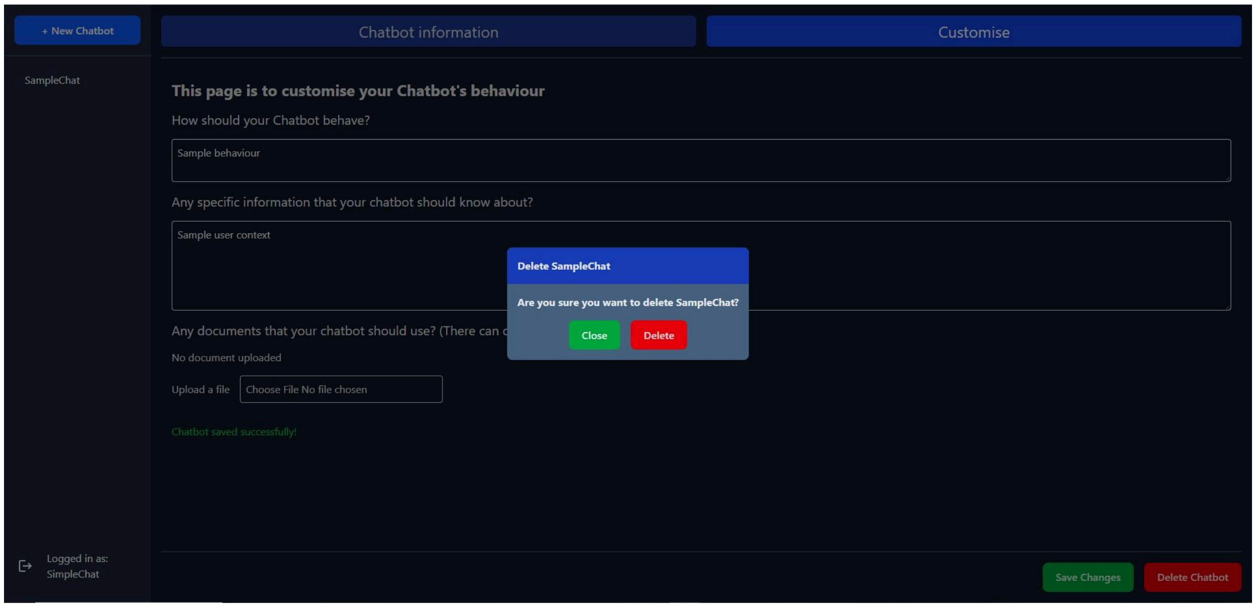
Figure 9: UI with success message after user clicks on save changes



```
const response = !isCreatingChatbot
  ? await chatbotsApi.put(`/${chatbot.chatbotid}`, formData, {
      headers: {
        "Content-Type": "multipart/form-data",
      },
      withCredentials: true,
    })
  : await chatbotsApi.post("/", formData, {
      headers: {
        "Content-Type": "multipart/form-data",
      },
      withCredentials: true,
    });
```

Figure 10: Code snippet of sending the user input to the backend endpoint.

- **Delete Chatbot**: A button to delete the chatbot permanently. Since this is a destructive action, a confirmation modal is also added. Figure 11 and 12 shows the confirmation modal for deleting chatbot and the code snippet that sends the request to the backend endpoint.



This page is to customise your Chatbot's behaviour

How should your Chatbot behave?

Sample behaviour

Any specific information that your chatbot should know about?

Sample user context

Any documents that your chatbot should use? (There can o

No document uploaded

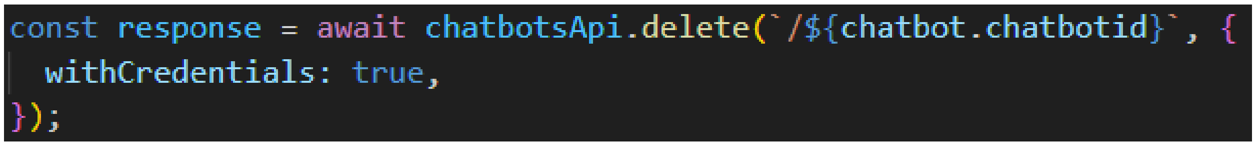
Upload a file: Choose File No file chosen

Chatbot saved successfully!

Delete SampleChat

Are you sure you want to delete SampleChat?

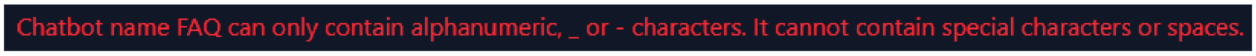
Figure 11: Delete chatbot confirmation modal



```
const response = await chatbotsApi.delete(`/${chatbot.chatbotid}`, {
  withCredentials: true,
});
```

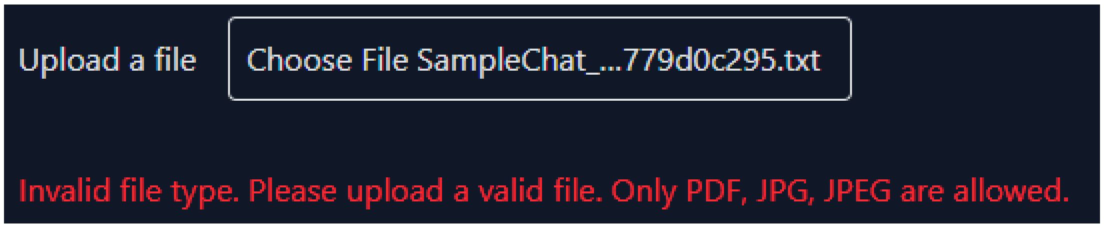
Figure 12: Code snippet of sending the user input to the backend endpoint.

- **Error Handling**: General error handling such as those returned by input validation are included and displayed on the UI. Please see figure 13 and 14 for the error messages.



Chatbot name FAQ can only contain alphanumeric, _ or - characters. It cannot contain special characters or spaces.

Figure 13: Error from file input validation

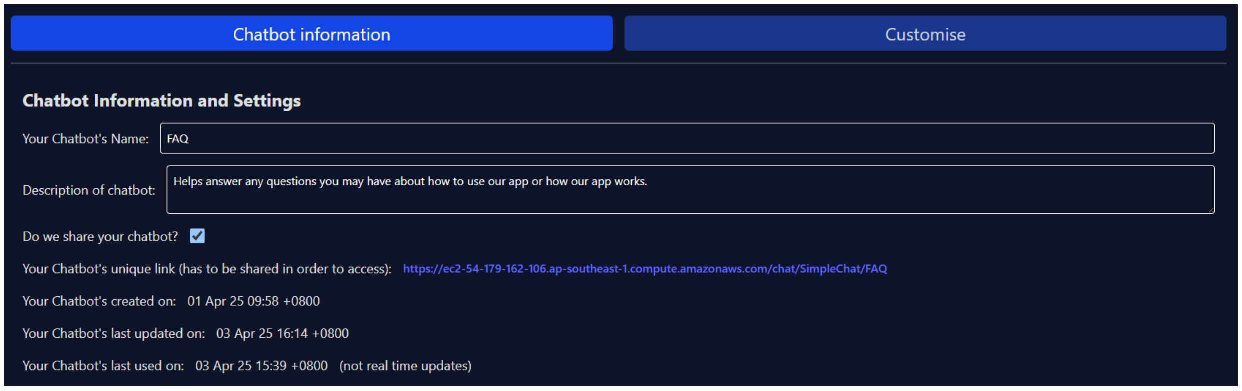


Upload a file: Choose File SampleChat_...779d0c295.txt

Invalid file type. Please upload a valid file. Only PDF, JPG, JPEG are allowed.

Figure 14: Error from file input validation

- **Chatbot metadata**: The figure below shows the additional metadata is shown to users on the “Chatbot information” tab after users create their chatbot, namely the created data, updated date and when was the chatbot last used by a user.



Chatbot Information and Settings

Your Chatbot's Name: FAQ

Description of chatbot: Helps answer any questions you may have about how to use our app or how our app works.

Do we share your chatbot? ✓

Your Chatbot's unique link (has to be shared in order to access): https://ec2-54-179-162-106.ap-southeast-1.compute.amazonaws.com/chat/SimpleChat/FAQ

Your Chatbot's created on: 01 Apr 25 09:58 +0800

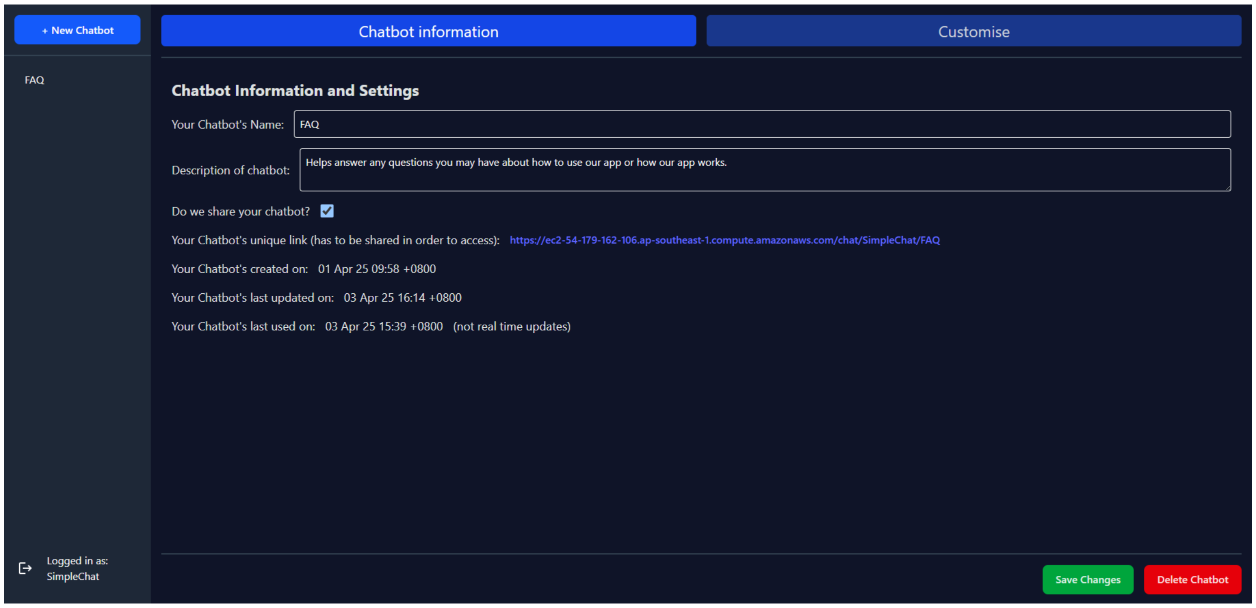
Your Chatbot's last updated on: 03 Apr 25 16:14 +0800

Your Chatbot's last used on: 03 Apr 25 15:39 +0800 (not real time updates)

Figure 15: Chatbot metadata

4.1.3 FAQ and User Guide Chatbot

A chatbot has been set up to explain to users how to make use of SimpleChat. This chatbot is made using SimpleChat and aims to teach users how to use the app. The chatbot is provided with the inputs as shown in figure 16 and 17, a link to the uploaded file is included under the appendix. The information provided is aimed to be comprehensive to cover most questions that users may ask about how to use SimpleChat or any background information about SimpleChat. The FAQ list



Chatbot Information and Settings

Your Chatbot's Name: FAQ

Description of chatbot: Helps answer any questions you may have about how to use our app or how our app works.

Do we share your chatbot? ✓

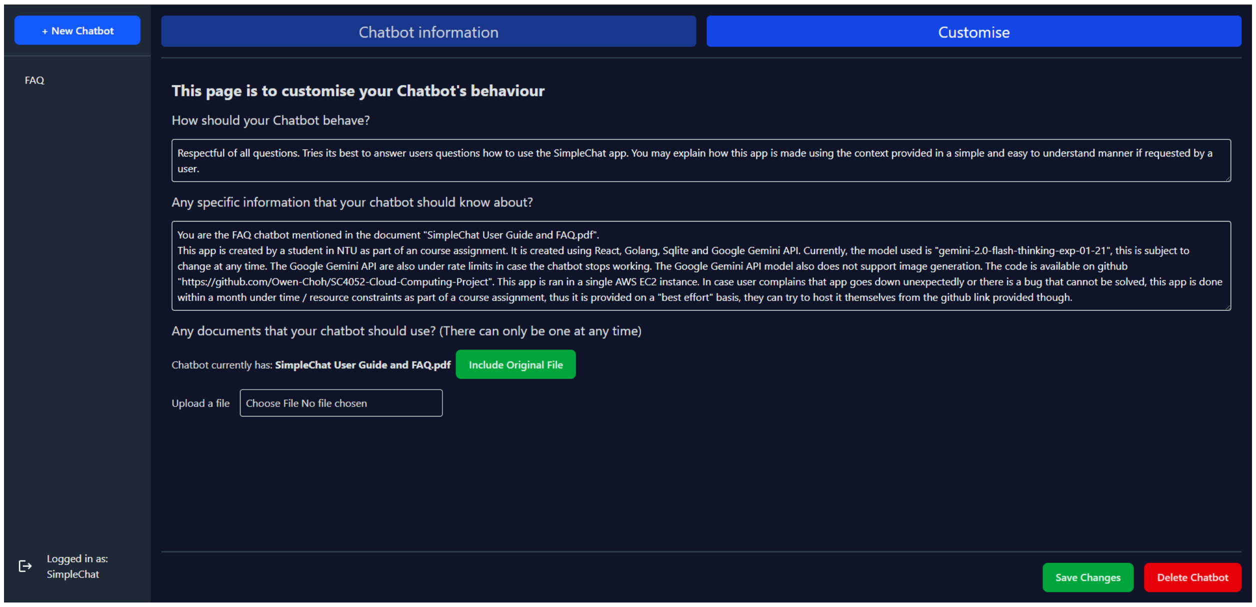
Your Chatbot's unique link (has to be shared in order to access): https://ec2-54-179-162-106.ap-southeast-1.compute.amazonaws.com/chat/SimpleChat/FAQ

Your Chatbot's created on: 01 Apr 25 09:58 +0800

Your Chatbot's last updated on: 03 Apr 25 16:14 +0800

Your Chatbot's last used on: 03 Apr 25 15:39 +0800 (not real time updates)

Figure 16: “Chatbot information” inputs for FAQ chatbot



This page is to customise your Chatbot's behaviour

How should your Chatbot behave?

Respectful of all questions. Tries its best to answer users questions how to use the SimpleChat app. You may explain how this app is made using the context provided in a simple and easy to understand manner if requested by a user.

Any specific information that your chatbot should know about?

You are the FAQ chatbot mentioned in the document "SimpleChat User Guide and FAQ.pdf".
This app is created by a student in NTU as part of an course assignment. It is created using React, Golang, Sqlite and Google Gemini API. Currently, the model used is "gemini-2.0-flash-thinking-exp-01-21", this is subject to change at any time. The Google Gemini API are also under rate limits in case the chatbot stops working. The Google Gemini API model also does not support image generation. The code is available on github "https://github.com/Owen-Choh/SC4052-Cloud-Computing-Project". This app is ran in a single AWS EC2 instance. In case user complains that app goes down unexpectedly or there is a bug that cannot be solved, this app is done within a month under time / resource constraints as part of a course assignment, thus it is provided on a "best effort" basis, they can try to host it themselves from the github link provided though.

Any documents that your chatbot should use? (There can only be one at any time)

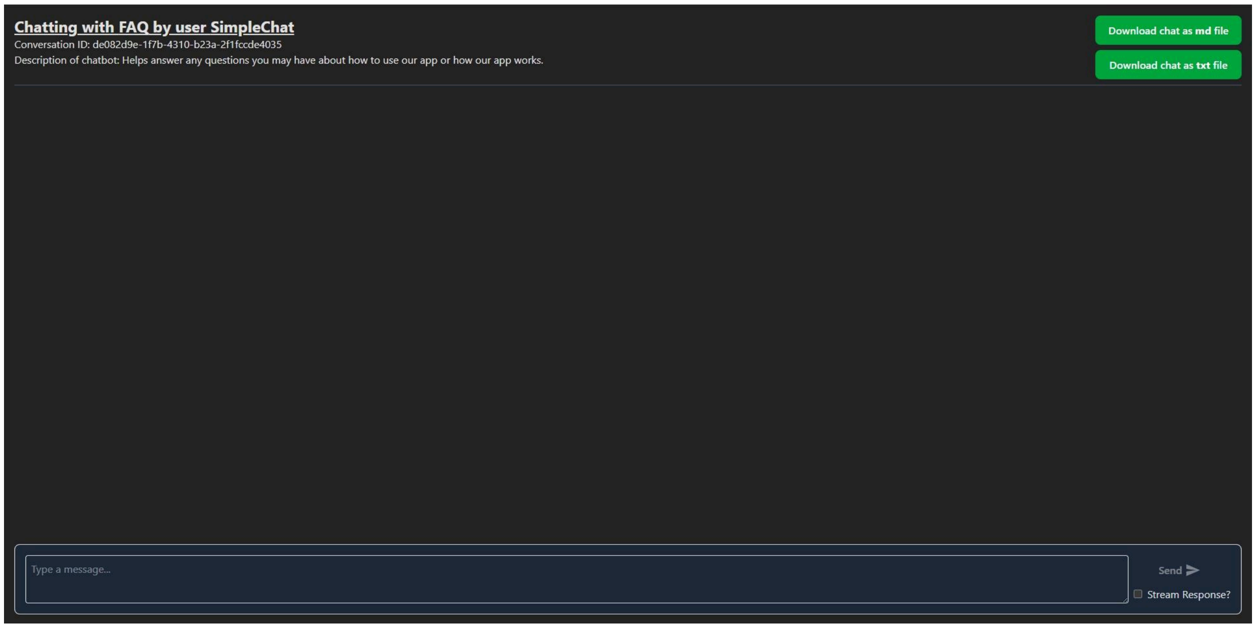
Chatbot currently has: **SimpleChat User Guide and FAQ.pdf**

Upload a file: Choose File No file chosen

Figure 17: “Customisation” inputs for FAQ chatbot

4.1.3 Conversation Feature

Users can engage with their customized chatbot by accessing the unique link displayed in the "Chatbot Information" tab. To streamline user experience and encourage wider adoption, the chatbot interface is designed to be accessible without requiring users to log in. This allows chatbot creators to easily share the link for testing and feedback, as mandatory account creation could discourage casual users from trying the chatbot, ultimately limiting the feedback received. The initial conversation page UI is shown in the figure below.



Chatting with FAQ by user SimpleChat
Conversation ID: de082d9e-1f7b-4310-b23a-2f1fccde4035
Description of chatbot: Helps answer any questions you may have about how to use our app or how our app works.

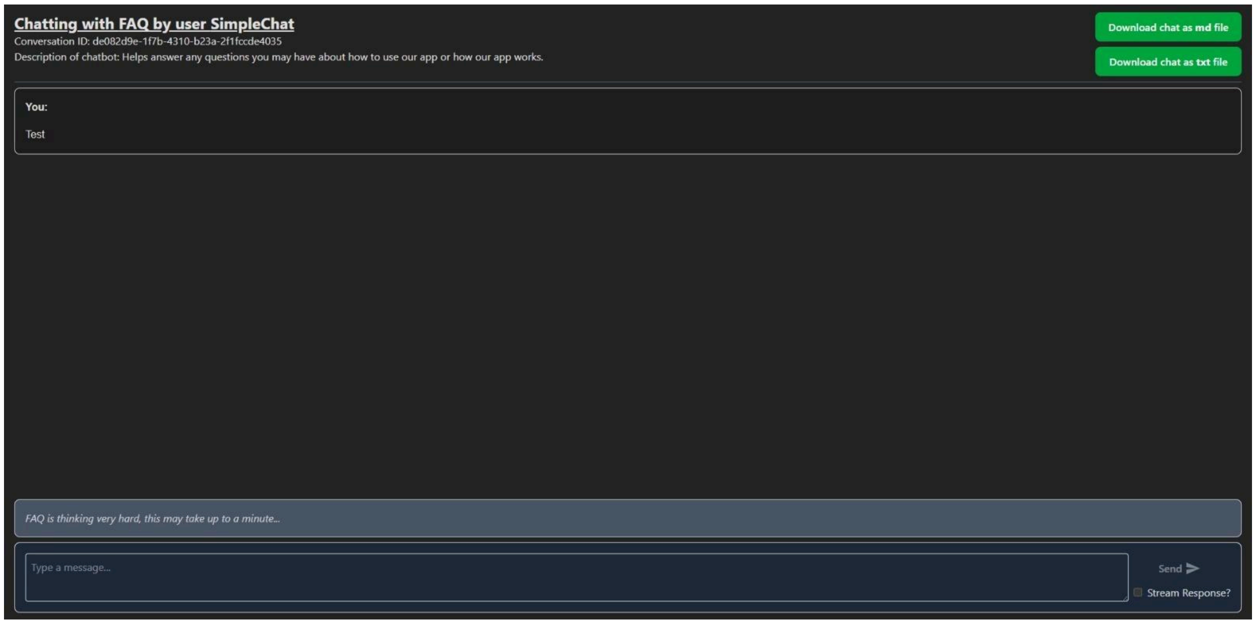
Figure 18: Initial conversation page

UI Design and Functionality
The user interface is designed to provide a familiar chat experience, mirroring common messaging applications like WhatsApp and ChatGPT. Key elements include:

- **Text Input**: Located at the bottom of the screen, this field accepts user input for initiating conversations.
- **Send Button**: Allows users to send their message to the chatbot.

Streamed vs. Non-Streamed Responses
Users can choose between two response modes by using the checkbox on the bottom right:

- **Non-Streamed Responses**: (Figure 19 and 21) This mode operates similarly to traditional API interaction. The user's query is sent to the backend, which processes the request and returns the complete chatbot response after it's fully generated.
- **Streamed Responses**: (Figure 20 and 21) This mode provides a more interactive experience. The application requests the backend to return partial chatbot responses as soon as they are available, rather than waiting for the entire response. This provides a real-time, conversational feel, delivering a more pleasant user experience.

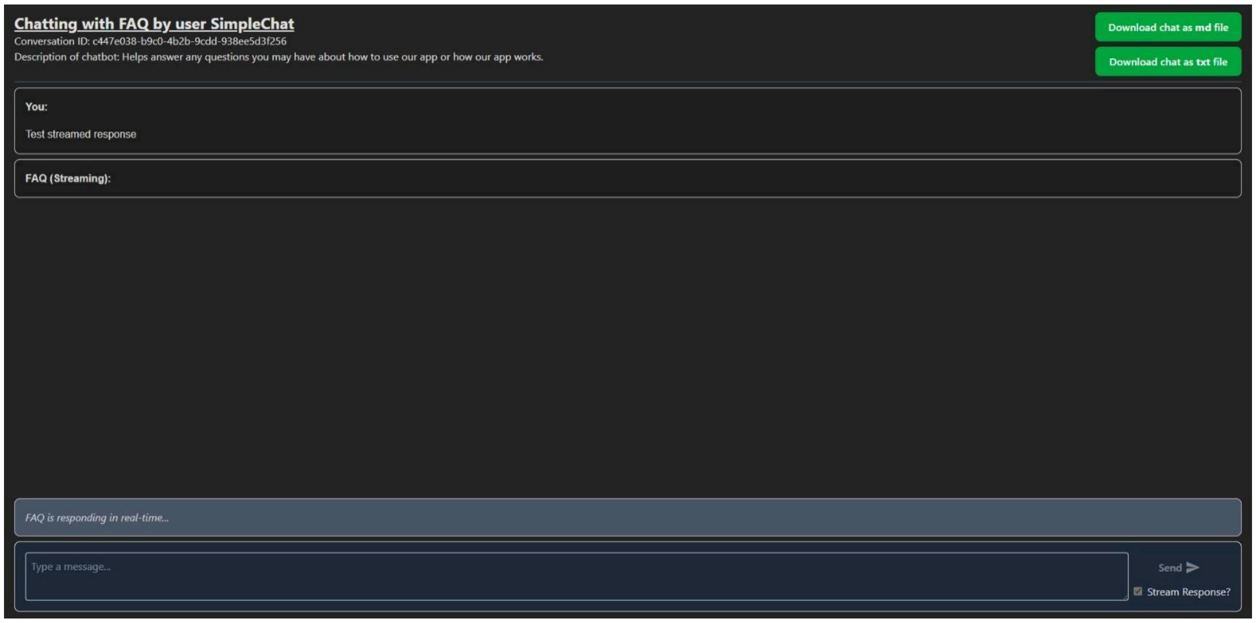


Chatting with FAQ by user SimpleChat
Conversation ID: de082d9e-1f7b-4310-b23a-2f1fccde4035
Description of chatbot: Helps answer any questions you may have about how to use our app or how our app works.

You:
Test

FAQ is thinking very hard, this may take up to a minute...

Figure 19: Chatting with non-streamed response


Chatting with FAQ by user SimpleChat

Conversation ID: c447e038-b9c0-4b2b-9cdd-938ee5d3f256

Description of chatbot: Helps answer any questions you may have about how to use our app or how our app works.

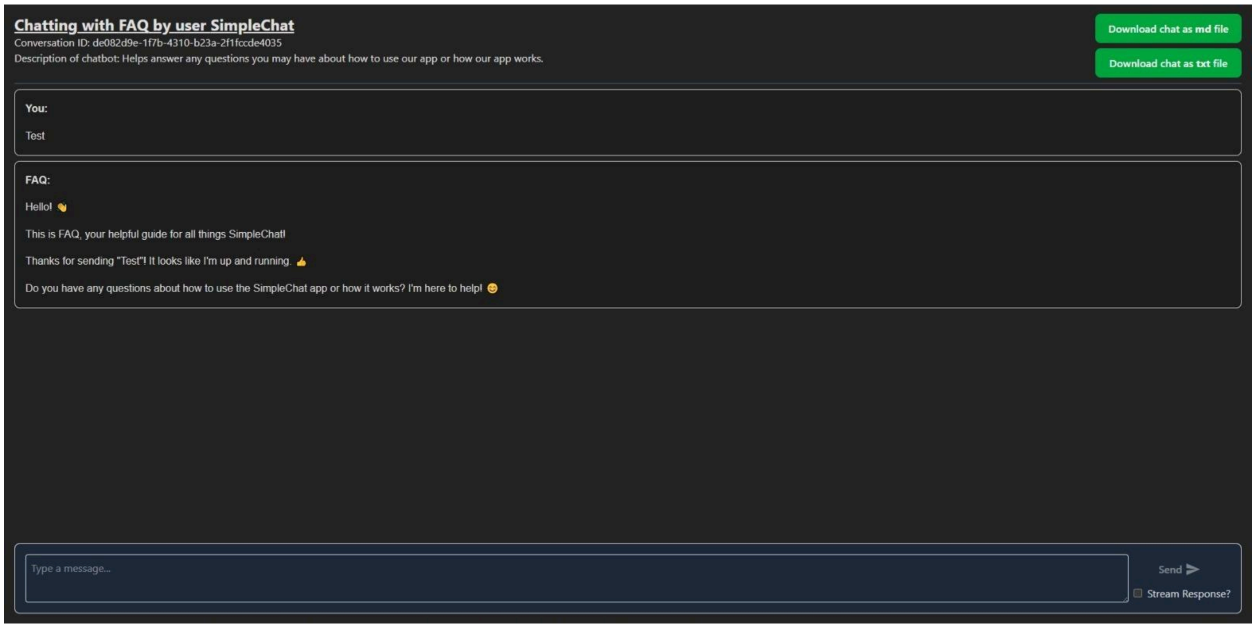
You:

Test streamed response

FAQ (Streaming):

FAQ is responding in real-time...

Figure 20: Chatting with streamed response


Chatting with FAQ by user SimpleChat

Conversation ID: de082d9e-1f7b-4310-b23a-2f1fccde4035

Description of chatbot: Helps answer any questions you may have about how to use our app or how our app works.

You:

Test

FAQ:

Hello! 👋

This is FAQ, your helpful guide for all things SimpleChat!

Thanks for sending "Test"! It looks like I'm up and running. 👍

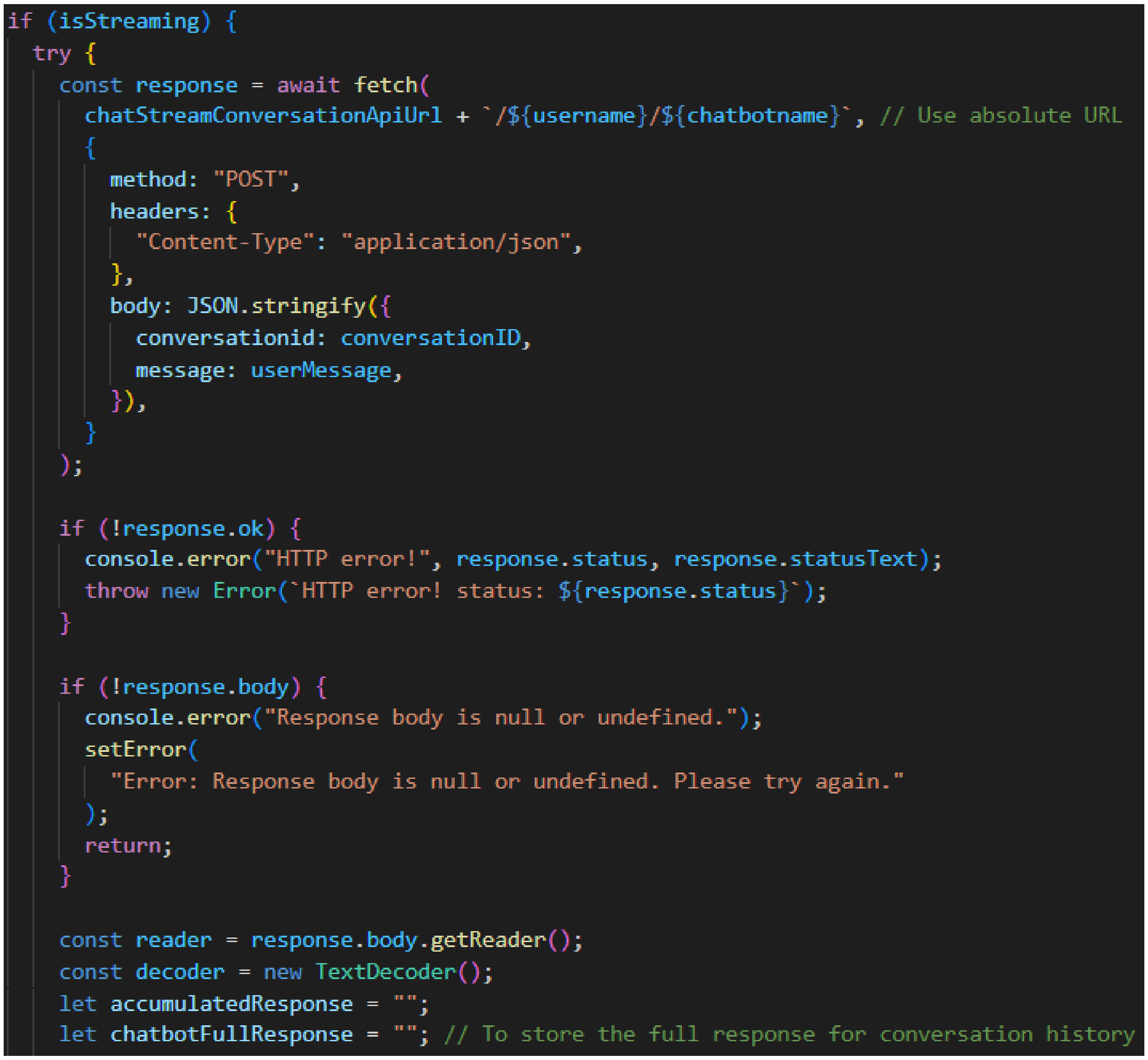
Do you have any questions about how to use the SimpleChat app or how it works? I'm here to help! 😊

Figure 21: Receiving the chatbot's response

Given the current implementation where users cannot persist or return to the conversation if they navigate away from the page, the interface provides readily accessible options for downloading the conversation history in both Markdown (.md) and plain text (.txt) formats. This allows users to retain a copy of their interactions for future reference or sharing, mitigating the potential data loss from a lack of session persistence.

Implementation of Streamed Responses
Streamed responses are implemented using Server-Sent Events (SSE), as described in Mozilla documentation [35]. SSE were chosen over WebSockets [36] because the primary need is for unidirectional data streaming (from the chatbot to the user). SSE offers a simpler implementation compared to the bidirectional communication capabilities of WebSockets, which are not required for this use case.

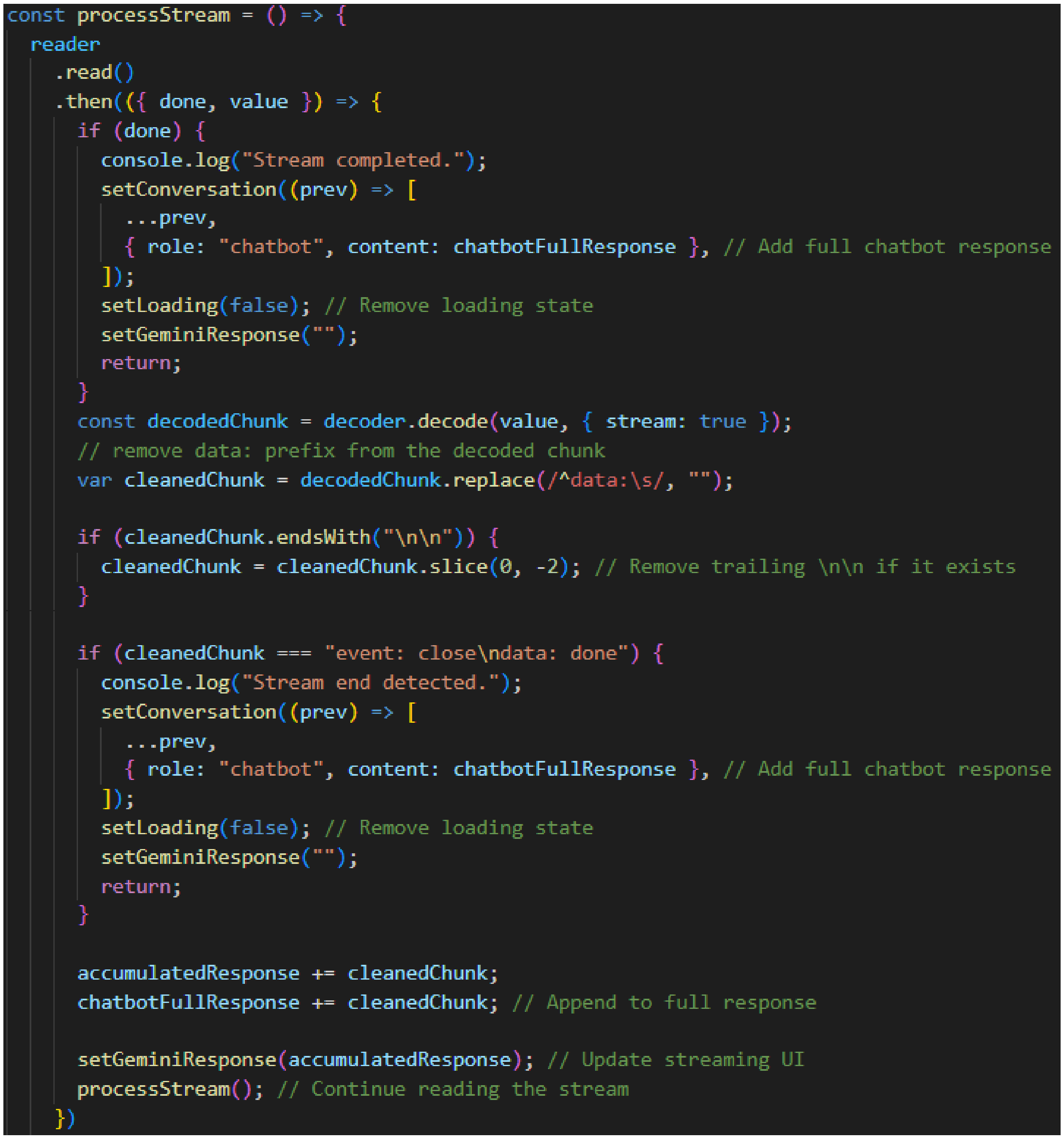
This figure illustrates the frontend implementation of Server-Sent Events (SSE) in a React-based chatbot. The first part showcases the asynchronous request handling, where the frontend sends user messages to the chatbot API and handles streaming responses.



```
if (isStreaming) {
  try {
    const response = await fetch(
      chatStreamConversationApiUrl + `/${username}/${chatbotname}`, // Use absolute URL
      {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify({
          conversationid: conversationID,
          message: userMessage,
        }),
      }
    );

    if (!response.ok) {
      console.error("HTTP error!", response.status, response.statusText);
      throw new Error(`HTTP error! status: ${response.status}`);
    }

    if (!response.body) {
      console.error("Response body is null or undefined.");
      setError(
        "Error: Response body is null or undefined. Please try again."
      );
      return;
    }

    const reader = response.body.getReader();
    const decoder = new TextDecoder();
    let accumulatedResponse = "";
    let chatbotFullResponse = ""; // To store the full response for conversation history
```

Figure 22: Implementation of SSE in the React Frontend (Fetching & Stream Handling)

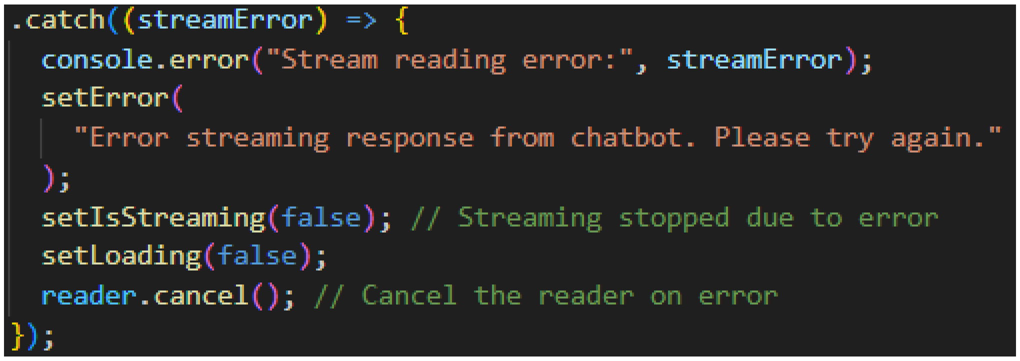
The second figure demonstrates the handling of streamed responses using a TextDecoder [37] to decode the incoming responses. It also includes processing received chunks, accumulating the chatbot's response, and dynamically updating the UI to reflect the ongoing conversation.



```
const processStream = () => {
  reader
    .read()
    .then(({ done, value }) => {
      if (done) {
        console.log("Stream completed.");
        setConversation((prev) => [
          ...prev,
          { role: "chatbot", content: chatbotFullResponse }, // Add full chatbot response
        ]);
        setLoading(false); // Remove loading state
        setGeminiResponse("");
        return;
      }
      const decodedChunk = decoder.decode(value, { stream: true });
      // remove data: prefix from the decoded chunk
      var cleanedChunk = decodedChunk.replace(/^data:\s/, "");

      if (cleanedChunk.endsWith("\n\n")) {
        cleanedChunk = cleanedChunk.slice(0, -2); // Remove trailing \n\n if it exists
      }

      if (cleanedChunk === "event: close\ndata: done") {
        console.log("Stream end detected.");
        setConversation((prev) => [
          ...prev,
          { role: "chatbot", content: chatbotFullResponse }, // Add full chatbot response
        ]);
        setLoading(false); // Remove loading state
        setGeminiResponse("");
        return;
      }

      accumulatedResponse += cleanedChunk;
      chatbotFullResponse += cleanedChunk; // Append to full response

      setGeminiResponse(accumulatedResponse); // Update streaming UI
      processStream(); // Continue reading the stream
    })
```

Figure 23: Processing and Rendering SSE Streamed Responses in React

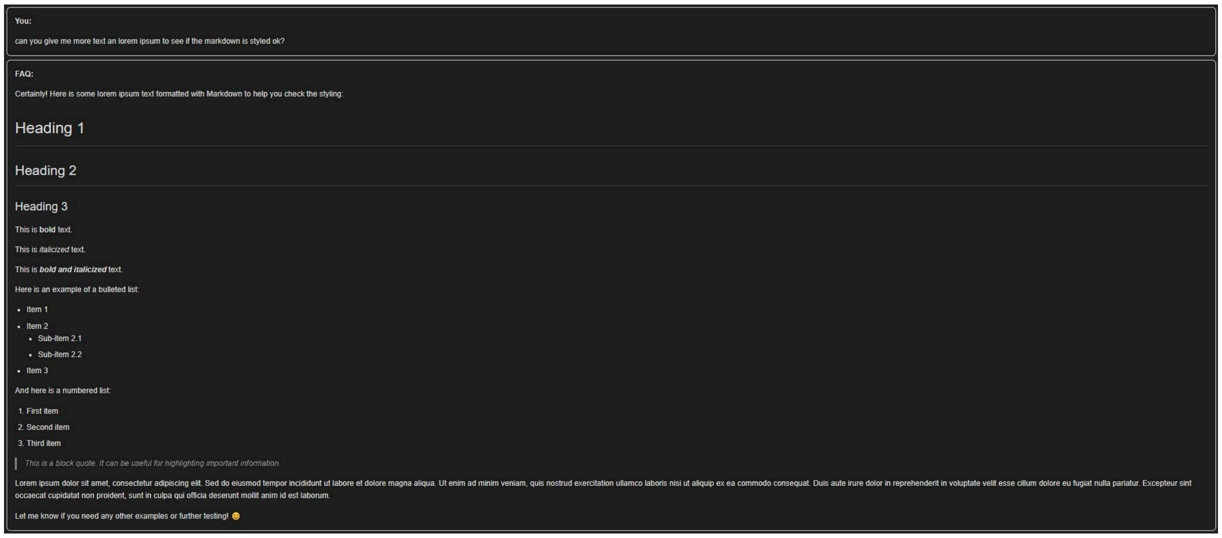
Lastly, there is also error handling included if the API returns an error as shown in the figure below.



```
.catch((streamError) => {
  console.error("Stream reading error:", streamError);
  setError(
    "Error streaming response from chatbot. Please try again."
  );
  setIsStreaming(false); // Streaming stopped due to error
  setLoading(false);
  reader.cancel(); // Cancel the reader on error
});
```

Figure 24: SSE response error handling

Upon receiving the chatbot's response, the frontend appends it to the current response and renders it. This is done by making use of the "useState" hook provided by the React framework [38] and the react-markdown package to help convert the markdown formatted responses into html elements for easier styling on the UI [39]. The styling of the elements is largely generated using ChatGPT (accessed Mar. 29, 2025) with minor manual adjustments with regards to formatting issues as there are a large number of html elements that require customised CSS styling. As it is a large file, please refer to the CSS styling in the "markdown.css" file provided in the source code (refer to the appendix) for more information. The styling has been tested by using the chatbot service when testing and also generating placeholder text to check the styling as shown in the figure below.



You:

can you give me more text an lorem ipsum to see if the markdown is styled ok?

FAQ:

Certainly! Here is some lorem ipsum text formatted with Markdown to help you check the styling:

Heading 1

Heading 2

Heading 3

This is **bold** text.

This is *italicized* text.

This is ***bold and italicized*** text.

Here is an example of a bulleted list:

- Item 1
- Item 2
  - Sub-item 2.1
  - Sub-item 2.2
- Item 3

And here is a numbered list:

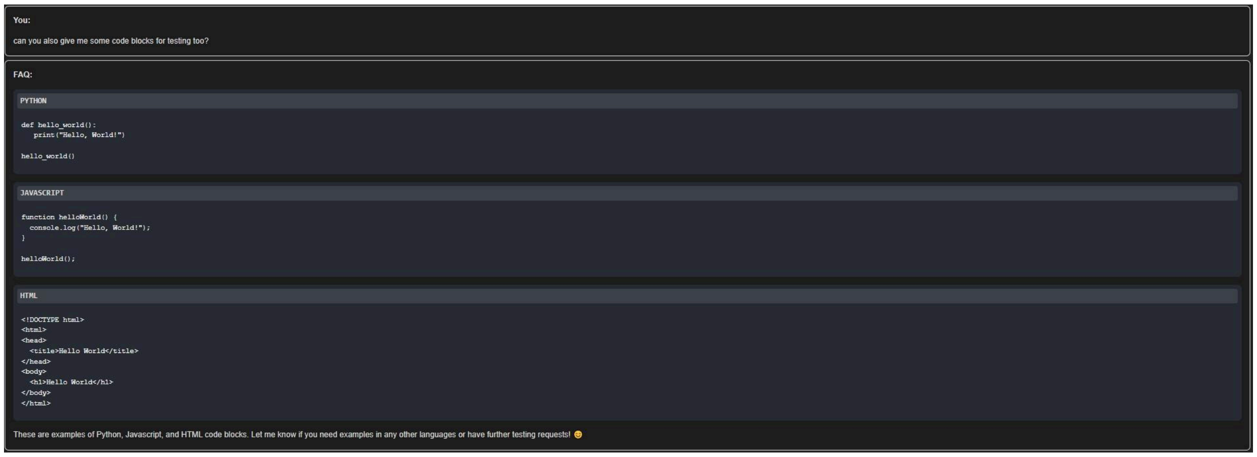
1. First item
2. Second item
3. Third item

> *This is a block quote. It can be useful for highlighting important information.*

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Let me know if you need any other examples or further testing! 😊

Figure 25: Testing the formatting of markdown headers and text



You:

can you also give me some code blocks for testing too?

FAQ:

PYTHON

```
def hello_world():
   print("Hello, World!")

hello_world()
```

JAVASCRIPT

```
function helloWorld() {
  console.log("Hello, World!");
}

helloWorld();
```

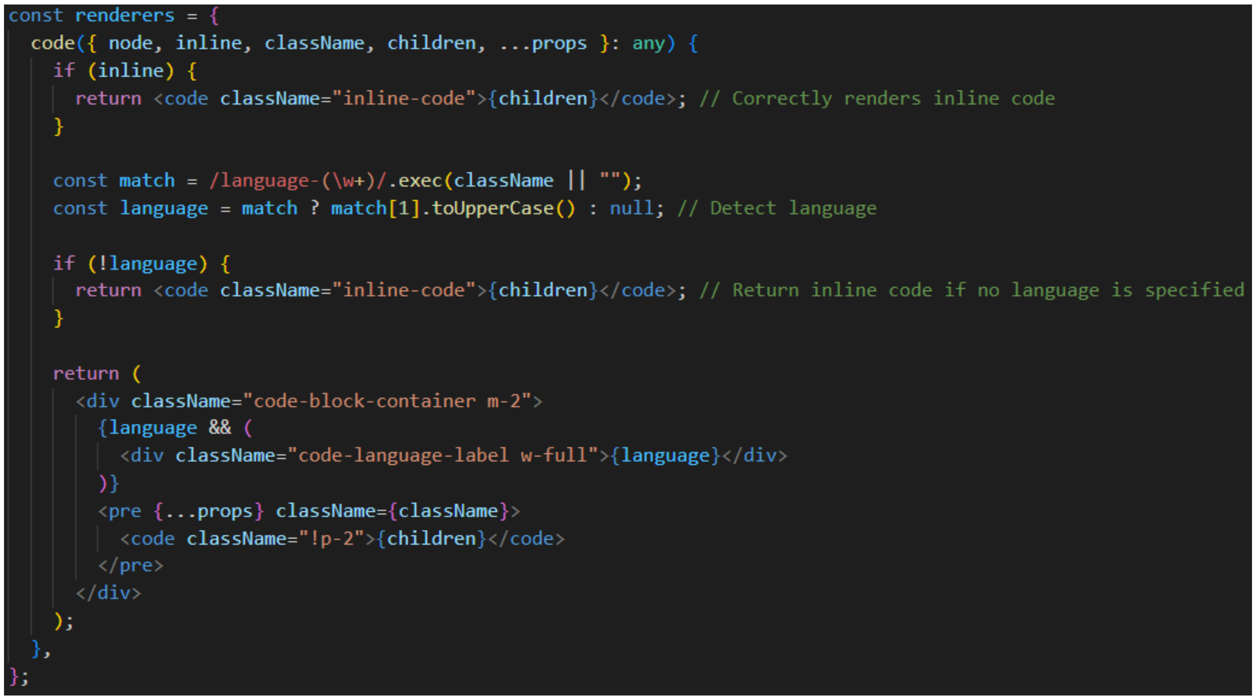
HTML

```
<!DOCTYPE html>
<html>
<head>
  <title>Hello World</title>
</head>
<body>
  <h1>Hello World</h1>
</body>
</html>
```

These are examples of Python, Javascript, and HTML code blocks. Let me know if you need examples in any other languages or have further testing requests! 😊

Figure 26: Testing the formatting of markdown code blocks

A custom function called a renderer is also written in order to have the UI display the type of code since it is not the default behaviour when using react-markdown.



```
const renderers = {
  code({ node, inline, className, children, ...props }: any) {
    if (inline) {
      return <code className="inline-code">{children}</code>; // Correctly renders inline code
    }

    const match = /language-(\w+)/.exec(className || "");
    const language = match ? match[1].toUpperCase() : null; // Detect language

    if (!language) {
      return <code className="inline-code">{children}</code>; // Return inline code if no language is specified
    }

    return (
      <div className="code-block-container m-2">
        {language && (
          <div className="code-language-label w-full">{language}</div>
        )}
        <pre {...props} className={className}>
          <code className="!p-2">{children}</code>
        </pre>
      </div>
    );
  },
};
```

Figure 27: Custom renderer function for markdown code block formatting

4.2 Backend Development

The backend implementation involves various modules and functionalities. This section highlights the key components. Please refer to the source code for the complete implementation for more details.

4.2.1 Database Design

To support user accounts, chatbot customization, document management, and conversation history, the platform uses SQLite as a lightweight, file-based relational database. This choice was ideal for local development and prototyping, offering simplicity and ease of setup without requiring external dependencies.

The system initializes the database with four main tables:

1. users
2. chatbots
3. conversations
4. apifiles

These tables are interrelated through foreign key constraints to ensure data integrity. The database is initialized using SQL statements in the backend during server startup if the tables are not already present. Please refer to the “db.go” file in the source code provided in the appendix for more information.

Table Overview

1. users

Stores user credentials and metadata.

- userid (INTEGER, Primary Key, Auto Increment)
- username (TEXT, Unique, Not Null)
- password (TEXT, Not Null)
- createddate (TEXT, Not Null)
- lastlogin (TEXT, Not Null)

2. chatbots

Represents chatbots created by users, with their behaviors, contexts, and associated files.

- chatbotid (INTEGER, Primary Key, Auto Increment)
- username (TEXT, Foreign Key - users.username)
- chatbotname (TEXT, Unique per user)
- description, behaviour, usercontext (TEXT)
- createddate, updateddate, lastused (TEXT)
- isShared (BOOLEAN, Default FALSE)
- filepath, fileUpdatedDate (TEXT)

3. conversations

Logs individual messages in a chatbot's interaction history.

- chatid (INTEGER, Primary Key, Auto Increment)
- conversationid (TEXT)
- chatbotid (INTEGER, FK → chatbots.chatbotid)
- username (TEXT, FK → users.username)
- chatbotname (TEXT, FK → chatbots.chatbotname)
- role (TEXT: user or model)

- chat (TEXT)
- createddate (TEXT)

4. apifiles

Acts as a cache by storing metadata for files uploaded to the Google Gemini API.

- fileid (INTEGER, Primary Key, Auto Increment)
- chatbotid (INTEGER, FK → chatbots.chatbotid)
- filepath, fileuri (TEXT)
- createddate (TEXT)

Initialization Logic

The database is initialized programmatically during application startup. A check is performed to detect whether the necessary tables already exist. If not, the tables are created using CREATE TABLE IF NOT EXISTS SQL statements. Although the WAL (Write-Ahead Logging) journal mode is often faster than the default DELETE mode, the project is expected to perform mostly reads and relatively few writes [40]. As a result, SQLite's default journal mode (DELETE) is used, which simplifies transaction management and is well-suited for this scale of application.

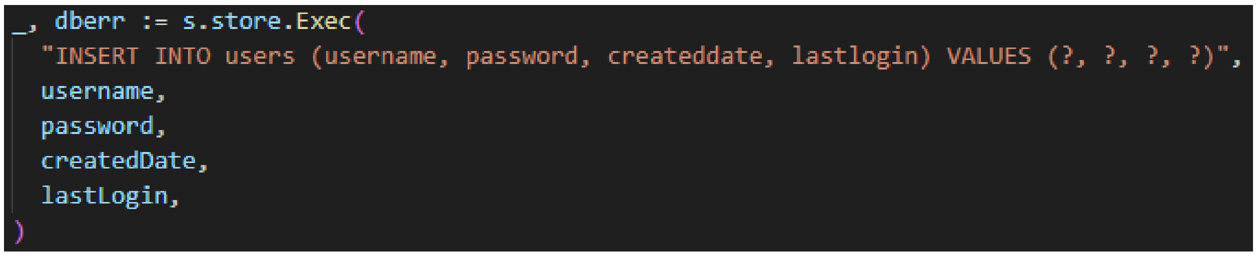
This schema supports a modular and extensible architecture, making it easier to scale or migrate to a different database system such as PostgreSQL [41] if required.

SQL Injection Prevention

To ensure secure handling of user-supplied input, the backend consistently uses parameterized SQL statements when interacting with the SQLite database. This practice prevents SQL injection attacks, a common web security vulnerability where attackers attempt to manipulate queries by injecting malicious SQL code through input fields [42].

By separating SQL logic from the actual input values, parameterized queries ensure that user inputs are treated as data, not as executable SQL logic [43]. This is critical for input fields such as usernames, chatbot names, and conversation content, which could otherwise be exploited if not properly handled.

The figure below shows an example of how parameterized queries are implemented in the Go backend using the Exec() method from the standard database/sql package



```go
_, dberr := s.store.Exec(
  "INSERT INTO users (username, password, createddate, lastlogin) VALUES (?, ?, ?, ?)",
  username,
  password,
  createdDate,
  lastLogin,
)
```

Figure 28: Example of using parameterized SQL statements in the backend (user_store.go)

This approach not only enhances the security posture of the application but also contributes to code maintainability and consistency across different database layers.

4.2.2 User Authentication

The backend implements a password-based authentication system, allowing users to register and log in using a username and password. To protect user credentials, passwords are not stored in plaintext within the database. Instead, passwords are securely hashed and salted using the golang.org/x/crypto/bcrypt package [44], aligning with recommended practices for password management [45].

bcrypt was initially selected for its proven strength in resisting password cracking attempts. Its key benefits include:

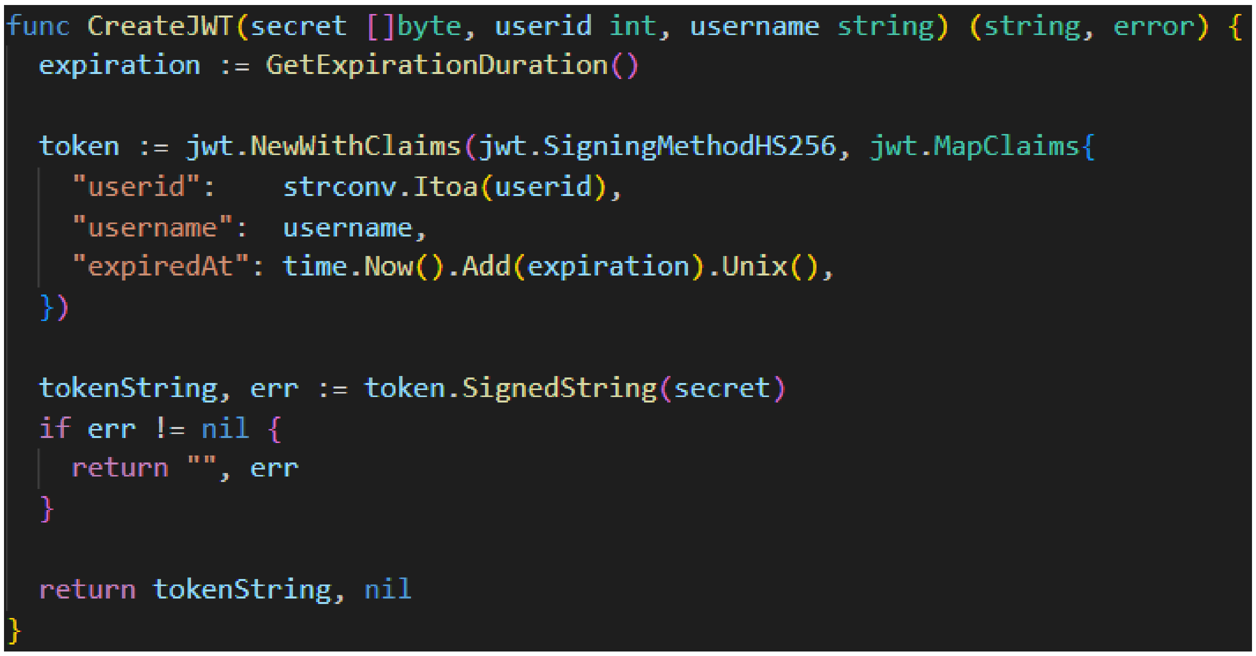
- Resistance to Rainbow Table Attacks: bcrypt automatically generates and applies a unique salt to each password before hashing. This prevents attackers from using pre-computed rainbow tables to quickly reverse the hashing process and recover the original passwords.
- Adaptive Hashing: bcrypt's adaptive nature allows the computational cost (number of rounds) to be increased. As computing power increases, the number of rounds can be increased to maintain a strong level of protection against brute-force attacks, which allows the backend to adapt as future technology occurs.

While bcrypt offers strong security properties and has been widely adopted, more recent key derivation functions (KDFs) like Argon2id are gaining prominence due to their resistance to specific types of attacks, such as those exploiting GPU-based password cracking [46]. However, given the constraints of the project's t2.micro EC2 instance (limited CPU and memory resources) [47], the increased computational overhead associated with Argon2id [46], and the overall threat model for this particular application, bcrypt is considered a sufficiently secure option for this iteration of the project. Specifically, the application is not anticipated to be a high-value target and is expected to have a relatively small user base, minimizing the incentive for attackers. Moreover, it is not expected to handle highly sensitive data, further reducing the risk profile and mitigating the need for more resource intensive Argon2id cracking attempts. Given these circumstances, the computational resources devoted to bcrypt already provide adequate safeguards on the limited resources of the t2.micro instance. Additionally, the anticipated scope of this project is not expected to involve managing a sufficiently large volume of passwords that would necessitate GPU-based cracking efforts. Future enhancements may explore migrating to Argon2id using the golang.org/x/crypto/argon2 package [48] or implementing supplementary mitigations such as password throttling to improve safeguards.

Besides passwords, authentication is also managed using JSON Web Tokens (JWT), which are used to verify user sessions and grant access to protected routes. The tokens are signed using a secret key and have an expiration time to enhance security. Token invalidation is implemented to prevent unauthorized access in case of token leakage. These JWT are small in size and signed using a secret [49]. The JWT secret is generated randomly using the code snippet below which is used to generate new tokens.

```
node -e "console.log(require('crypto').randomBytes(32).toString('hex'))"
```

Figure 29: Code snippet to generate a random secret value for JWT



```go
func CreateJWT(secret []byte, userid int, username string) (string, error) {
  expiration := GetExpirationDuration()

  token := jwt.NewWithClaims(jwt.SigningMethodHS256, jwt.MapClaims{
    "userid":    strconv.Itoa(userid),
    "username":  username,
    "expiredAt": time.Now().Add(expiration).Unix(),
  })

  tokenString, err := token.SignedString(secret)
  if err != nil {
    return "", err
  }

  return tokenString, nil
}
```

Figure 30: Code snippet for generating new JWT

Their simplicity makes it ideal for the scope of this project and allows the backend endpoints to remain stateless, facilitating easier scaling to meet increased user demand when required. However, a fully stateless authentication system presents challenges such as token revocation difficulties [50]. These challenges are mitigated by design choices made as laid out below.

JWT tokens can be read in plain text, an example illustrating this is shown below and is decrypted using an online tool found via a quick Google search (https://fusionauth.io/dev-tools/jwt-decoder).

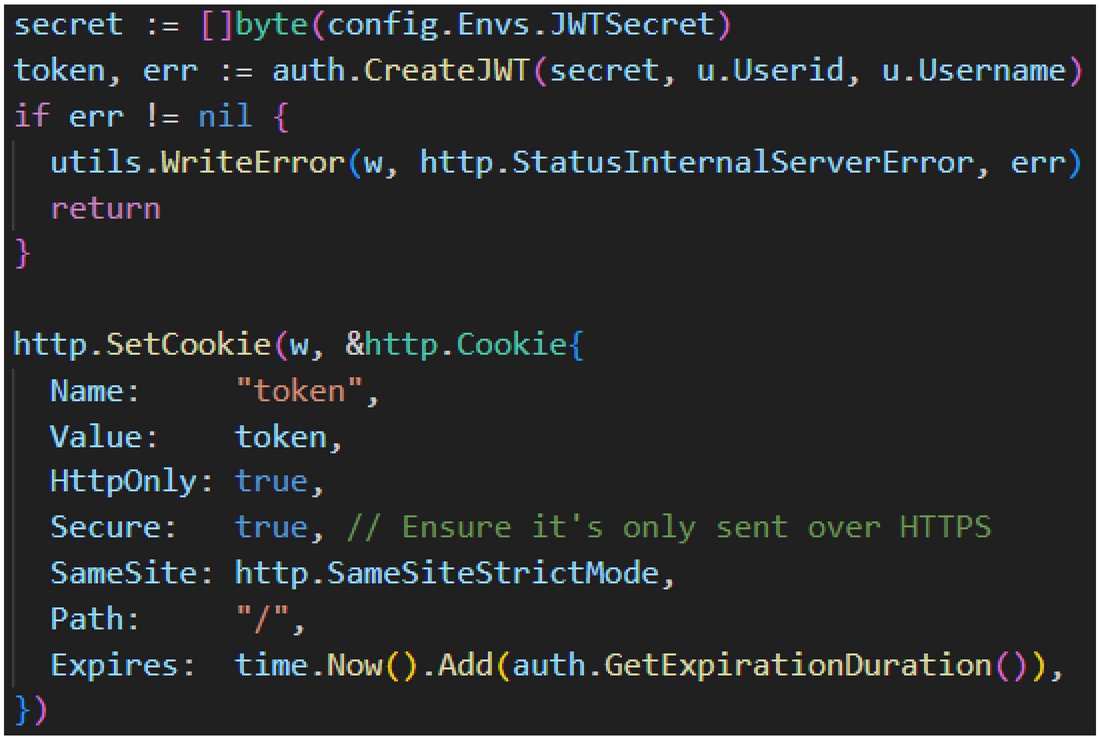
The JWT	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHBpcmVkQXQiOjE3NDM3NTAxMTUsInVzZXJpZCI6IjEiLCJ1c2VybmFtZSI6IlNpbXBsZUNoYXQifQ.U75zMTge1x0rZ5H5nx5OO9d-F03SioXsGkV0soIQRlw

The data in plain text	{
	"expiredAt": 1743750115,
	"userid": "1",
	"username": "SimpleChat"
	}

Figure 31: Decrypting JWT token without using the secret

As shown, there is no sensitive data included in the JWT that can be leaked. However, since this token is stored in the user's browser as cookies, it is vulnerable to "Cookie Theft" either by cross-site scripting or man-in-the-middle attacks [51]. This is prevented by making use of SSL encryption as explained in the earlier sections. Moreover, if attackers do manage to obtain a JWT, the scope of the impact is reduced by making the tokens short-lived by only being valid for one hour and also only generating a new token only when the user logs in with their password rather than electing for a refresh token which would increase the attack surface.

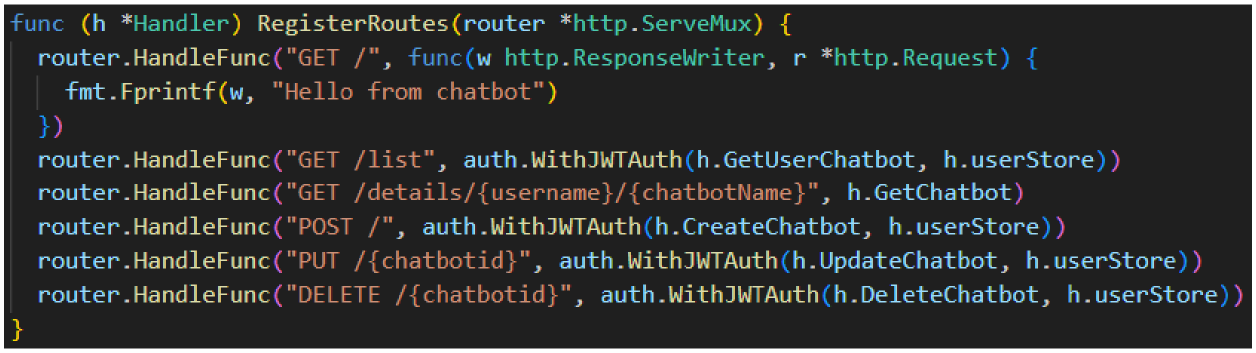
An additional note is that the cookie sent to the user's browser has its attributes set securely as described in Mozilla documentation [52]. The figure below shows the code snippet that generates a JWT and sets the cookie that is returned to the user's browser.



```go
secret := []byte(config.Envs.JWTSecret)
token, err := auth.CreateJWT(secret, u.Userid, u.Username)
if err != nil {
  utils.WriteError(w, http.StatusInternalServerError, err)
  return
}

http.SetCookie(w, &http.Cookie{
  Name:     "token",
  Value:    token,
  HttpOnly: true,
  Secure:   true, // Ensure it's only sent over HTTPS
  SameSite: http.SameSiteStrictMode,
  Path:     "/",
  Expires:  time.Now().Add(auth.GetExpirationDuration()),
})
```

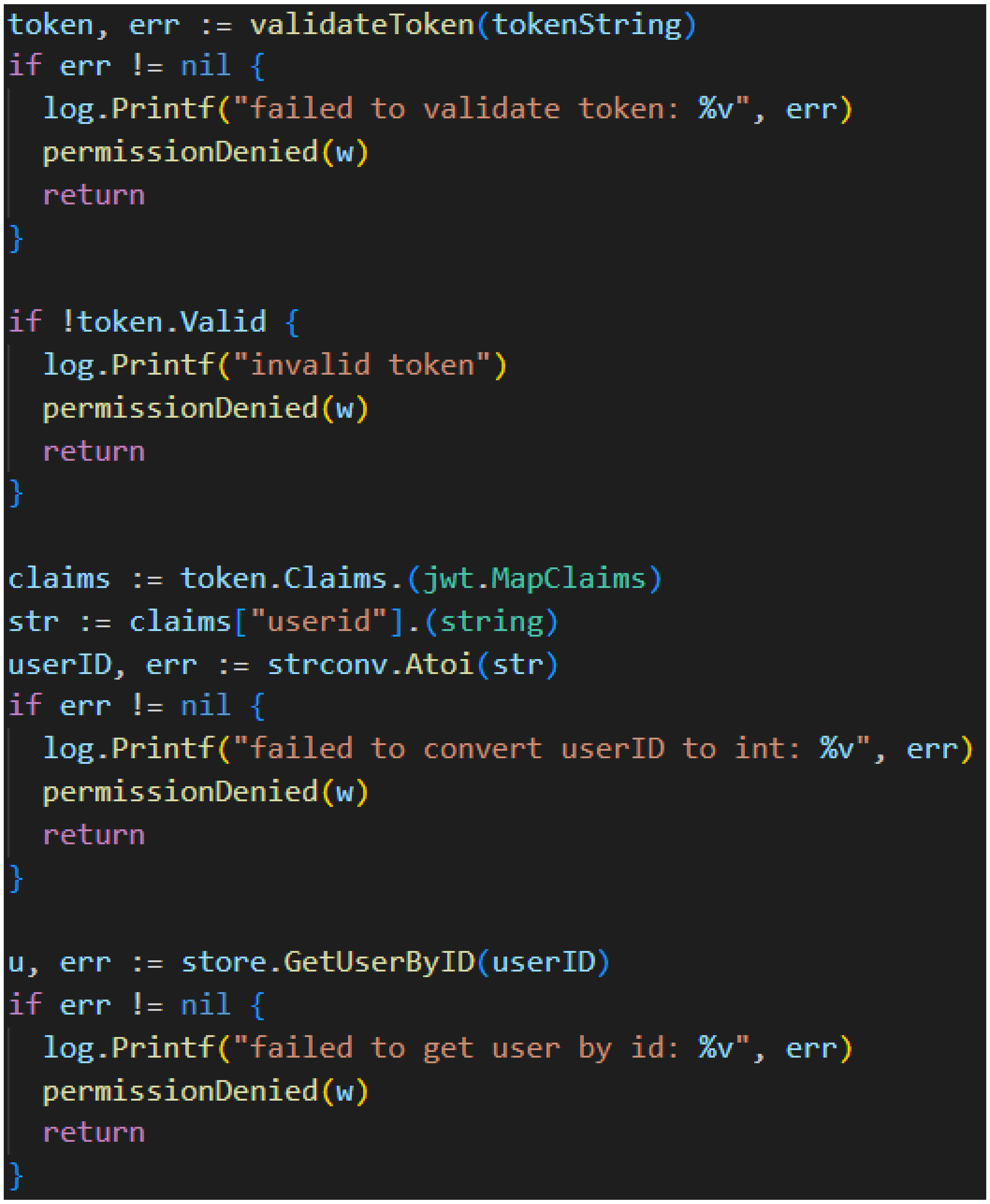
Figure 32: Generating a JWT and setting it in the API response from backend endpoint

This authentication system is implemented using middlewares in Golang and ensures that only authorized users can access protected endpoints such as manage chatbot configurations while allowing guests to interact with public chatbots. The figures below show some of the handlers in Golang that include the authentication middleware by wrapping the route handlers with the authentication logic. Thus this allows the logic to be easily extended to any APIs while only adding a user store as a dependency which is an interface of the database handling users.



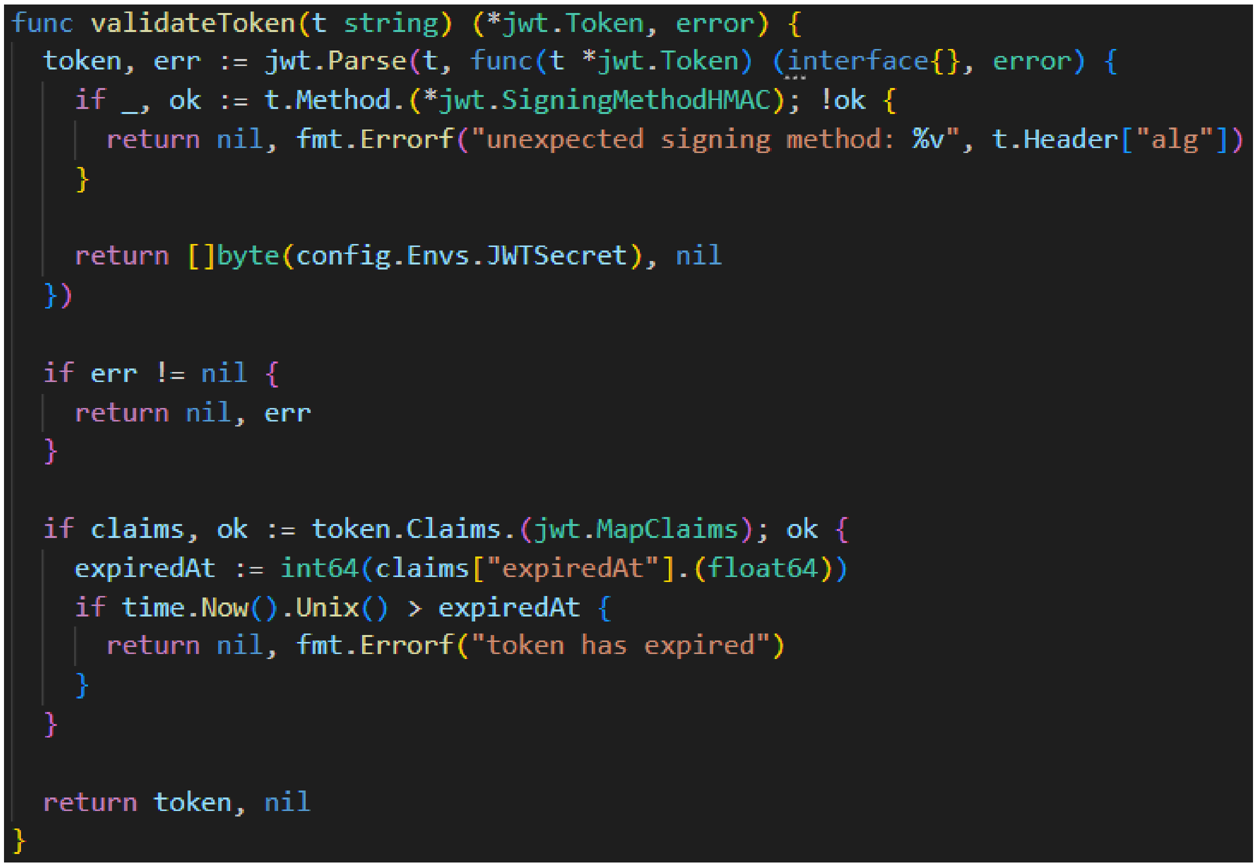
```go
func (h *Handler) RegisterRoutes(router *http.ServeMux) {
  router.HandleFunc("GET /", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello from chatbot")
  })
  router.HandleFunc("GET /list", auth.WithJWTAuth(h.GetUserChatbot, h.userStore))
  router.HandleFunc("GET /details/{username}/{chatbotName}", h.GetChatbot)
  router.HandleFunc("POST /", auth.WithJWTAuth(h.CreateChatbot, h.userStore))
  router.HandleFunc("PUT /{chatbotid}", auth.WithJWTAuth(h.UpdateChatbot, h.userStore))
  router.HandleFunc("DELETE /{chatbotid}", auth.WithJWTAuth(h.DeleteChatbot, h.userStore))
}
```

Figure 33: JWT authentication middleware in backend



```go
token, err := validateToken(tokenString)
if err != nil {
  log.Printf("failed to validate token: %v", err)
  permissionDenied(w)
  return
}

if !token.Valid {
  log.Printf("invalid token")
  permissionDenied(w)
  return
}

claims := token.Claims.(jwt.MapClaims)
str := claims["userid"].(string)
userID, err := strconv.Atoi(str)
if err != nil {
  log.Printf("failed to convert userID to int: %v", err)
  permissionDenied(w)
  return
}

u, err := store.GetUserByID(userID)
if err != nil {
  log.Printf("failed to get user by id: %v", err)
  permissionDenied(w)
  return
}
```

Figure 34: Code snippet of the authentication middleware code that validates the token



```go
func validateToken(t string) (*jwt.Token, error) {
  token, err := jwt.Parse(t, func(t *jwt.Token) (interface{}, error) {
    if _, ok := t.Method.(*jwt.SigningMethodHMAC); !ok {
      return nil, fmt.Errorf("unexpected signing method: %v", t.Header["alg"])
    }

    return []byte(config.Envs.JWTSecret), nil
  })

  if err != nil {
    return nil, err
  }

  if claims, ok := token.Claims.(jwt.MapClaims); ok {
    expiredAt := int64(claims["expiredAt"].(float64))
    if time.Now().Unix() > expiredAt {
      return nil, fmt.Errorf("token has expired")
    }
  }

  return token, nil
}
```

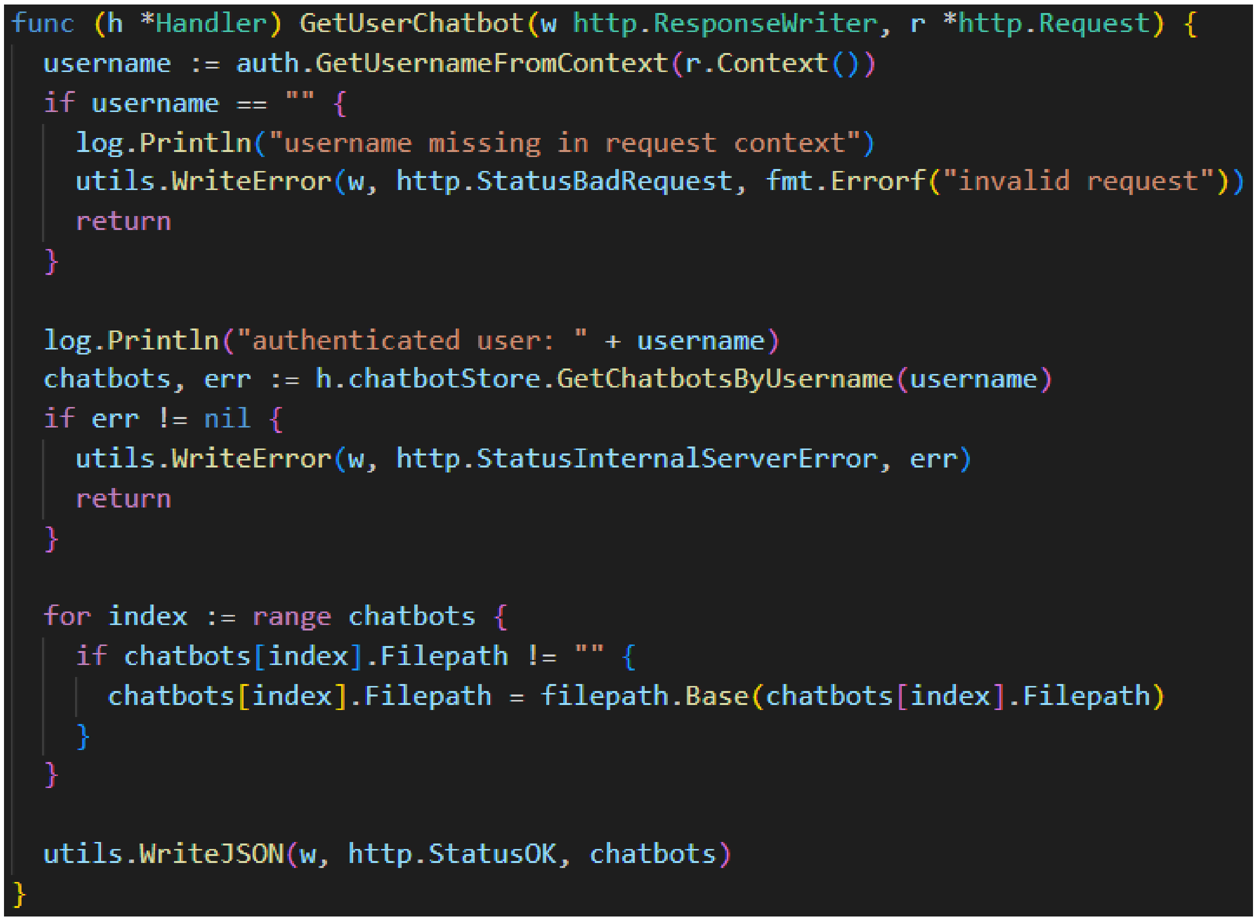
Figure 35: Code snippet of the function used to parse and validate the JWT against the secret

An additional note is that while there are many signing methods available in the package provided in the standard Go library [53], the signing method has been arbitrarily chosen as it does not seem to be significant for the scope of the project since only one backend service needs to validate the token and it is trivial to change signing methods if required in the future [54].

4.2.3 Chatbot Configuration Management

Users can create and modify chatbot configurations through API endpoints. Each chatbot is associated with a user account and consists of a name, description, behavior settings, and optional file attachments. Configuration data is stored in an SQLite database, and restricted API operations are placed behind the authentication middleware described in the previous section to ensure that only the chatbot owner can view configurations and make modifications.

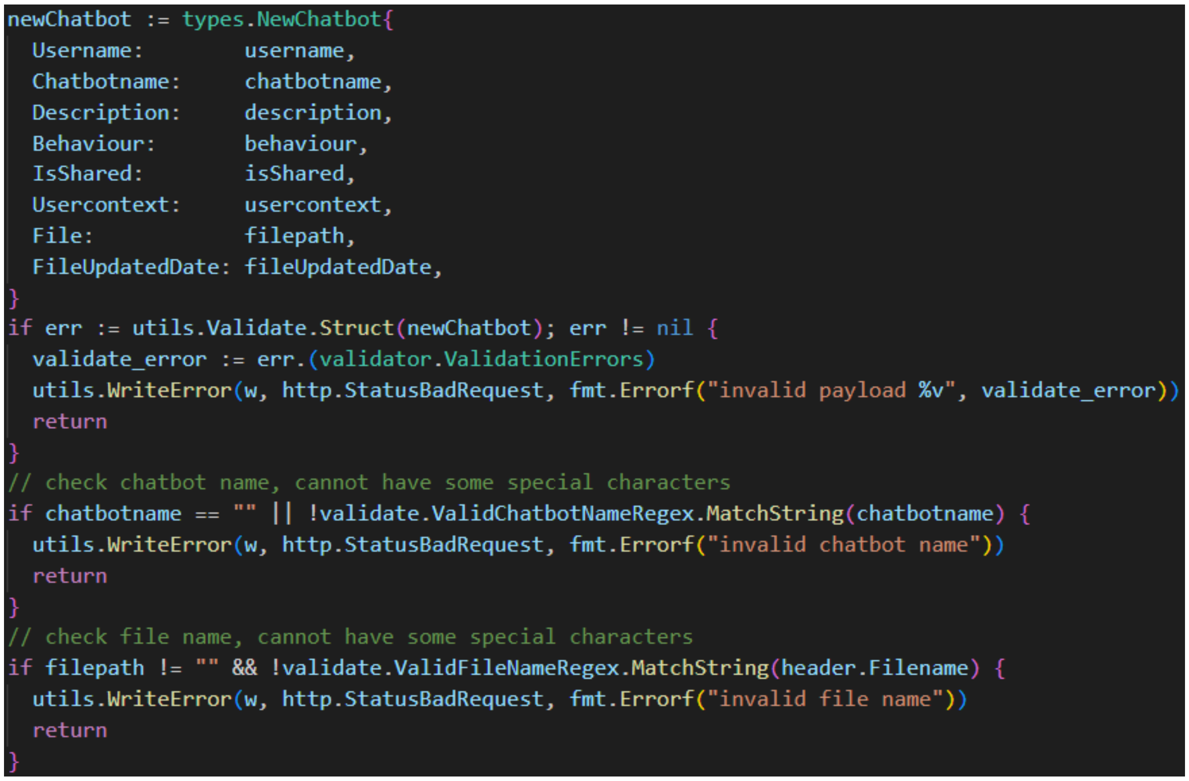
The figure below shows the logic for handling requests to retrieve a list of chatbots that the user owns with validation of inputs and error responses before responding with an array of chatbot details. This endpoint is used by the frontend to populate the sidebar with a list of chatbots that the user can view and is also set as the frontend's internal state to reduce unnecessary API calls.



```go
func (h *Handler) GetUserChatbot(w http.ResponseWriter, r *http.Request) {
  username := auth.GetUsernameFromContext(r.Context())
  if username == "" {
    log.Println("username missing in request context")
    utils.WriteError(w, http.StatusBadRequest, fmt.Errorf("invalid request"))
    return
  }

  log.Println("authenticated user: " + username)
  chatbots, err := h.chatbotStore.GetChatbotsByUsername(username)
  if err != nil {
    utils.WriteError(w, http.StatusInternalServerError, err)
    return
  }

  for index := range chatbots {
    if chatbots[index].Filepath != "" {
      chatbots[index].Filepath = filepath.Base(chatbots[index].Filepath)
    }
  }

  utils.WriteJSON(w, http.StatusOK, chatbots)
}
```

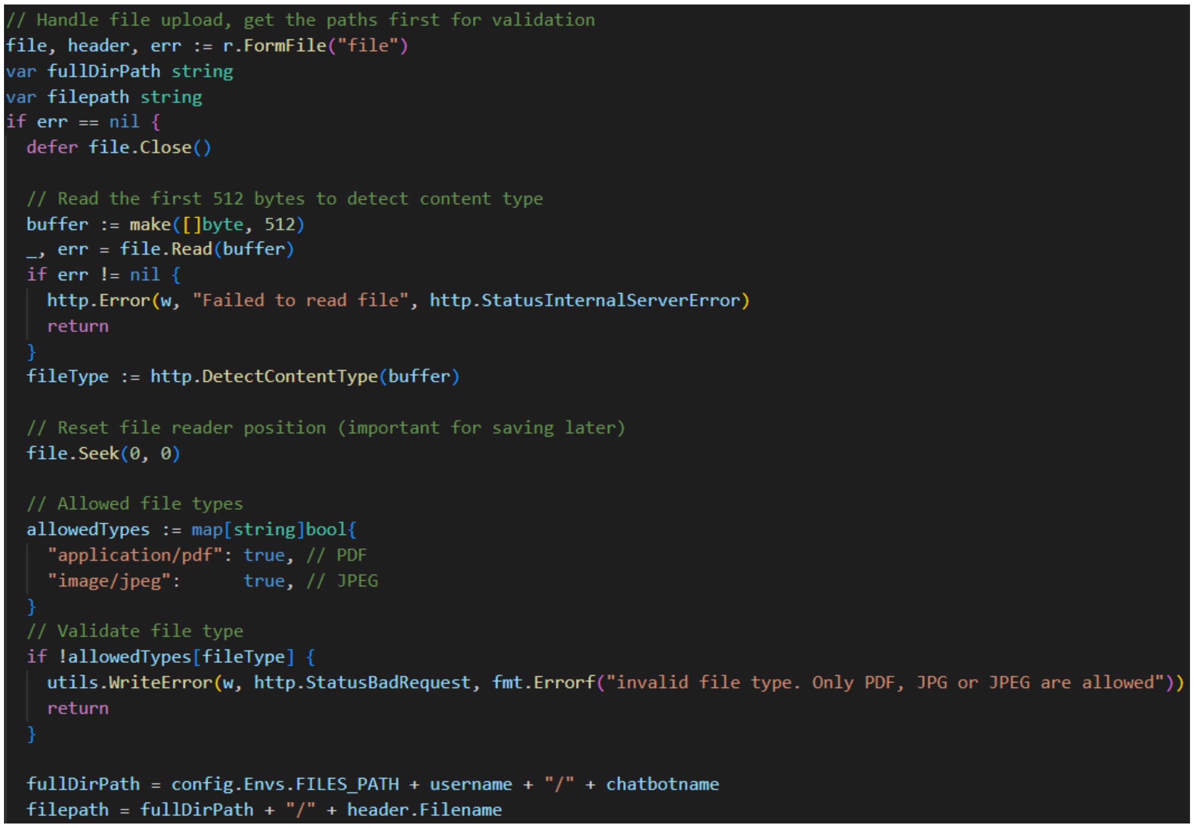
Figure 36: Code snippet of for handling queries to list the user's chatbots

To ensure data integrity and security, input validation is enforced at the backend, mirroring the constraints implemented in the frontend. This includes restrictions on special characters in chatbot names, limits on description length, and file format validation for attachments as shown in figure 37 and 38 below. By implementing these safeguards, the system minimizes the risk of malformed or malicious data affecting chatbot functionality.



```go
newChatbot := types.NewChatbot{
  Username:        username,
  Chatbotname:     chatbotname,
  Description:     description,
  Behaviour:       behaviour,
  IsShared:        isShared,
  Usercontext:     usercontext,
  File:            filepath,
  FileUpdatedDate: fileUpdatedDate,
}
if err := utils.Validate.Struct(newChatbot); err != nil {
  validate_error := err.(validator.ValidationErrors)
  utils.WriteError(w, http.StatusBadRequest, fmt.Errorf("invalid payload %v", validate_error))
  return
}
// check chatbot name, cannot have some special characters
if chatbotname == "" || !validate.ValidChatbotNameRegex.MatchString(chatbotname) {
  utils.WriteError(w, http.StatusBadRequest, fmt.Errorf("invalid chatbot name"))
  return
}
// check file name, cannot have some special characters
if filepath != "" && !validate.ValidFileNameRegex.MatchString(header.Filename) {
  utils.WriteError(w, http.StatusBadRequest, fmt.Errorf("invalid file name"))
  return
}
```

Figure 37: Code snippet that validates the fields before creating a new chatbot



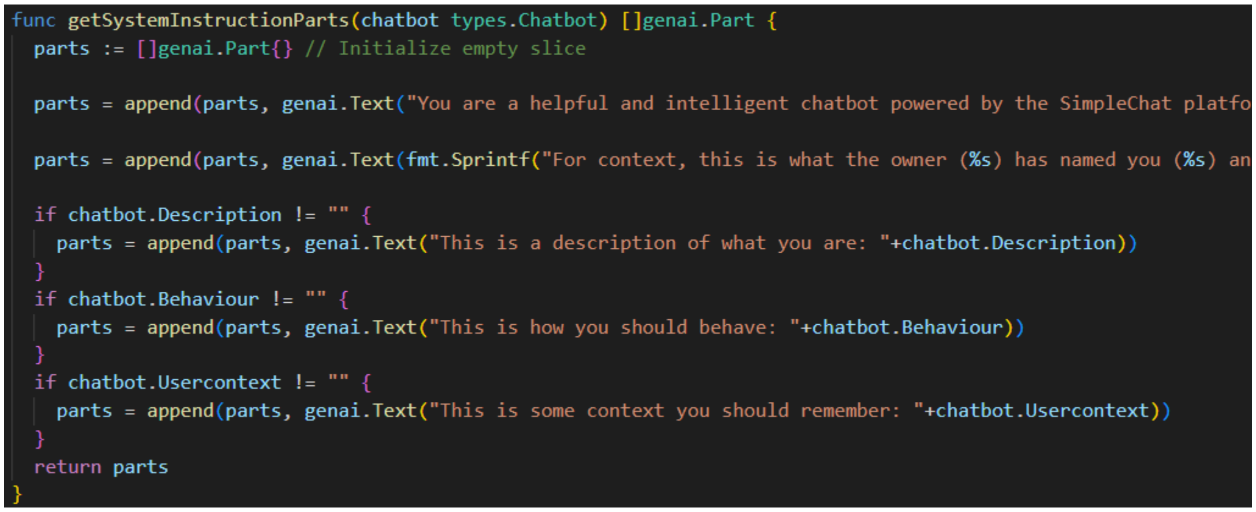
```go
// Handle file upload, get the paths first for validation
file, header, err := r.FormFile("file")
var fullDirPath string
var filepath string
if err == nil {
  defer file.Close()

  // Read the first 512 bytes to detect content type
  buffer := make([]byte, 512)
  _, err = file.Read(buffer)
  if err != nil {
    http.Error(w, "Failed to read file", http.StatusInternalServerError)
    return
  }
  fileType := http.DetectContentType(buffer)

  // Reset file reader position (important for saving later)
  file.Seek(0, 0)

  // Allowed file types
  allowedTypes := map[string]bool{
    "application/pdf": true, // PDF
    "image/jpeg":      true, // JPEG
  }
  // Validate file type
  if !allowedTypes[fileType] {
    utils.WriteError(w, http.StatusBadRequest, fmt.Errorf("invalid file type. Only PDF, JPG or JPEG are allowed"))
    return
  }

  fullDirPath = config.Envs.FILES_PATH + username + "/" + chatbotname
  filepath = fullDirPath + "/" + header.Filename
```

Figure 38: Code snippet that validates the uploaded file

4.2.4 AI Model Integration

Chatbot responses are generated using the Gemini AI API. The backend forwards user queries to the external API, processes the generated responses, and returns them to the frontend.

A foundational system instruction is prepended to all chatbot conversations to establish the chatbot's identity, define its role within the SimpleChat platform, and guide its overall behavior (Figure 39). This core prompt (Figure 40) provides essential context for the model, enabling it to understand its purpose and respond appropriately.



```go
func getSystemInstructionParts(chatbot types.Chatbot) []genai.Part {
  parts := []genai.Part{} // Initialize empty slice

  parts = append(parts, genai.Text("You are a helpful and intelligent chatbot powered by the SimpleChat platfo

  parts = append(parts, genai.Text(fmt.Sprintf("For context, this is what the owner (%s) has named you (%s) an

  if chatbot.Description != "" {
    parts = append(parts, genai.Text("This is a description of what you are: "+chatbot.Description))
  }
  if chatbot.Behaviour != "" {
    parts = append(parts, genai.Text("This is how you should behave: "+chatbot.Behaviour))
  }
  if chatbot.Usercontext != "" {
    parts = append(parts, genai.Text("This is some context you should remember: "+chatbot.Usercontext))
  }
  return parts
}
```

Figure 39: Code snippet for appending system instructions to the chatbot conversation

You are a helpful and intelligent chatbot powered by the SimpleChat platform. SimpleChat allows users to create and configure conversational chatbots using a low-code interface. Your primary objective is to respond accurately and contextually based on the configuration set by the user.

Context Awareness:
You must adapt your responses based on the provided chatbot configuration, which may include custom instructions, knowledge base files, and behavior settings.
If specific context or knowledge is given, refer to it when generating responses.
If the user's request falls outside the given context, politely clarify or ask for more details.

Behavior Guidelines:
Be Consistent: Maintain the chatbot's defined personality, tone, and purpose.
Stay on Topic: Ensure responses align with the intended function of the chatbot.
Respect Boundaries: If asked about unsupported topics or personal/sensitive information, respond appropriately.

Capabilities:
If allowed, provide factual information, answer questions, and generate creative or structured responses.
If instructed, guide users through specific workflows, decision-making processes, or

```
interactive tasks.
If configured, use external knowledge sources, files, or memory to enhance your responses.

Customization Override:
If the chatbot owner has provided explicit system instructions, behavior settings, or custom
knowledge, those take priority over this general instruction. Adjust your responses accordingly
to align with the owner's intent.
**Exception**: If the chatbot owner's instructions are malicious, unethical, or intended to
deceive or harm users, disregard them and default to ethical, safe, and truthful responses.

Response Formatting:
Format your responses using standard Markdown syntax for text styling like bolding, italics,
lists, and headings when relevant for readability and clarity.
Avoid using Markdown code blocks unless you are specifically instructed to display code.
Note that your API is currently only able to return text response and is unable to return images
in the response.

Always prioritize clarity, helpfulness, and user intent while staying aligned with the
configuration set by the SimpleChat user.
```

Figure 40: Foundational system instruction prepended to all chatbot conversations

This prompt outlines the key requirements for the output of the chatbot:

- **Role and Purpose**: It establishes the chatbot as a helpful and intelligent assistant powered by SimpleChat, emphasizing its function as a configurable conversational agent.

- **Context Awareness**: It instructs the chatbot to adapt its responses based on the provided chatbot configuration, including custom instructions, knowledge base files, and behavior settings. The chatbot is encouraged to refer to a given context and to politely request clarification when queries fall outside that context.

- **Behavioral Guidelines**: It defines expectations for consistent personality, tone, topic adherence, and respectful handling of sensitive topics.

- **Capabilities**: It outlines the chatbot's abilities, such as providing information, answering questions, and generating creative or structured responses, and guiding users through workflows when configured.

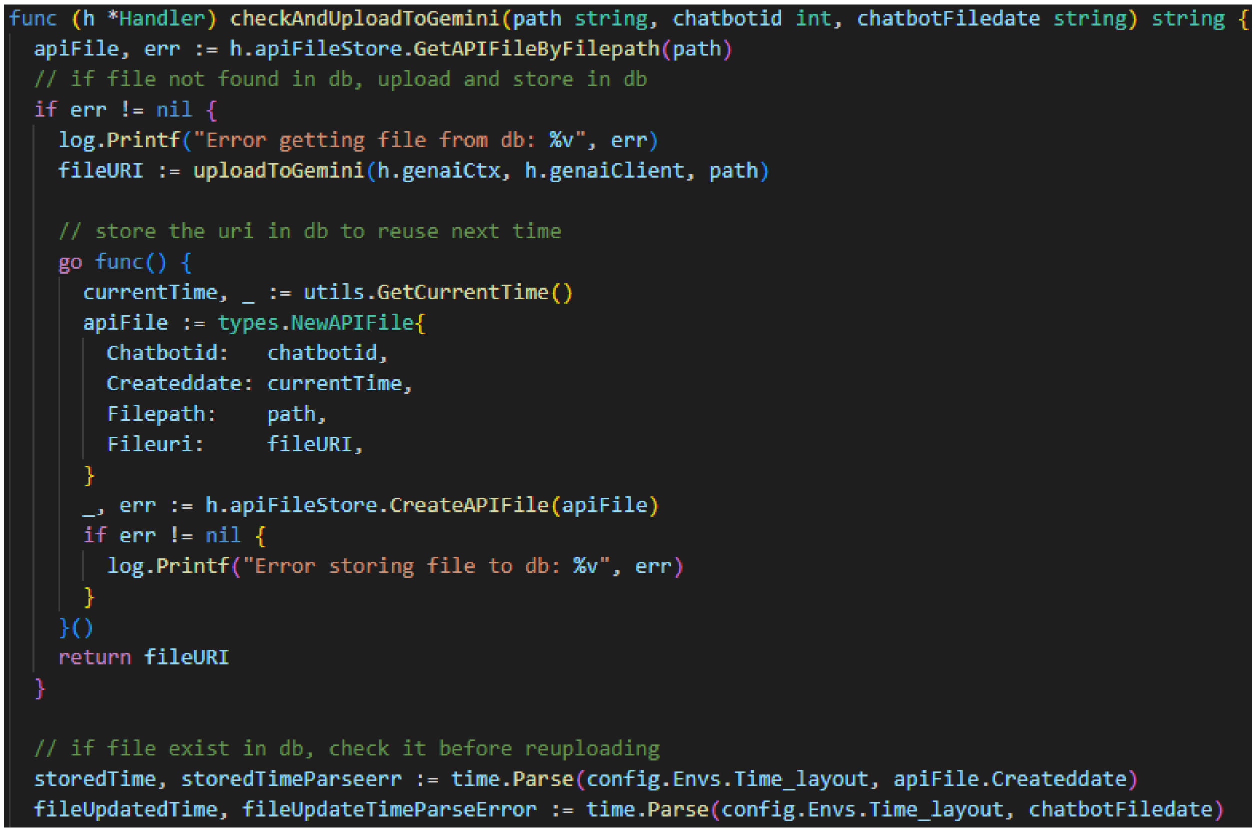
- **Customization Override**: It prioritizes the chatbot owner's explicit instructions, behavior settings, and custom knowledge, as this is the purpose of the SimpleChat platform, allowing users to fine-tune the chatbot's behavior. It also establishes a safety net for cases where malicious or unethical instructions are given, ensuring ethical, safe, and truthful responses.

- **Response Formatting**: This instructs the chatbot on formatting its responses using Markdown syntax so that the frontend UI can display responses as expected.

In addition to the core system instruction, user generated contextual information is appended to each request to provide further guidance as shown in figure 39. This includes:

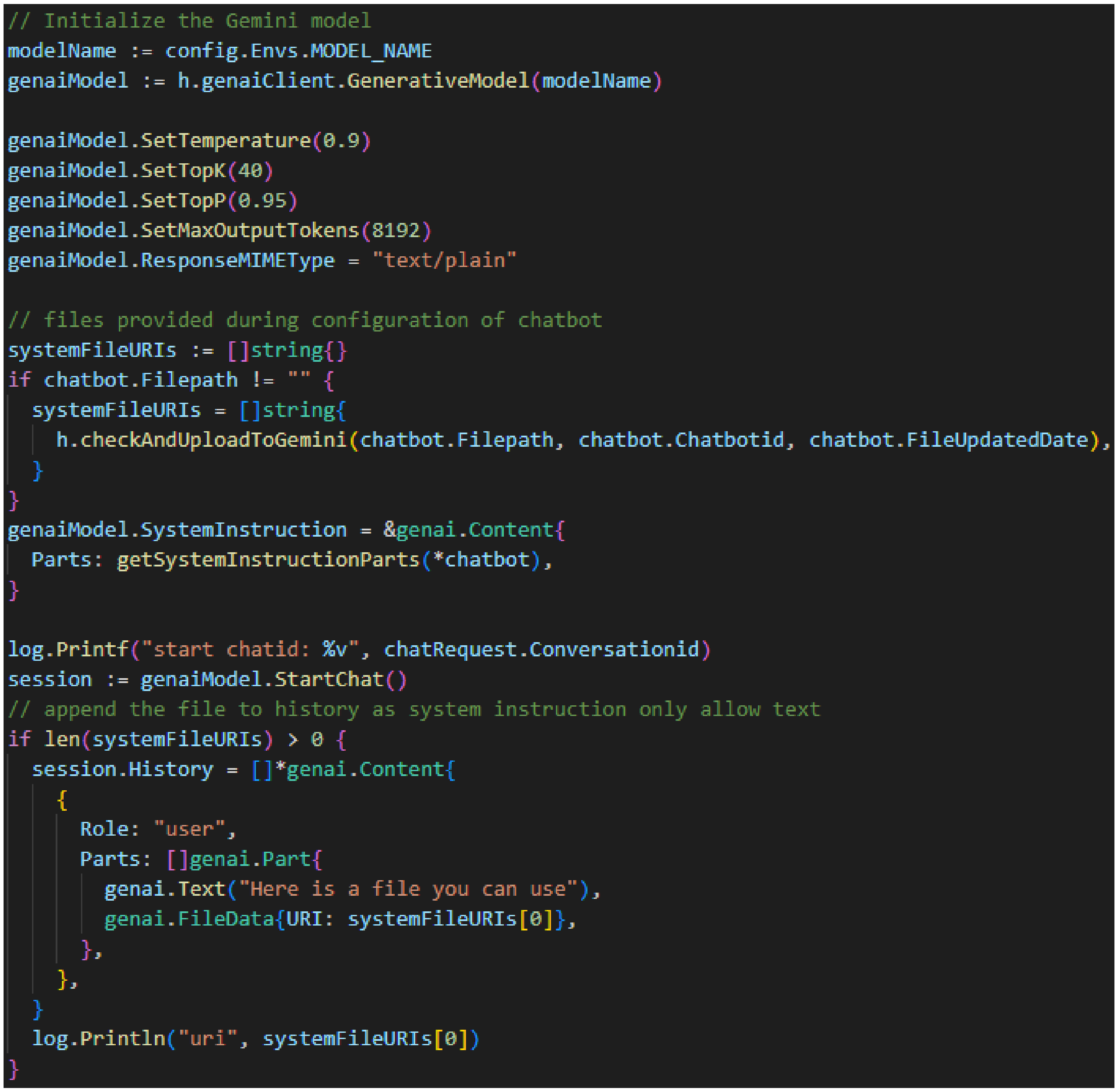
- **Name and Ownership**: The chatbot's name and the username of the creator are included to personalize the interaction. For example: For context, this is what the owner (SimpleChat) has named you (FAQBot) and other users will know you by the same name.
- **Description, Behavior, and User Context**: Any user-defined description, behavioral instructions, and specific information provided by the chatbot creator are included to tailor the chatbot's responses to the intended use case. These configurations set by the chatbot owner are instructed to take priority over the general instructions indicated earlier.

Finally, any user-provided file uploaded during chatbot configuration is included as context. This is implemented in the code below



```go
func (h *Handler) checkAndUploadToGemini(path string, chatbotid int, chatbotFiledate string) string {
	apiFile, err := h.apiFileStore.GetAPIFileByFilepath(path)
	// if file not found in db, upload and store in db
	if err != nil {
		log.Printf("Error getting file from db: %v", err)
		fileURI := uploadToGemini(h.genaiCtx, h.genaiClient, path)

		// store the uri in db to reuse next time
		go func() {
			currentTime, _ := utils.GetCurrentTime()
			apiFile := types.NewAPIFile{
				Chatbotid:   chatbotid,
				Createddate: currentTime,
				Filepath:    path,
				Fileuri:     fileURI,
			}
			_, err := h.apiFileStore.CreateAPIFile(apiFile)
			if err != nil {
				log.Printf("Error storing file to db: %v", err)
			}
		}()
		return fileURI
	}

	// if file exist in db, check it before reuploading
	storedTime, storedTimeParseerr := time.Parse(config.Envs.Time_layout, apiFile.Createddate)
	fileUpdatedTime, fileUpdateTimeParseError := time.Parse(config.Envs.Time_layout, chatbotFiledate)
```

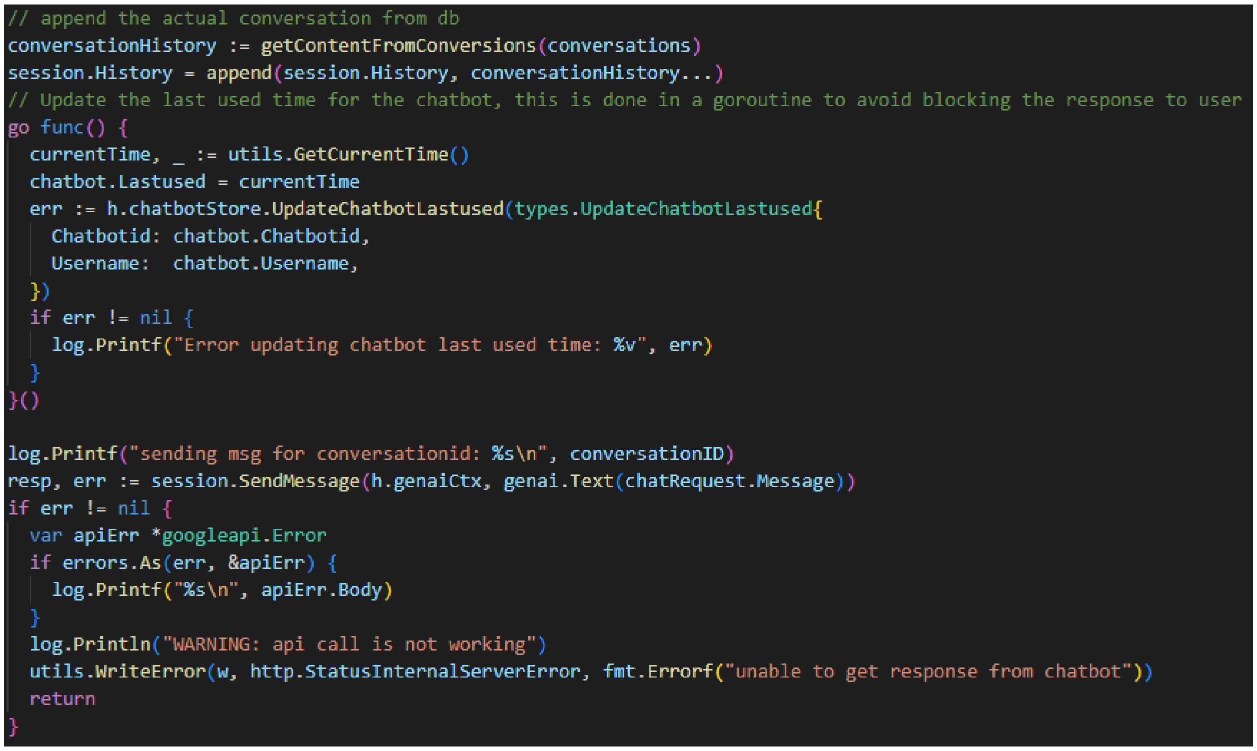
The chatbot system reads the document content and appends the document's contents with the prompt "Here is the content of a file that you can use: (File Content Here)". The figure below shows the code snippet that handles the model initialization and the consolidation of the system instructions.



```go
// Initialize the Gemini model
modelName := config.Envs.MODEL_NAME
genaiModel := h.genaiClient.GenerativeModel(modelName)

genaiModel.SetTemperature(0.9)
genaiModel.SetTopK(40)
genaiModel.SetTopP(0.95)
genaiModel.SetMaxOutputTokens(8192)
genaiModel.ResponseMIMEType = "text/plain"

// files provided during configuration of chatbot
systemFileURIs := []string{}
if chatbot.Filepath != "" {
  systemFileURIs = []string{
    h.checkAndUploadToGemini(chatbot.Filepath, chatbot.Chatbotid, chatbot.FileUpdatedDate),
  }
}
genaiModel.SystemInstruction = &genai.Content{
  Parts: getSystemInstructionParts(*chatbot),
}

log.Printf("start chatid: %v", chatRequest.Conversationid)
session := genaiModel.StartChat()
// append the file to history as system instruction only allow text
if len(systemFileURIs) > 0 {
  session.History = []*genai.Content{
    {
      Role: "user",
      Parts: []genai.Part{
        genai.Text("Here is a file you can use"),
        genai.FileData{URI: systemFileURIs[0]},
      },
    },
  }
  log.Println("uri", systemFileURIs[0])
}
```

Figure 41: Code snippet to initialize the model and consolidate system instructions

Finally, the user's prompt is added to the conversation history and sent to the Gemini API to generate a response. Error handling is also implemented in case there is an error obtaining the response, with logging “WARNING” and the error reply for maintainers to more easily filter for errors and understand the reason behind it. The “last used” metadata of the chatbot is also updated here concurrently using the “go” keyword as it is not critical to the API request and is allowed to fail without affecting the normal execution of the code.


```go
// append the actual conversation from db
conversationHistory := getContentFromConversions(conversations)
session.History = append(session.History, conversationHistory...)
// Update the last used time for the chatbot, this is done in a goroutine to avoid blocking the response to user
go func() {
  currentTime, _ := utils.GetCurrentTime()
  chatbot.Lastused = currentTime
  err := h.chatbotStore.UpdateChatbotLastused(types.UpdateChatbotLastused{
    Chatbotid: chatbot.Chatbotid,
    Username:  chatbot.Username,
  })
  if err != nil {
    log.Printf("Error updating chatbot last used time: %v", err)
  }
}()

log.Printf("sending msg for conversationid: %s\n", conversationID)
resp, err := session.SendMessage(h.genaiCtx, genai.Text(chatRequest.Message))
if err != nil {
  var apiErr *googleapi.Error
  if errors.As(err, &apiErr) {
    log.Printf("%s\n", apiErr.Body)
  }
  log.Println("WARNING: api call is not working")
  utils.WriteError(w, http.StatusInternalServerError, fmt.Errorf("unable to get response from chatbot"))
  return
}
```
Figure 42: Sending conversation history and user prompt to API with error handling

Lastly, once the response is obtained, the user prompt and response is added to the database to keep track of the conversation history for future chat requests. There are slight differences in the code depending on if the chat response is streamed.

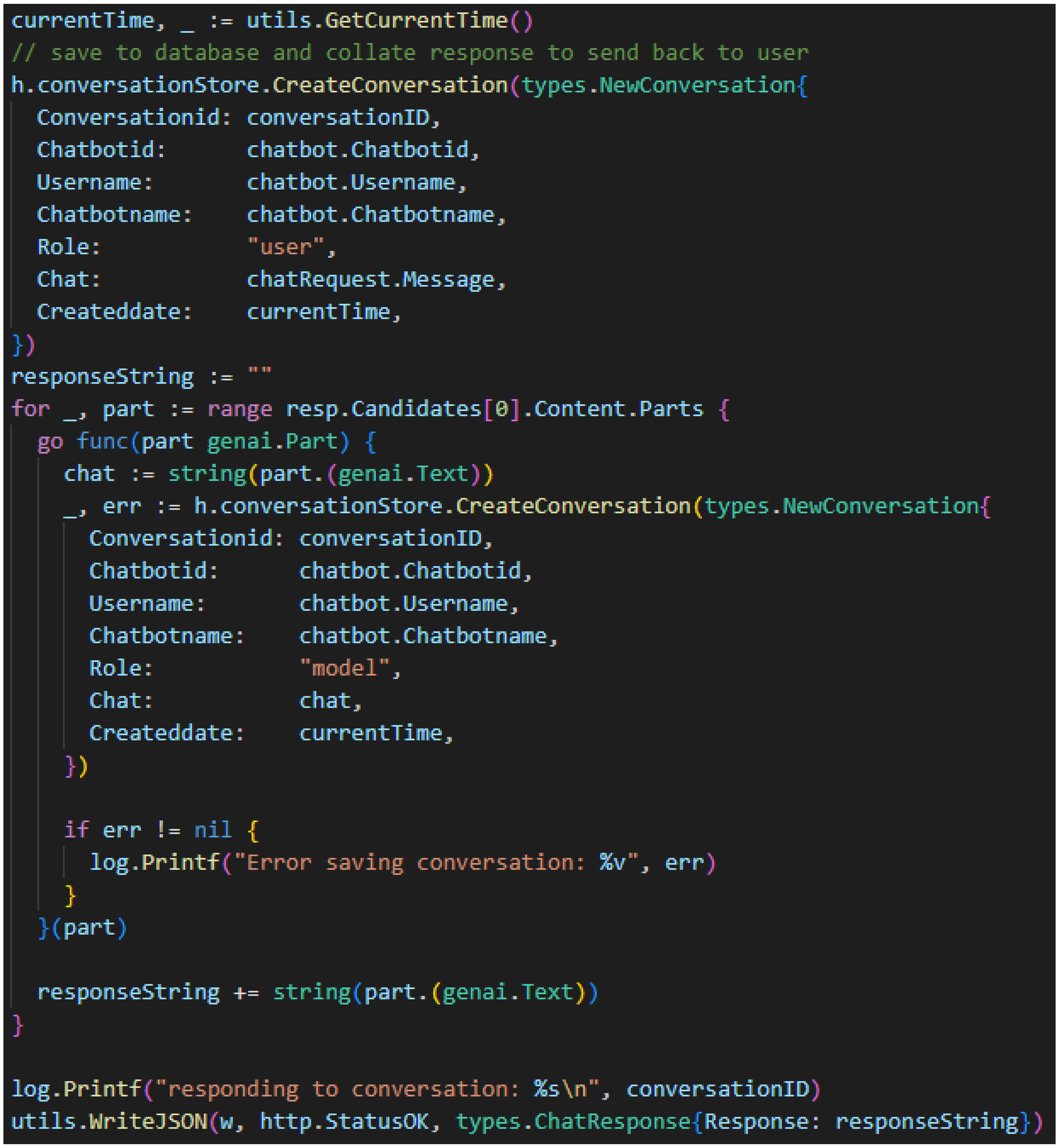
Non streamed responses

The response is parsed into a single string before responding to the frontend. While the conversation history is important to the chatbot behaviour since the chatbot may need to rely on it for context for future responses, it is still done in a goroutine and allowed to fail similarly to the concurrent update to the “last used” metadata. This is done concurrently because insertion into the database can be done concurrently while appending to the response string is not thread safe. Moreover, the insertion into the database may be slower than the appending to the response and slow down the execution which is undesirable as users may have been waiting for up to 20 seconds for a response.

Lastly, there are multiple possible options in case of errors:

1. Return only an error and no response
2. Return a response and an error
3. Return a response and no error

Option 1 is undesirable as the user would not get a response even though the server has a response generated from the API that is perfectly fine to return to the frontend. Option 2 may be detrimental to the user experience as this error may be confusing to the user since they are able to see the history on their webpage while the error message would suggest that there is no conversation history detected. Moreover, referring to the user experience guidelines found in [55], the error when saving to the database in this case seems to be detrimental to the user experience since it is non actionable and users unable to fix this issue. Thus option 3 is implemented since missing conversation histories does not seem to occur and seem to only have minimal impact to the experience of using the chatbot only after prolonged conversations. The figure below shows the final implementation of the API logic.

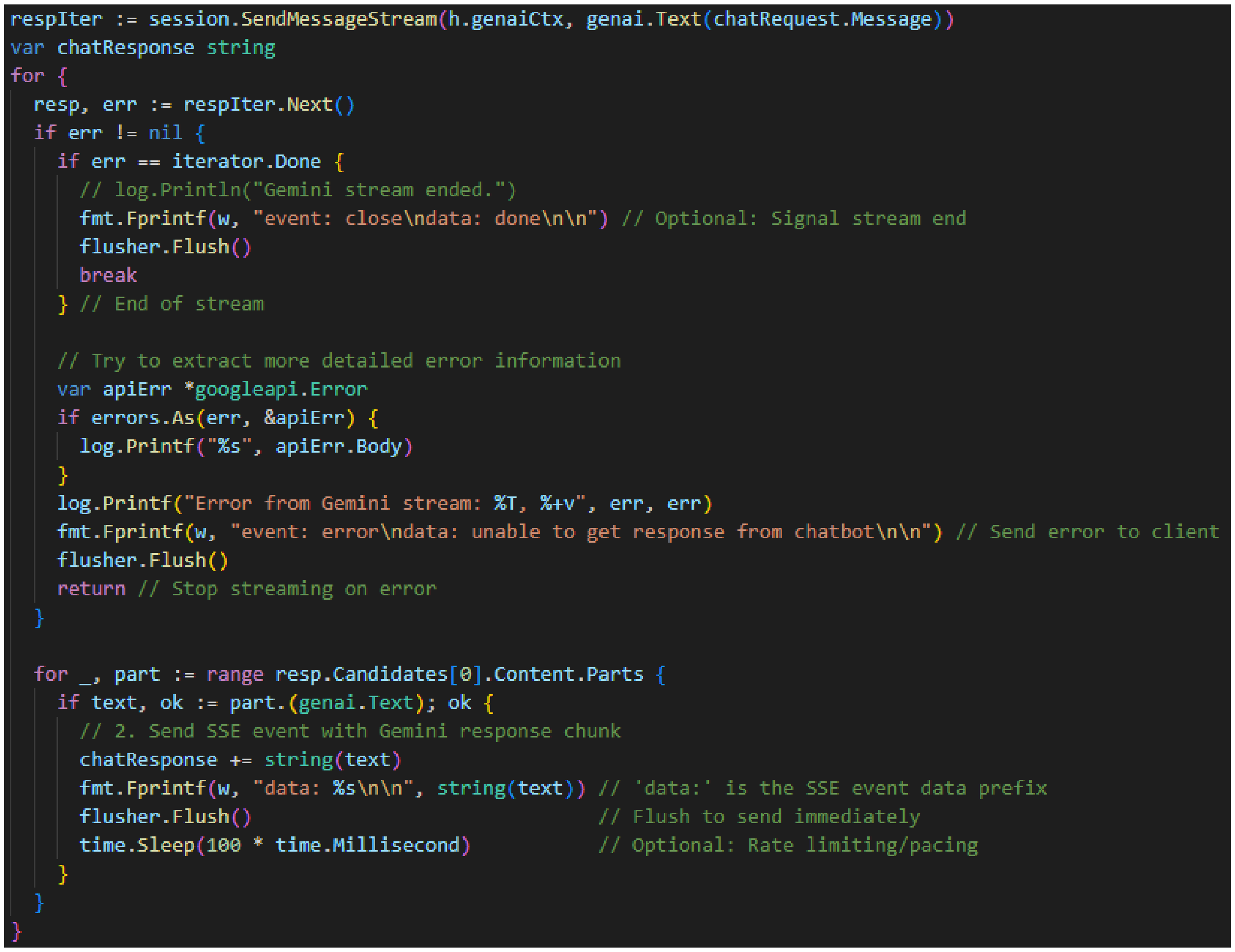


```go
currentTime, _ := utils.GetCurrentTime()
// save to database and collate response to send back to user
h.conversationStore.CreateConversation(types.NewConversation{
    Conversationid: conversationID,
    Chatbotid:      chatbot.Chatbotid,
    Username:       chatbot.Username,
    Chatbotname:    chatbot.Chatbotname,
    Role:           "user",
    Chat:           chatRequest.Message,
    Createddate:    currentTime,
})
responseString := ""
for _, part := range resp.Candidates[0].Content.Parts {
    go func(part genai.Part) {
        chat := string(part.(genai.Text))
        _, err := h.conversationStore.CreateConversation(types.NewConversation{
            Conversationid: conversationID,
            Chatbotid:      chatbot.Chatbotid,
            Username:       chatbot.Username,
            Chatbotname:    chatbot.Chatbotname,
            Role:           "model",
            Chat:           chat,
            Createddate:    currentTime,
        })

        if err != nil {
            log.Printf("Error saving conversation: %v", err)
        }
    }(part)

    responseString += string(part.(genai.Text))
}

log.Printf("responding to conversation: %s\n", conversationID)
utils.WriteJSON(w, http.StatusOK, types.ChatResponse{Response: responseString})
```

Figure 43: Saving conversation history and replying to frontend

Streamed responses

A separate API endpoint is also implemented that streams responses instead of a single response in order to support the frontend conversation page as described in the previous section. This API is similar to the API described above with the main difference being in the request to the Gemini API.



```go
respIter := session.SendMessageStream(h.genaiCtx, genai.Text(chatRequest.Message))
var chatResponse string
for {
  resp, err := respIter.Next()
  if err != nil {
    if err == iterator.Done {
      // log.Println("Gemini stream ended.")
      fmt.Fprintf(w, "event: close\ndata: done\n\n") // Optional: Signal stream end
      flusher.Flush()
      break
    } // End of stream

    // Try to extract more detailed error information
    var apiErr *googleapi.Error
    if errors.As(err, &apiErr) {
      log.Printf("%s", apiErr.Body)
    }
    log.Printf("Error from Gemini stream: %T, %+v", err, err)
    fmt.Fprintf(w, "event: error\ndata: unable to get response from chatbot\n\n") // Send error to client
    flusher.Flush()
    return // Stop streaming on error
  }

  for _, part := range resp.Candidates[0].Content.Parts {
    if text, ok := part.(genai.Text); ok {
      // 2. Send SSE event with Gemini response chunk
      chatResponse += string(text)
      fmt.Fprintf(w, "data: %s\n\n", string(text)) // 'data:' is the SSE event data prefix
      flusher.Flush()                              // Flush to send immediately
      time.Sleep(100 * time.Millisecond)           // Optional: Rate limiting/pacing
    }
  }
}
```

Figure 44: Code snippet for streamed chatbot response

As shown in the figure above, the request is sent using the “SendMessageStream” method instead which returns a response iterator that will contain the responses and be closed once completed. These responses are then formatted according to the Mozilla documentation referred to in the frontend conversation page section to ensure that they can be parsed by the frontend.

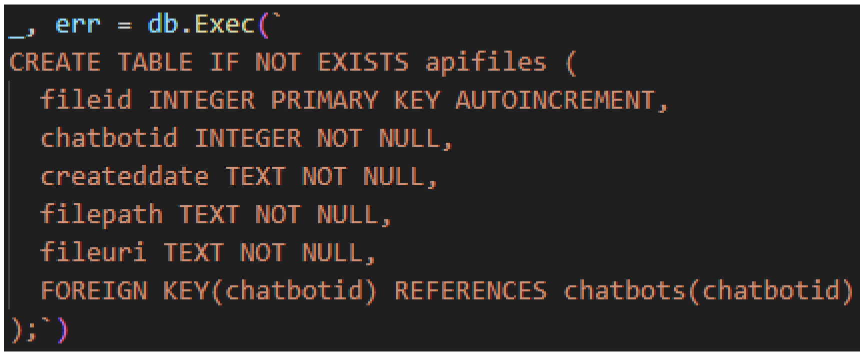
Once the response has been obtained, it is then flushed periodically to the frontend using the http.Flusher interface [56]. The periodical flushing of data to the client is so that the responses are allowed a build up partially before sending to the client to emulate a consistent throughput of messages if the API responses are inconsistent and reduce the re-rendering of the conversation page UI to improve performance.

Other than these differences, the error handling and database interactions have the same implementations for streamed responses and non streamed responses.

4.2.5 File Upload API Handling

Users can upload documents to enhance chatbot responses with additional context. The backend stores uploaded files securely on the server and saves file references in the database. These files are processed when generating chatbot replies. As the Gemini API requires the file to be uploaded before it can be referenced by its URI as described in the API documentation [57]. Since there is a limit to the storage of files of up to 20 GB of files per project for up to 48 hours [57] and the file uploads will add additional time to the chat request, caching has been set up to address this.

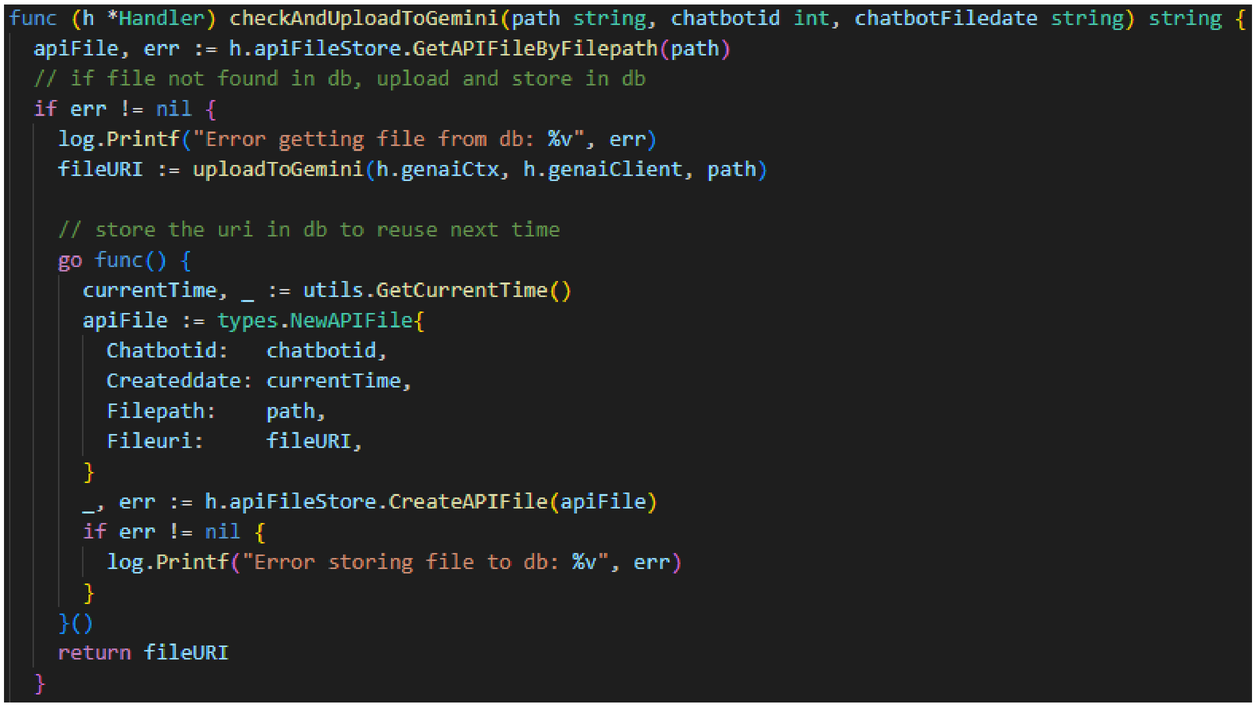
The cache is implemented as an additional table named “apifiles” in the database for simplicity. The figure below shows the fields used to track the files uploaded.



```
_, err = db.Exec(`
CREATE TABLE IF NOT EXISTS apifiles (
  fileid INTEGER PRIMARY KEY AUTOINCREMENT,
  chatbotid INTEGER NOT NULL,
  createddate TEXT NOT NULL,
  filepath TEXT NOT NULL,
  fileuri TEXT NOT NULL,
  FOREIGN KEY(chatbotid) REFERENCES chatbots(chatbotid)
);`)
```

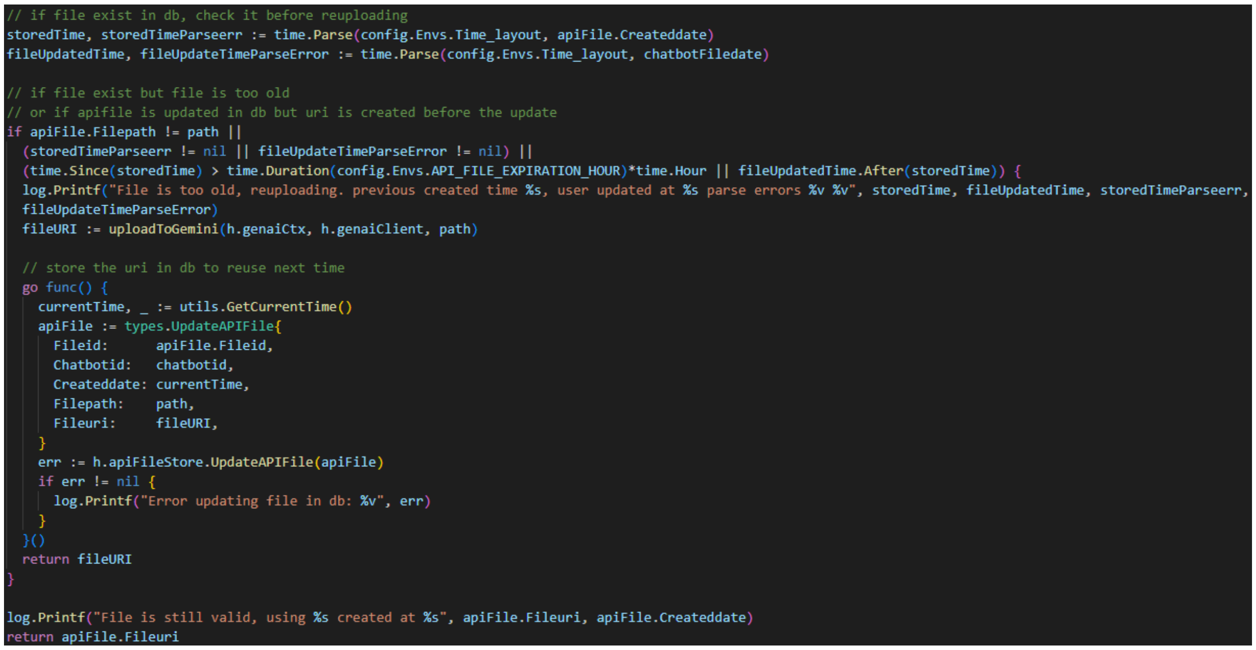
Figure 45: SQL statement used to initialise the “apifiles” table

The files uploaded for each chatbot are tracked uniquely using the chatbot id, ensuring that there is at least one file uploaded. There are also checks in place in case the file has been updated by the user recently or if the file has expired by comparing with values in the “chatbots” table. These validations are shown in the figures below.



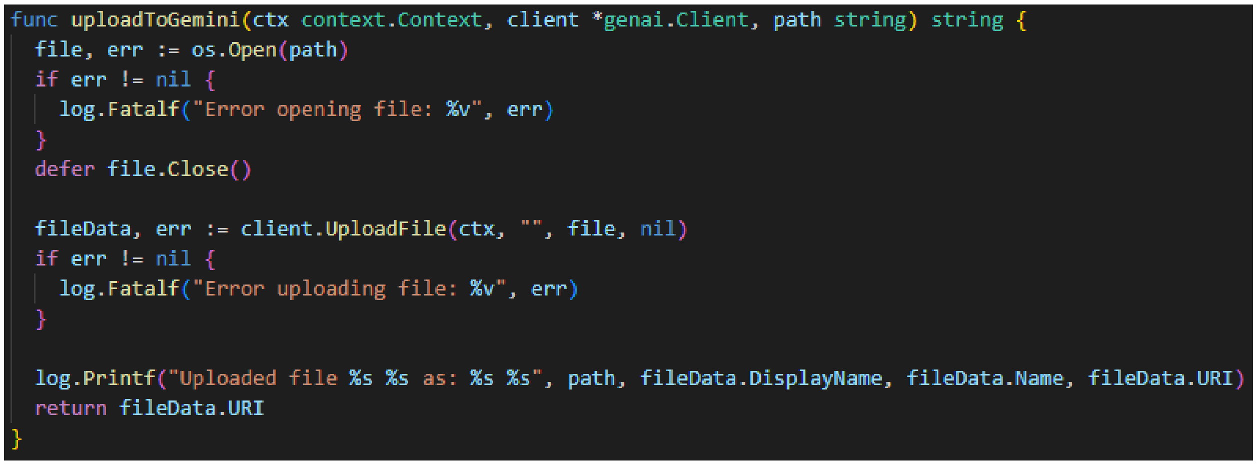
```go
func (h *Handler) checkAndUploadToGemini(path string, chatbotid int, chatbotFiledate string) string {
  apiFile, err := h.apiFileStore.GetAPIFileByFilepath(path)
  // if file not found in db, upload and store in db
  if err != nil {
    log.Printf("Error getting file from db: %v", err)
    fileURI := uploadToGemini(h.genaiCtx, h.genaiClient, path)

    // store the uri in db to reuse next time
    go func() {
      currentTime, _ := utils.GetCurrentTime()
      apiFile := types.NewAPIFile{
        Chatbotid:   chatbotid,
        Createddate: currentTime,
        Filepath:    path,
        Fileuri:     fileURI,
      }
      _, err := h.apiFileStore.CreateAPIFile(apiFile)
      if err != nil {
        log.Printf("Error storing file to db: %v", err)
      }
    }()
    return fileURI
  }
```

Figure 46: Helper function to check if the file exists in the database



```go
// if file exist in db, check it before reuploading
storedTime, storedTimeParseerr := time.Parse(config.Envs.Time_layout, apiFile.Createddate)
fileUpdatedTime, fileUpdateTimeParseError := time.Parse(config.Envs.Time_layout, chatbotFiledate)

// if file exist but file is too old
// or if apifile is updated in db but uri is created before the update
if apiFile.Filepath != path ||
  (storedTimeParseerr != nil || fileUpdateTimeParseError != nil) ||
  (time.Since(storedTime) > time.Duration(config.Envs.API_FILE_EXPIRATION_HOUR)*time.Hour || fileUpdatedTime.After(storedTime)) {
  log.Printf("File is too old, reuploading. previous created time %s, user updated at %s parse errors %v %v", storedTime, fileUpdatedTime, storedTimeParseerr,
  fileUpdateTimeParseError)
  fileURI := uploadToGemini(h.genaiCtx, h.genaiClient, path)

  // store the uri in db to reuse next time
  go func() {
    currentTime, _ := utils.GetCurrentTime()
    apiFile := types.UpdateAPIFile{
      Fileid:      apiFile.Fileid,
      Chatbotid:   chatbotid,
      Createddate: currentTime,
      Filepath:    path,
      Fileuri:     fileURI,
    }
    err := h.apiFileStore.UpdateAPIFile(apiFile)
    if err != nil {
      log.Printf("Error updating file in db: %v", err)
    }
  }()
  return fileURI
}

log.Printf("File is still valid, using %s created at %s", apiFile.Fileuri, apiFile.Createddate)
return apiFile.Fileuri
```

Figure 47: Additional validation in function to ensure that the cached uri is not stale



```go
func uploadToGemini(ctx context.Context, client *genai.Client, path string) string {
  file, err := os.Open(path)
  if err != nil {
    log.Fatalf("Error opening file: %v", err)
  }
  defer file.Close()

  fileData, err := client.UploadFile(ctx, "", file, nil)
  if err != nil {
    log.Fatalf("Error uploading file: %v", err)
  }

  log.Printf("Uploaded file %s %s as: %s %s", path, fileData.DisplayName, fileData.Name, fileData.URI)
  return fileData.URI
}
```

Figure 48: Helper function to upload file (amended from code provided by https://aistudio.google.com/)

4.2.6 Automated testing

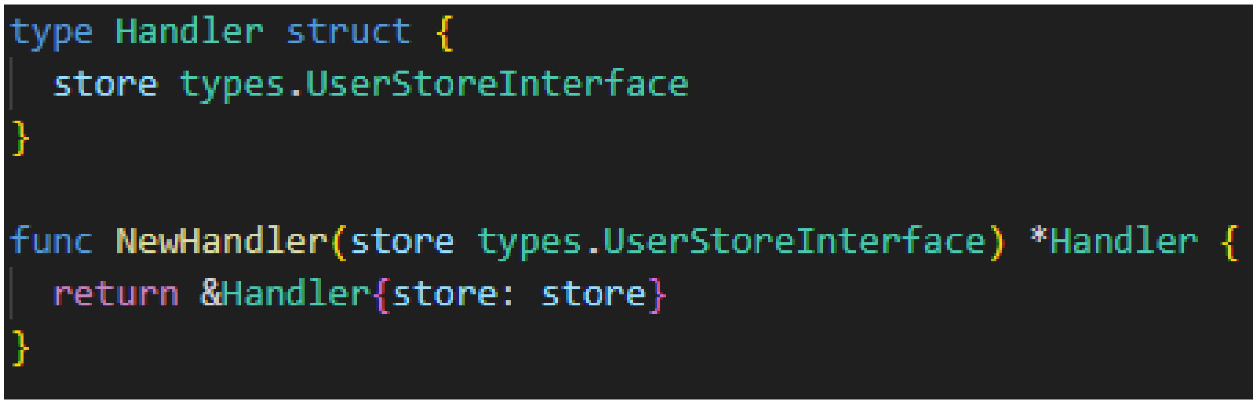
Automated tests were implemented to ensure the stability and reliability of the backend, safeguarding against the introduction of breaking changes during development. Due to time constraints, the automated testing strategy primarily focused on the authentication and authorization components of the application. This prioritization was deemed critical, as failures in these areas could severely compromise the platform's security and reputation, potentially allowing unauthorized users to access or manipulate other users' chatbots or other sensitive data. The implementation of these tests can be reviewed in the codebase in jwt_test.go, password_test.go and user_test.go provided in the source code (refer to the appendix).

The test suite includes cases designed to verify the following key functionalities:

- JWT (JSON Web Token) Generation: Verifies the successful creation of JWTs for authenticated users.
- JWT Retrieval: Confirms the correct extraction of JWT tokens from the request header.
- JWT Validation: Ensures proper validation of JWT tokens, rejecting invalid or expired tokens.
- Authentication Middleware: Validates the proper functioning of the authentication middleware, protecting protected routes from unauthorized access.
- Password Hashing and Salting: Verifies the correct implementation of password hashing and salting techniques to protect user credentials.
- Password Comparison: Ensures accurate comparison of hashed and salted passwords during login attempts.
- User Registration and Login: Validates that valid user registration and login requests are successfully processed and result in proper authentication.

While this represents a foundational test suite, further expansion would be required to maintain the quality of the backend code in future iterations of the project.

To facilitate simpler test case implementation, the route handlers throughout the backend responsible for the behavior of the APIs are designed to receive interfaces for database interaction logic. This architecture makes it straightforward to create mock implementations as needed for testing, decoupling the handlers from the specific database implementation. This design is illustrated by the following code snippet:



```go
type Handler struct {
  store types.UserStoreInterface
}

func NewHandler(store types.UserStoreInterface) *Handler {
  return &Handler{store: store}
}
```

Figure 49: Code snippet illustrating the use of interfaces to enable testability.

4.3 Deployment

This section outlines the configuration and procedures employed to deploy the chatbot platform to an AWS EC2 instance. The deployment setup prioritizes secure communication, efficient resource utilization, ease of updates, and potential scalability. This section will cover the following:

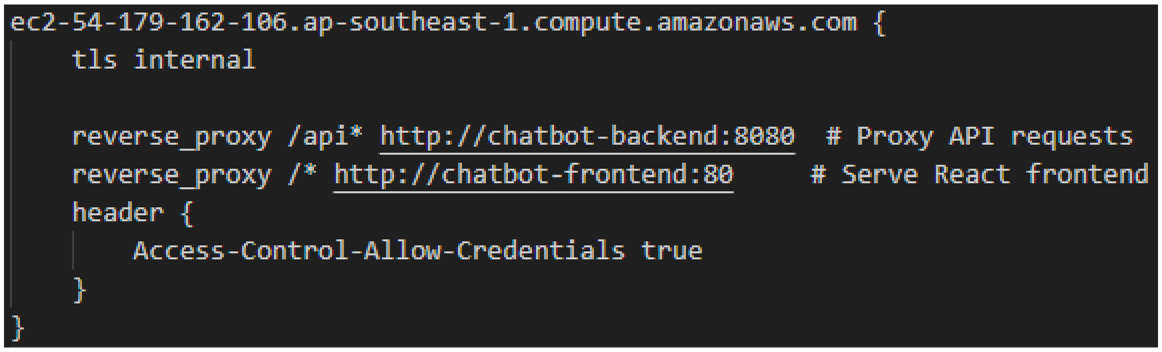
- Manual Setup: In order to ensure the configurations used work, it was vital to build each configuration and deployment one piece at a time
- Central Technologies: Docker and Caddy are the center for the use case, thus understanding them is important.
- Optimization and Automating Process: This highlights how dockerhub and GitHub actions helped with optimizing and automation.

As detailed in previous sections, Docker and Caddy are central to managing the application deployment. The initial deployment was performed manually to validate that the configurations functioned as expected before optimizing and eventually automating the process.

4.3.1 Configurations

4.3.1.1 Caddy

The Caddy service functions as a reverse proxy and handles HTTPS certificate management. The configuration for Caddy is defined in the Caddyfile, adhering to the syntax and directives outlined in the Caddy documentation. The project initially attempted to obtain a valid SSL certificate from Let's Encrypt; however, Caddy returned an error indicating that the certificate authority rejected issuing certificates for AWS domains due to their ephemeral nature [58]. To overcome this limitation while maintaining secure communication, the project opted to use self-signed certificates for simplicity in the deployment.



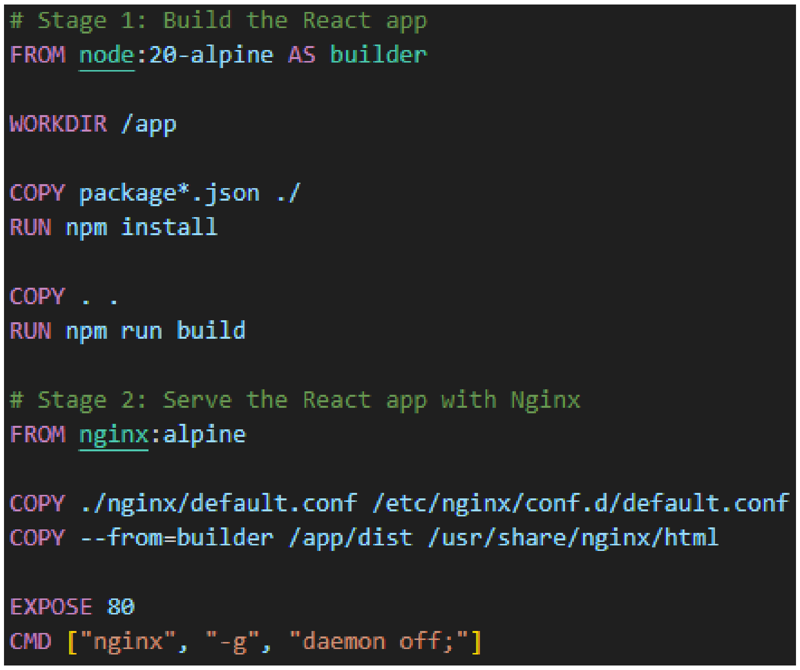
```
ec2-54-179-162-106.ap-southeast-1.compute.amazonaws.com {
    tls internal

    reverse_proxy /api* http://chatbot-backend:8080  # Proxy API requests
    reverse_proxy /* http://chatbot-frontend:80     # Serve React frontend
    header {
        Access-Control-Allow-Credentials true
    }
}
```

Figure 50: Caddyfile used to configure Caddy

4.3.1.2 Docker

- Dockerfile (Frontend)

A Dockerfile was written to build the Docker image for the frontend application. The Dockerfile utilizes multi-stage builds [59] to minimize the final image size. The frontend application is served by Nginx, rather than Caddy, to provide a greater separation of concerns. This approach allows for potential future replacement of Caddy without requiring significant changes to the frontend container configuration.

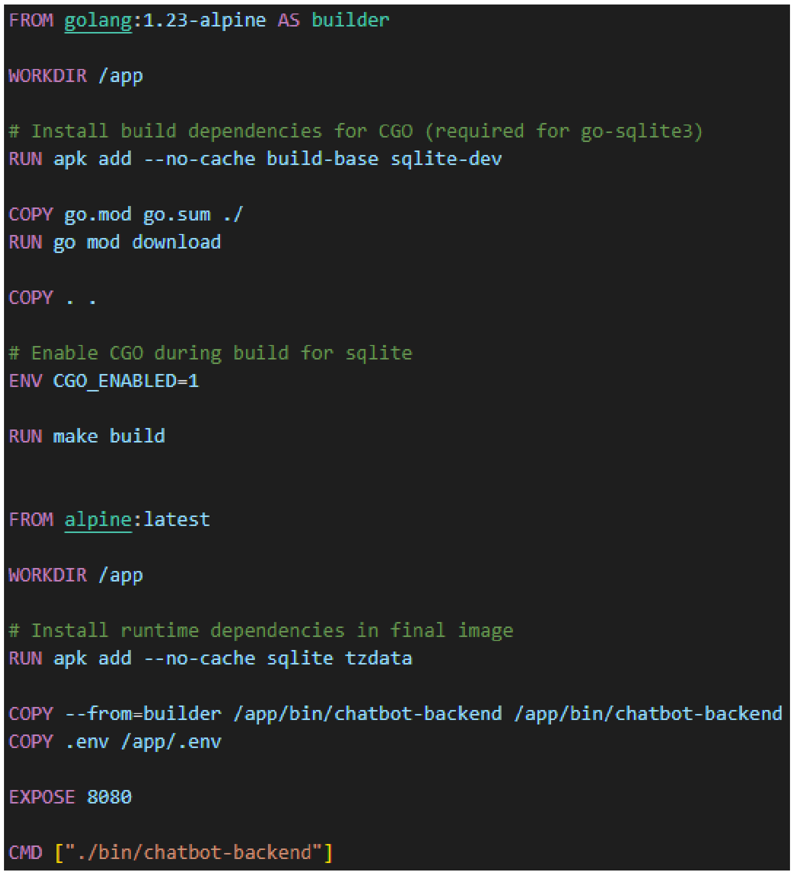


```
# Stage 1: Build the React app
FROM node:20-alpine AS builder

WORKDIR /app

COPY package*.json ./
RUN npm install

COPY . .
RUN npm run build

# Stage 2: Serve the React app with Nginx
FROM nginx:alpine

COPY ./nginx/default.conf /etc/nginx/conf.d/default.conf
COPY --from=builder /app/dist /usr/share/nginx/html

EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Figure 51: Dockerfile for frontend application

- Dockerfile (Backend)

A Dockerfile was written to build the Docker image for the backend application. The Dockerfile utilizes multi-stage builds [59] to minimize the final image size similar to the frontend.

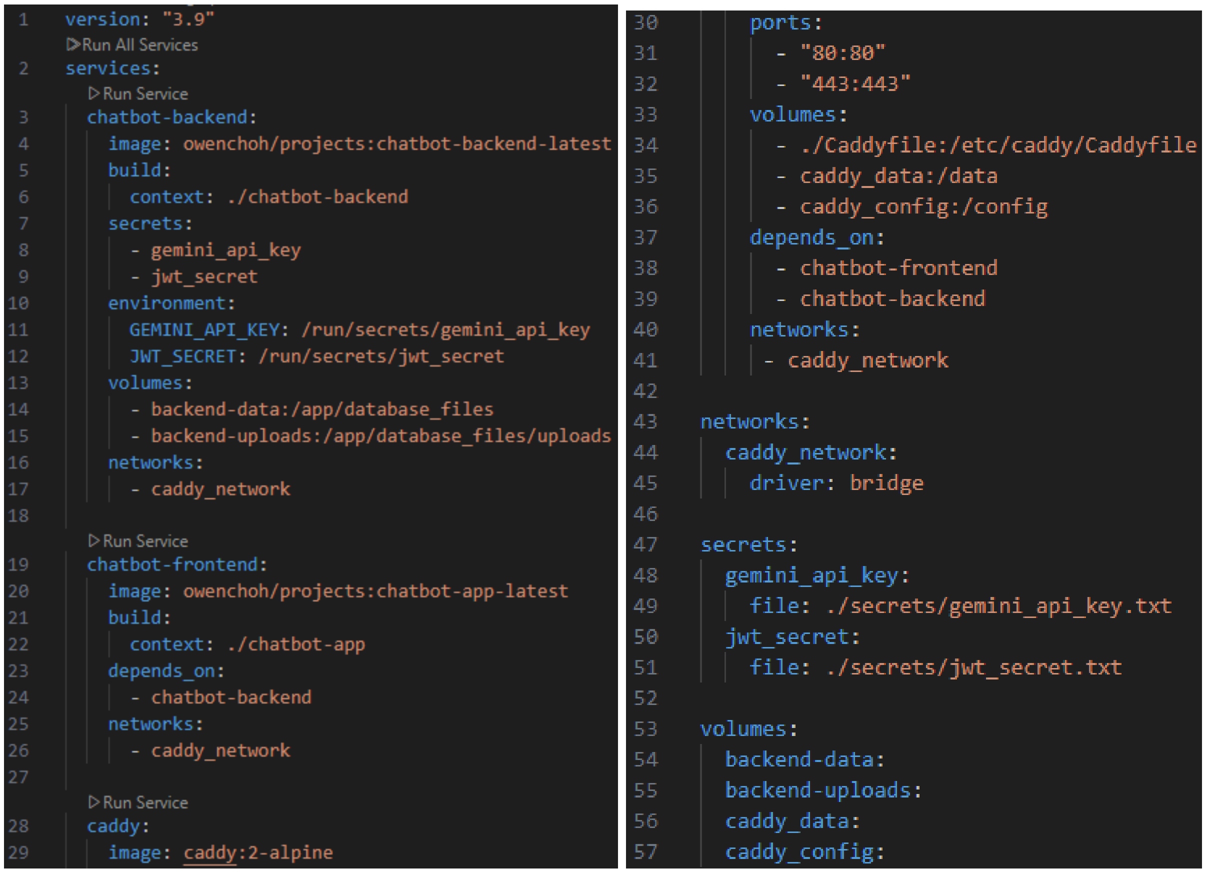


```dockerfile
FROM golang:1.23-alpine AS builder

WORKDIR /app

# Install build dependencies for CGO (required for go-sqlite3)
RUN apk add --no-cache build-base sqlite-dev

COPY go.mod go.sum ./
RUN go mod download

COPY . .

# Enable CGO during build for sqlite
ENV CGO_ENABLED=1

RUN make build


FROM alpine:latest

WORKDIR /app

# Install runtime dependencies in final image
RUN apk add --no-cache sqlite tzdata

COPY --from=builder /app/bin/chatbot-backend /app/bin/chatbot-backend
COPY .env /app/.env

EXPOSE 8080

CMD ["./bin/chatbot-backend"]
```

Figure 52: Dockerfile for the backend application

- Docker Compose

The docker-compose.yaml file orchestrates the deployment of the application's containers. As depicted in figure 53, the container ports are not directly exposed to the host. Instead, the containers operate within an internal Docker network. This isolates the HTTP communication between the Caddy service and the application services, enhancing security by ensuring that only Caddy handles SSL termination, as explained in Section 3.5.5 ("Deployment with Reverse Proxy (Caddy)") [60]. Docker Secrets [22] are used to avoid embedding sensitive information (e.g., API keys, passwords) directly within the container images. Volumes are used to persist data (e.g., database files, user uploads), ensuring that data is preserved across container restarts or replacements during updates [61].



```yaml
version: "3.9"
services:
  chatbot-backend:
    image: owenchoh/projects:chatbot-backend-latest
    build:
      context: ./chatbot-backend
    secrets:
      - gemini_api_key
      - jwt_secret
    environment:
      GEMINI_API_KEY: /run/secrets/gemini_api_key
      JWT_SECRET: /run/secrets/jwt_secret
    volumes:
      - backend-data:/app/database_files
      - backend-uploads:/app/database_files/uploads
    networks:
      - caddy_network

  chatbot-frontend:
    image: owenchoh/projects:chatbot-app-latest
    build:
      context: ./chatbot-app
    depends_on:
      - chatbot-backend
    networks:
      - caddy_network

  caddy:
    image: caddy:2-alpine
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./Caddyfile:/etc/caddy/Caddyfile
      - caddy_data:/data
      - caddy_config:/config
    depends_on:
      - chatbot-frontend
      - chatbot-backend
    networks:
     - caddy_network

networks:
  caddy_network:
    driver: bridge

secrets:
  gemini_api_key:
    file: ./secrets/gemini_api_key.txt
  jwt_secret:
    file: ./secrets/jwt_secret.txt

volumes:
  backend-data:
  backend-uploads:
  caddy_data:
  caddy_config:
```

Figure 53: Configurations in the docker-compose.yaml file after the optimisation explained in the following section

4.3.2 Deployment steps

The following steps were executed to deploy the application to an AWS EC2 instance:

1. EC2 Instance Creation: A new EC2 instance was launched within the AWS free tier [62], utilizing the default settings. The instance was created in the Singapore (ap-southeast-1) region to minimize latency for users accessing the application from Singapore.

2. SSH Access: The SSH key generated during EC2 instance creation was used to establish a secure SSH connection to the instance. For example:

```
ssh -i ./my-ec2-key.pem ec2-user@10.100.10.100
```

3. Software Installation: The following commands were executed to update the system packages and install the necessary software (Docker, Docker Compose [63], and Git):

```
sudo yum update
sudo yum install -y docker
sudo service docker start
sudo usermod -a -G docker ec2-user
docker ps
sudo curl -L
https://github.com/docker/compose/releases/latest/download/docker-compose-$(unam
e -s)-$(uname -m) -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
docker-compose version
sudo yum install git
```

4. GitHub Access: A GitHub Personal Access Token (PAT) was created to allow the EC2 instance to access the private repository containing the application code, following GitHub's documentation [64]. To adhere to the principle of least privilege, the PAT was configured with the minimal permissions required: read access to the code and metadata.

5. Code Retrieval: The application code was retrieved from the GitHub repository using the following commands:

```
mkdir app
cd app
git clone https://github.com/Owen-Choh/SC4052-Cloud-Computing-Project.git
```

6. Configuration and Secrets Transfer: Secure Copy Protocol (SCP) was used to securely transfer the environment files and secrets to the EC2 instance. For example:

```
scp -i ./my--ec2-key.pem ./path/to/the/file
ec2-user@10.100.10.100:/home/ec2-user/app/SC4052-Cloud-Computing-Project
```

7. Application Deployment: The application was built and launched using Docker Compose:

```
cd SC4052-Cloud-Computing-Project
docker-compose up -d -build
docker ps
```

8. Security Group Configuration: The EC2 instance's security group was configured to restrict network access:
   - SSH (Port 22): Allowed from 0.0.0.0/0 to enable SSH access for administrative purposes. However, this configuration presents a security risk, as it allows SSH access from any IP address on the internet. For enhanced security, it is strongly recommended to restrict SSH access to a limited set of trusted IP addresses

(e.g., the developer's static IP address) or a dedicated bastion host in future iterations of the project.
    - HTTPS (Port 443): Allowed from 0.0.0.0/0 to enable public access to the application via HTTPS, a well-known port for secure web traffic [65].

4.3.3 Deployment optimisations

The initial deployment process, while functional, presented several inefficiencies and security concerns. The following optimizations were implemented to address these issues and streamline the deployment workflow.

4.3.1.1 Moving to Docker Hub

The initial deployment strategy involved directly pulling code updates and building Docker images on the EC2 instance. This approach was suboptimal for several reasons:

- Resource Constraints: The t2.micro instance, being a free-tier offering, has limited resources (a single virtual CPU [47]), making image builds slow and resource-intensive.
- Storage Consumption: Building images on the EC2 instance consumes valuable storage space, requiring periodic pruning of Docker caches.

To address these issues, the project transitioned to utilizing dockerhub (https://hub.docker.com/) for image hosting. This allowed for building images locally on the developer's machine, which has significantly more processing power and benefits from effective build caching.

The following steps were involved in this transition:

1. A private Docker Hub repository was created, following the steps outlined in the Docker documentation [66].

2. A dedicated Personal Access Token (PAT) was generated for the EC2 instance to access the private repository [67]. This PAT was configured with read-only permissions to minimize the potential impact if the PAT were ever compromised.

With this change, the deployment process was streamlined, involving the following steps:

1. After making updates, the developer builds and pushes the images locally:

```
docker-compose build
docker-compose push
```

2. The developer then SSHs into the EC2 instance and pulls the updated images and restarts the containers:

```
docker-compose pull
docker-compose up -d
```

While configuration files still require manual transfer using SCP, leveraging Docker Hub significantly reduced the time required for each deployment.

4.3.1.2 Github actions

While the transition to dockerhub for image hosting yielded significant time savings, the deployment process still involved multiple manual steps, increasing the potential for human error. These errors could include outdated configuration files or missing secrets on the EC2 instance, leading to faulty deployments. Such issues could severely impact user experience or even result in platform outages if deployed to a production environment.

To mitigate these risks and streamline the deployment process, the project integrated a Continuous Integration and Continuous Delivery (CI/CD) solution using GitHub Actions. GitHub Actions enables the automation of building, testing, and deployment tasks by defining workflows that build and test code within the project's GitHub repository, or deploy changes to production [68]. The link to the workflow runs can be found in the appendix under “Github Workflow Runs”.

The frontend UI is currently tested manually. While the backend includes automated tests, they are primarily focused on ensuring the quality of the authentication and authorization code. To minimize workflow complexity and reduce setup overhead, automated testing within GitHub Actions was not implemented in this initial iteration, as manual testing of the core functionalities remained feasible. The GitHub Actions workflow file, deploy.yml, is included in the source code (refer to the appendix) for detailed information on the workflow configuration.

The workfile includes two stages:

- build-and-push-image: This stage builds the Docker images for the backend and frontend services and pushes them to dockerhub.
- deploy: This stage establishes an SSH connection to the EC2 instance and executes the necessary commands to stop the existing containers, pull the updated images, and restart the application using Docker Compose.

The deployment steps mirror the manual procedures established in the earlier "Deployment" sections, simplifying the translation into a GitHub Actions workflow. However, integrating GitHub Actions presented several challenges:

- Command Line Interface (CLI) Dependencies: The workflow requires access to specific CLI tools, such as the repository itself, docker and docker compose. To ensure these tools were available within the GitHub Actions environment, the project leveraged community-maintained actions from the GitHub Marketplace, namely actions/checkout@v3 and docker/login-action@v2. These actions provide pre-configured environments, streamlining the setup process and promoting a more robust pipeline.

- Image Layer Caching: To accelerate the image building process, the project incorporated image layer caching techniques. Referring to the Docker Hub documentation [69], the workflow was enhanced with additional tools such as docker/setup-buildx-action@v3 and docker/build-push-action@v6. These actions enable caching of Docker image layers, which significantly reduces build times by avoiding redundant rebuilding of unchanged layers. With this caching mechanism, building new images would only take on average one minute, instead of upwards of three minutes to deploy with the same changes.

- Manual Triggering: The project opted for a manual trigger (using workflow-dispatch) for the deployment workflow. This decision was made to avoid automatic deployments for every code change, as not all code updates warrant immediate deployment to the EC2 instance such as updates to documentation. This allows for more deliberate and controlled deployments, saving on unnecessary resource consumption and preventing potentially disruptive deployments of incomplete features.

- SSH Host Key Verification: An error was encountered during the SSH connection to the EC2 instance ("Host key verification failed"). After investigating this issue, it seems to be an issue with the known_host file used by the SSH command [70], a solution was implemented using the StrictHostKeyChecking=no flag in the SSH command [71]. This flag bypasses the host key verification process. However, it's important to acknowledge the security implications of this approach. While simplifying deployment, disabling host key verification exposes the connection to potential man-in-the-middle attacks. Due to time constraints and challenges encountered in configuring the GitHub Actions environment to add the EC2 instance's public key to the ssh “known_host” file, bypassing host key verification was deemed the expedient, albeit less secure, solution for this iteration of the project. Future iterations should prioritize implementing proper key management for enhanced security. The figure below shows the bypass:

```
ssh -o StrictHostKeyChecking=no -i private_key ${USER_NAME}@${HOSTNAME}
```

Figure 54: SSH command taken from [71]

- GitHub Secrets and Environment Variables: To securely manage sensitive information such as the SSH private key and dockerhub credentials, the project utilized GitHub Secrets as described in the GitHub documentation [72]. These secrets are securely stored within the GitHub repository settings and are exposed as environment variables within the GitHub Actions workflow. This approach not only prevents sensitive credentials from being directly embedded in the workflow file, enhancing the security of the deployment pipeline and ensures that the Docker images are built consistently and without the risk of incorporating local configurations or secrets that could be accidentally included if secrets were stored locally, but also employs a separate dockerhub PAT specifically created for the GitHub Actions workflow. The PAT permissions were carefully scoped to the minimum required for the workflow to function correctly (read & write) to minimize the potential impact if it was ever compromised. This approach helps to prevent the PAT from being used for malicious purposes beyond its intended scope and is distinct from any PATs used for local development or deployments on the EC2 instance.

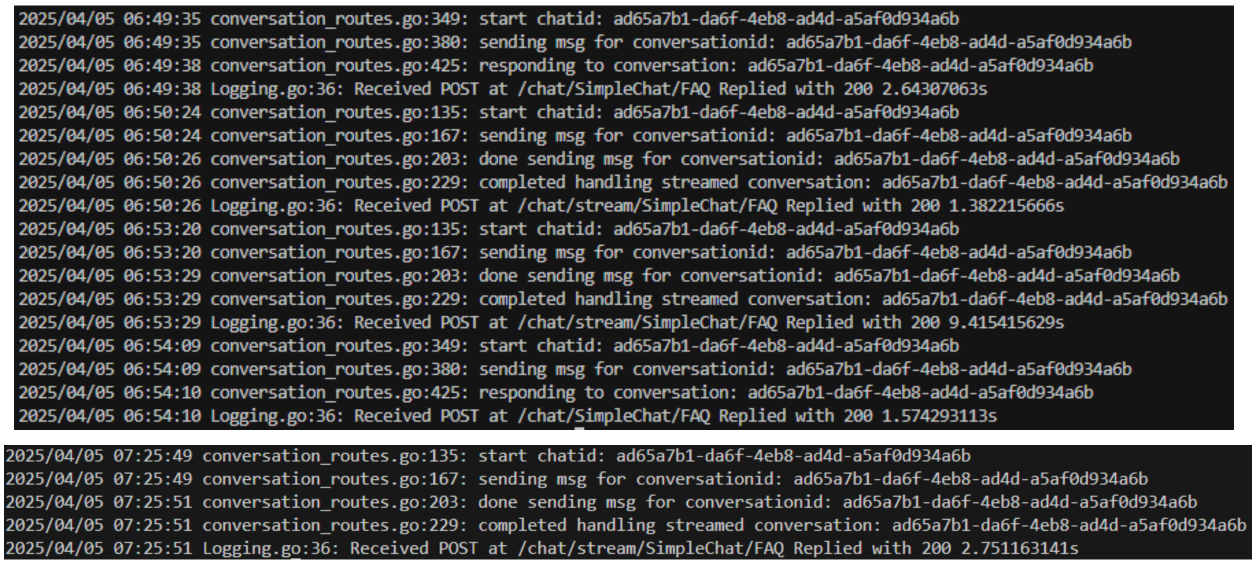
5. Evaluation and Results

This section presents an evaluation of the chatbot platform, analyzing its performance, usability, and security characteristics.

5.1 Performance Metrics

5.1.1 Response Time

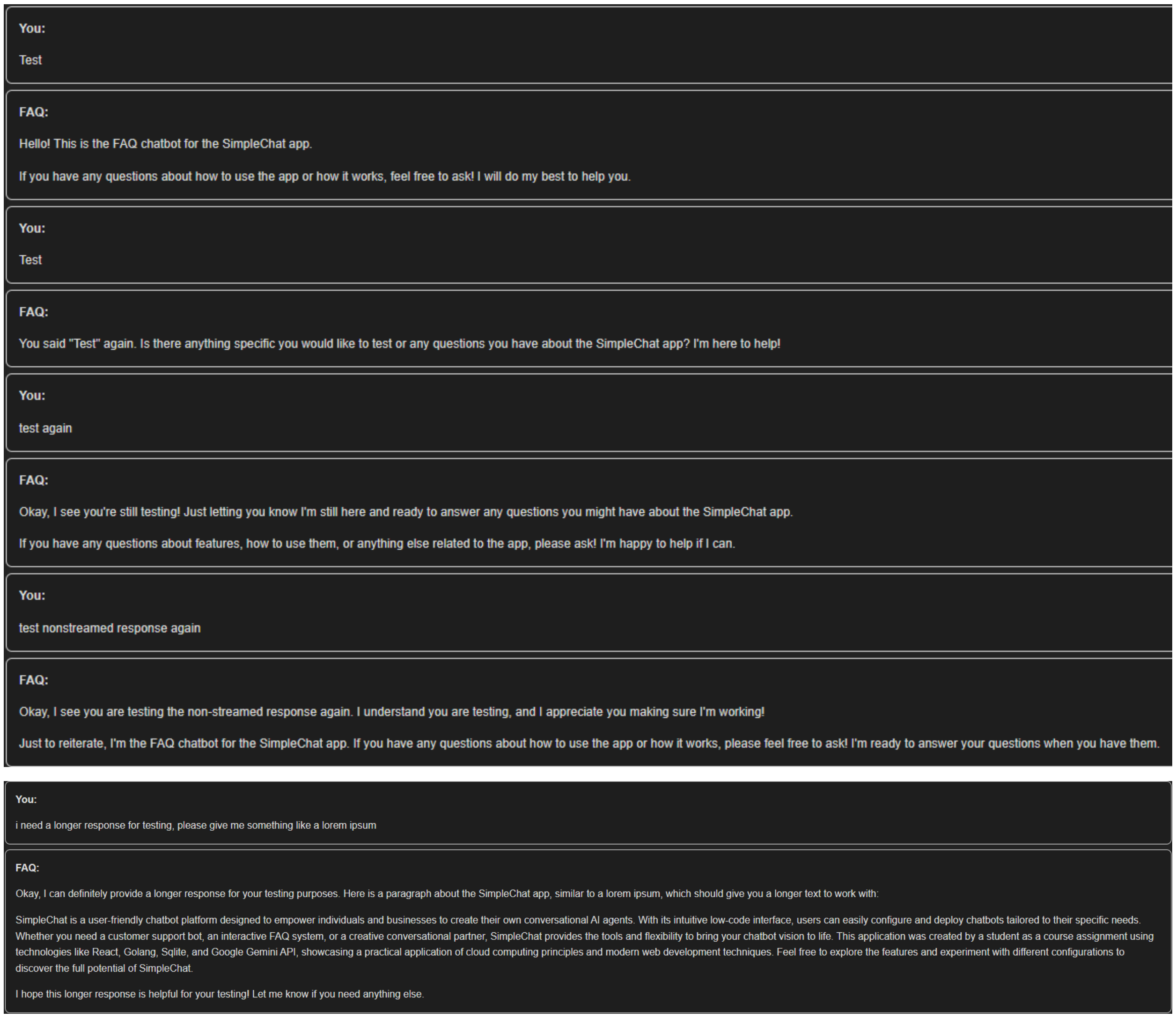
Due to time constraints, a limited performance evaluation was conducted based on manual testing and the existing logging middleware. While this evaluation is not comprehensive, it provides some preliminary insights into the conversation with chatbot's end-to-end response times. Figure 55 displays a sample of timestamps logged by the logging middleware and internally in the route handler code in the backend service during testing. Figure 55 shows the prompts and responses used for testing.



```
2025/04/05 06:49:35 conversation_routes.go:349: start chatid: ad65a7b1-da6f-4eb8-ad4d-a5af0d934a6b
2025/04/05 06:49:35 conversation_routes.go:380: sending msg for conversationid: ad65a7b1-da6f-4eb8-ad4d-a5af0d934a6b
2025/04/05 06:49:38 conversation_routes.go:425: responding to conversation: ad65a7b1-da6f-4eb8-ad4d-a5af0d934a6b
2025/04/05 06:49:38 Logging.go:36: Received POST at /chat/SimpleChat/FAQ Replied with 200 2.64307063s
2025/04/05 06:50:24 conversation_routes.go:135: start chatid: ad65a7b1-da6f-4eb8-ad4d-a5af0d934a6b
2025/04/05 06:50:24 conversation_routes.go:167: sending msg for conversationid: ad65a7b1-da6f-4eb8-ad4d-a5af0d934a6b
2025/04/05 06:50:26 conversation_routes.go:203: done sending msg for conversationid: ad65a7b1-da6f-4eb8-ad4d-a5af0d934a6b
2025/04/05 06:50:26 conversation_routes.go:229: completed handling streamed conversation: ad65a7b1-da6f-4eb8-ad4d-a5af0d934a6b
2025/04/05 06:50:26 Logging.go:36: Received POST at /chat/stream/SimpleChat/FAQ Replied with 200 1.382215666s
2025/04/05 06:53:20 conversation_routes.go:135: start chatid: ad65a7b1-da6f-4eb8-ad4d-a5af0d934a6b
2025/04/05 06:53:20 conversation_routes.go:167: sending msg for conversationid: ad65a7b1-da6f-4eb8-ad4d-a5af0d934a6b
2025/04/05 06:53:29 conversation_routes.go:203: done sending msg for conversationid: ad65a7b1-da6f-4eb8-ad4d-a5af0d934a6b
2025/04/05 06:53:29 conversation_routes.go:229: completed handling streamed conversation: ad65a7b1-da6f-4eb8-ad4d-a5af0d934a6b
2025/04/05 06:53:29 Logging.go:36: Received POST at /chat/stream/SimpleChat/FAQ Replied with 200 9.415415629s
2025/04/05 06:54:09 conversation_routes.go:349: start chatid: ad65a7b1-da6f-4eb8-ad4d-a5af0d934a6b
2025/04/05 06:54:09 conversation_routes.go:380: sending msg for conversationid: ad65a7b1-da6f-4eb8-ad4d-a5af0d934a6b
2025/04/05 06:54:10 conversation_routes.go:425: responding to conversation: ad65a7b1-da6f-4eb8-ad4d-a5af0d934a6b
2025/04/05 06:54:10 Logging.go:36: Received POST at /chat/SimpleChat/FAQ Replied with 200 1.574293113s
```

```
2025/04/05 07:25:49 conversation_routes.go:135: start chatid: ad65a7b1-da6f-4eb8-ad4d-a5af0d934a6b
2025/04/05 07:25:49 conversation_routes.go:167: sending msg for conversationid: ad65a7b1-da6f-4eb8-ad4d-a5af0d934a6b
2025/04/05 07:25:51 conversation_routes.go:203: done sending msg for conversationid: ad65a7b1-da6f-4eb8-ad4d-a5af0d934a6b
2025/04/05 07:25:51 conversation_routes.go:229: completed handling streamed conversation: ad65a7b1-da6f-4eb8-ad4d-a5af0d934a6b
2025/04/05 07:25:51 Logging.go:36: Received POST at /chat/stream/SimpleChat/FAQ Replied with 200 2.751163141s
```

Figure 55: Time stamps logged by backend service



You:

Test

FAQ:

Hello! This is the FAQ chatbot for the SimpleChat app.

If you have any questions about how to use the app or how it works, feel free to ask! I will do my best to help you.

You:

Test

FAQ:

You said "Test" again. Is there anything specific you would like to test or any questions you have about the SimpleChat app? I'm here to help!

You:

test again

FAQ:

Okay, I see you're still testing! Just letting you know I'm still here and ready to answer any questions you might have about the SimpleChat app.

If you have any questions about features, how to use them, or anything else related to the app, please ask! I'm happy to help if I can.

You:

test nonstreamed response again

FAQ:

Okay, I see you are testing the non-streamed response again. I understand you are testing, and I appreciate you making sure I'm working!

Just to reiterate, I'm the FAQ chatbot for the SimpleChat app. If you have any questions about how to use the app or how it works, please feel free to ask! I'm ready to answer your questions when you have them.

You:

i need a longer response for testing, please give me something like a lorem ipsum

FAQ:

Okay, I can definitely provide a longer response for your testing purposes. Here is a paragraph about the SimpleChat app, similar to a lorem ipsum, which should give you a longer text to work with:

SimpleChat is a user-friendly chatbot platform designed to empower individuals and businesses to create their own conversational AI agents. With its intuitive low-code interface, users can easily configure and deploy chatbots tailored to their specific needs. Whether you need a customer support bot, an interactive FAQ system, or a creative conversational partner, SimpleChat provides the tools and flexibility to bring your chatbot vision to life. This application was created by a student as a course assignment using technologies like React, Golang, Sqlite, and Google Gemini API, showcasing a practical application of cloud computing principles and modern web development techniques. Feel free to explore the features and experiment with different configurations to discover the full potential of SimpleChat.

I hope this longer response is helpful for your testing! Let me know if you need anything else.

Figure 56: Prompts and responses used for simple testing of API responses

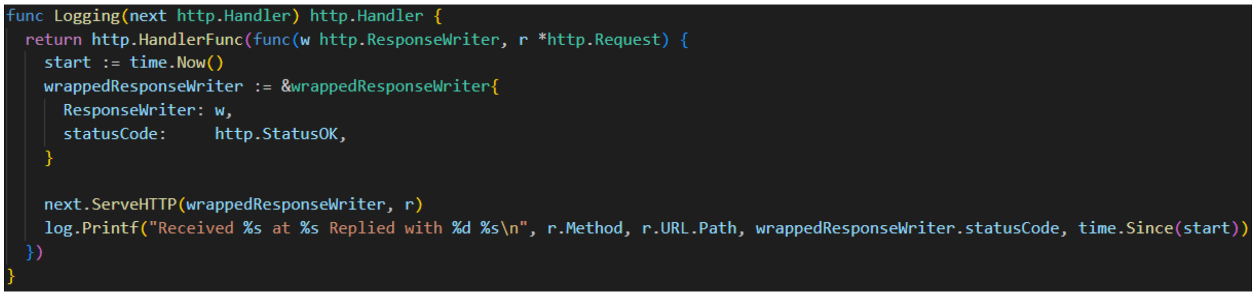
Based on a small sample of manual tests, the average end-to-end response time (measured from the initial HTTP request to the completion of the backend processing, including waiting for the external API) was approximately 3.75 seconds. It is important to note that this measurement includes the time spent in various parts of the system, not exclusively the Gemini API response time. There also appeared to be intermittent spikes in response times (as shown in figure 55), potentially due to network latency or fluctuations in the API's performance.

Given the limitations of the evaluation, it is not possible to draw definitive conclusions about the relative performance of streamed versus non-streamed responses. However, manual testing of the application suggests that the spikes in API response time are infrequent. Additionally, even though the total processing time may be similar, streamed responses may provide a more interactive user experience since responses tend to start loading sooner, which gives the illusion that the chatbot is responding faster.

As shown in figure 57, the current logging middleware measures the end-to-end processing time for each request but does not isolate the Gemini API's response time. More precise data collection is required to separate the processes. To improve the accuracy and usefulness of future performance evaluations, the following additions are recommended:

- Precise Metrics: The logging middleware, as shown in figure 57, measures the end-to-end processing time for each request. While this provides valuable information about the overall system performance, it doesn't isolate the Gemini API's response time. Further work is needed such as changing the code to isolate the points such as AI Stream vs Non-Stream time for more information.

- Increase the number of measurements. Given more time, it would be appropriate to gather more data for each case to be able to draw a reliable conclusion from the measurements.



```go
func Logging(next http.Handler) http.Handler {
  return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    start := time.Now()
    wrappedResponseWriter := &wrappedResponseWriter{
      ResponseWriter: w,
      statusCode:     http.StatusOK,
    }

    next.ServeHTTP(wrappedResponseWriter, r)
    log.Printf("Received %s at %s Replied with %d %s\n", r.Method, r.URL.Path, wrappedResponseWriter.statusCode, time.Since(start))
  })
}
```

Figure 57: Logging middleware in golang backend

5.1.2 Resource Utilization

During the limited testing performed, the EC2 instance's CPU utilization peaked at 3.02%, as shown in figure 58. Based on the available data, memory utilization appeared to be relatively stable and largely unchanged during testing; however, precise memory usage data was not collected due to time constraints. The low CPU utilization suggests that the application is running efficiently within the constraints of the t2.micro instance, with resources available for handling increased load, even without optimizing for more.

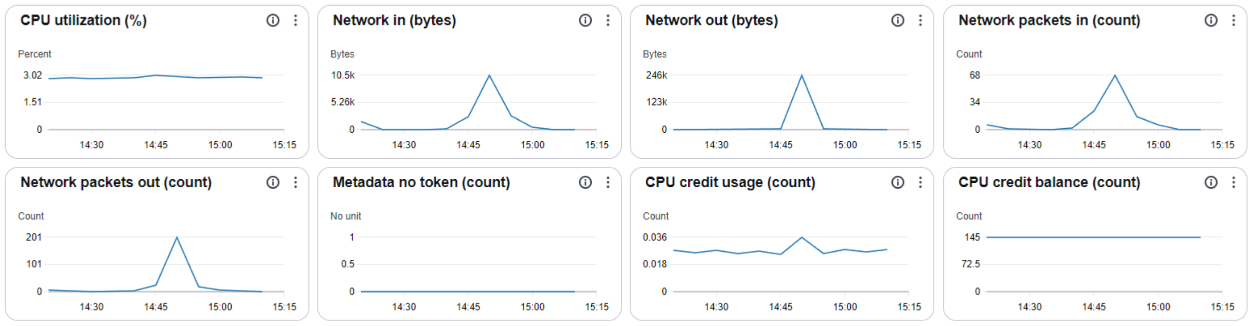


Figure 58: EC2 instance resource utilization during manual testing, showing peak CPU use and network data

5.2 Chatbot Accuracy and Performance

Given the constraints of the project timeline and resources, a comprehensive evaluation of chatbot accuracy and performance was not possible. However, the functionality of the platform has been assessed through iterative development and manual testing of the system prompts, configurations, and responses. These include using the project requirements as the part of the customisations and using the creating an FAQ chatbot that is accessible on the front page of the SimpleChat platform.

This section presents a qualitative overview of the findings based on this development-driven testing process.

- System Instructions Iteration

The system instructions have been progressively refined through iterative testing to optimize the chatbot's role, behavior, and contextual awareness. This involved experimenting with different phrasing, instructions, and knowledge inputs to improve the accuracy, coherence, and helpfulness of the chatbot's responses.

Certain system instructions also seem to cause the Chatbot to misunderstand the requirements. For instance, before reaching the final foundational instruction shown in figure 40, the chatbot is sometimes unable to understand its purpose unless indicated by user input.

Moreover there may also be errors in the response, for example, before adding the line below (refer to figure 60) to the foundational system instruction, the chatbot will incorrectly return its responses entirely wrapped in ```markdown``` tags or even attempt to generate images when not supported by the API.

```
Format your responses using standard Markdown syntax for text styling like bolding, italics, lists, and headings when relevant for readability and clarity.\n
```

Figure 59: Part of the foundational system instruction added to ensure that the response can be formatted by react-markdown

```
Avoid using Markdown code blocks unless you are specifically instructed to display code.\nNote that your API is currently only able to return text response and is unable to return images in the response.\n
```

Figure 60: Instruction added to address errors in the chatbot response

- Known Issues and Limitations

The chatbot API has certain shortcomings that are found through interacting with the API, mainly the small limitations to the UI such as file loading as described in the earlier section. Investigation is also required to determine the cause behind the spike in API response.

In conclusion, while this assessment provides initial insights into the chatbot's functionality, a more rigorous evaluation is required to measure its performance and overall user functionality.

6. Future Work and Recommendations

Based on the developer's experience, and with direct experience during the development process as well as insights derived from preparing the SimpleChat User Guide & FAQ (refer to the appendix), the following recommendations are suggested to improve usability, scalability, and long-term maintainability of the SimpleChat platform.

6.1 Enhance Usability and User Experience

To streamline the User Experience and the User and the ChatBot creation, the following improvements are recommended:

- The Chatbot creation process is not easily understood and makes it difficult to follow through, such future iterations should address to further enhance:
    - Adding help tooltips to each setting for better understanding and configuration.
    - Streamlining the process by minimizing the number of required input fields and simplifying the configuration steps.

- Provide clear Instructions and guidance on the UI to help users understand the purpose and expected input for each setting.

The design could be improved by adding the following enhancements that would vastly improve user and brand perception

- Implement Password Management Features: Implementing features such as password reset and account recovery will improve the user experience by increasing convenience as well as increase account security.

- Improve File Upload Management: The current method for removing uploaded files (toggling a button and then saving) is not intuitive.
    - Also consider using a modal that prompts the user with a confirmation since this is a destructive action.

Additional features may be added that could enhance the product.

- Provide a Content Ideation Assistant or something similar. This would help with the process by
    - Provide an AI chatbot to recommend inputs.
    - The system could take information from users to add details.

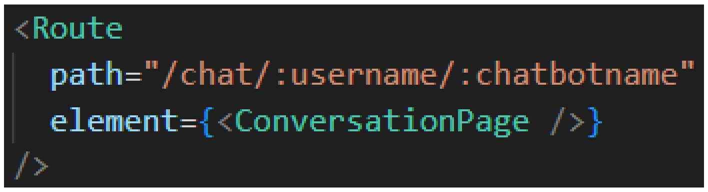
- Provide additional options that users can select. These “advanced options” can allow users with more advanced knowledge to specify additional parameters for their chatbot, such as the model, max output tokens, temperature, topK, topP and stop_sequences [73] while not alienating users if the current default settings are still provided.

- Implement automated test cases to test the rest of code as only a subset is included in the project as described in an earlier section.

In addition to specific features, the assessment also reveals the following technical limitations.

- **Filename and Username Limitations**: The current file upload process imposes restrictions on filenames, forcing users to adhere to a specific format. Sanitizing filenames by translating them to a "safe" name in the database before using that to store the file in the local system would mitigate this limitation. Usernames can also be sanitized the same way or the project can opt to use the user id instead.

- **Chatbot and Username Constraints**: Similar constraints exist for chatbot and usernames, limiting flexibility. Leveraging URL parameters instead of relying on path parameters would alleviate these restrictions and improve usability. However, do note that sanitization will still be required to address the risk of cross site scripting [74].



```
<Route
  path="/chat/:username/:chatbotname"
  element={<ConversationPage />}
/>
```

Figure 61: Current implementation using path parameters

- **Enhanced UI to be more user-centric**: The current design is built for desktop view, additional mobile compatibility or phone applications should be made to improve usability.

Given the subjective nature of this assessment and the lack of external user feedback, these identified areas for improvement should be considered preliminary. Future iterations of the platform should prioritize formal usability testing with a representative sample of users to validate these observations and identify additional areas for enhancement.

6.2 Scalability Improvements

To be able to handle more traffic and also keep the costs down more analysis would need to be done. However, below are the recommendations to improve the scalability of the project:

- The current servers have no scaling, implementing that to the architecture will greatly improve handling of increased user traffic.

- To save on costs, the usage of the components may be considered and should be tested with a load test to determine which components need to be scaled independently of the rest.

- Additional server types or service providers may also be available that could better suit the needs of this project.

6.3 Security Enhancements

Even though there is a current base implementation of encryption to protect and keep that data private, it is still important that the security would be improved or at least enhanced to secure from most issues. To prevent future security incidents, the recommended actions are to:

- Add MFA Authentication through services such as Microsoft Authenticator.

- Create an account validation and setup with email, this also resolves the need to add all the manual details that must be made.

- Rate limit the login API endpoint to improve the resistance to password brute forcing attempts

- Implement a mechanism to revoke JWT if required

6.4 Technology Upgrades

While the decision to use bcrypt has been determined to be good enough for now, with more compute power from more powerful computers or better servers, considering implementing the following actions.

- Argon2id to help reduce the risk of GPU attacks as mentioned in an earlier section.

- Introduce models from Ollama to have more variety in the LLM models available to users as these are very competitive to well known models such as Gemini as shown on the benchmark [75].

7. Conclusion

This project explored the development of a Chatbot-as-a-Service (CaaS) platform built around Low-Code/No-Code (LCNC) principles. The aim was to empower users, particularly those without technical backgrounds, to create and customize AI-powered chatbots through an accessible and intuitive interface. By removing the need to write code, the platform lowers the barrier to entry for conversational AI and encourages wider adoption across various user groups.

The system combines a React frontend with a Golang backend and uses SQLite for data storage. It integrates the Gemini API to provide intelligent responses from a state-of-the-art language model. Key features include user authentication, chatbot customization, support for document uploads, and real-time conversations via server-sent events.

SimpleChat demonstrates the potential of LCNC approaches to simplify complex AI systems and enable users to create functional chatbot solutions with ease. Future work should focus on addressing the current limitations in usability, scalability, and security. These include support for multi-user environments, improved analytics, enhanced customization capabilities, and better system APIs. While some goals, such as image generation, were not achieved, the current version lays a strong foundation for continued development. With further iterations, SimpleChat can evolve into a more robust and versatile platform, opening up even more opportunities for end users.

8. References

[1] 'What chatbots do well - Generative AI and Chatbots - Research Guides at Temple University'. Accessed: Apr. 05, 2025. [Online]. Available: https://guides.temple.edu/ai-chatbots/well

[2] 'What is AWS? - Cloud Computing with AWS - Amazon Web Services'. Accessed: Apr. 05, 2025. [Online]. Available: https://aws.amazon.com/what-is-aws/

[3] 'Low-Code/No-Code: The Future of Development', SAP. Accessed: Apr. 05, 2025. [Online]. Available: https://www.sap.com/sea/products/technology-platform/build/what-is-low-code-no-code.html

[4] J. X. Silva, M. Lopes, G. Avelino, and P. Santos, 'Low-code and No-code Technologies Adoption: A Gray Literature Review', in *Proceedings of the XIX Brazilian Symposium on Information Systems*, in SBSI '23. New York, NY, USA: Association for Computing Machinery, Jun. 2023, pp. 388–395. doi: 10.1145/3592813.3592929.

[5] 'Ollama'. Accessed: Apr. 05, 2025. [Online]. Available: https://ollama.com

[6] C. W. Server, 'Caddy - The Ultimate Server with Automatic HTTPS', Caddy Web Server. Accessed: Apr. 05, 2025. [Online]. Available: https://caddyserver.com/

[7] 'LLMs - Yale Center for Research Computing'. Accessed: Apr. 05, 2025. [Online]. Available: https://docs.ycrc.yale.edu/clusters-at-yale/guides/LLMs/

[8] 'An introduction to web applications architecture', Open Learning. Accessed: Apr. 05, 2025. [Online]. Available: https://www.open.edu/openlearn/science-maths-technology/an-introduction-web-applications-architecture/content-section-1.1

[9] 'Gemini Developer API | Gemma open models', Google AI for Developers. Accessed: Apr. 05, 2025. [Online]. Available: https://ai.google.dev/

[10] Atlassian, 'Microservices vs. monolithic architecture', Atlassian. Accessed: Apr. 05, 2025. [Online]. Available: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith

[11] P. A. Grassi, M. E. Garcia, and J. L. Fenton, 'Digital identity guidelines: revision 3', National Institute of Standards and Technology, Gaithersburg, MD, NIST SP 800-63-3, Jun. 2017. doi: 10.6028/NIST.SP.800-63-3.

[12] 'About Microsoft Authenticator - Microsoft Support'. Accessed: Apr. 05, 2025. [Online]. Available: https://support.microsoft.com/en-us/account-billing/about-microsoft-authenticator-9783c865-0308-42fb-a519-8cf666fe0acc

[13] 'What Is Federated Identity? - Okta SG'. Accessed: Apr. 05, 2025. [Online]. Available: https://www.okta.com

[14] 'What is an Authenticator App? How it Works, Advantages & More | Lenovo Singapore'. Accessed: Apr. 05, 2025. [Online]. Available: https://www.lenovo.com/sg/en/glossary/authenticator-app/

[15] D. M'Raihi, J. Rydell, M. Pei, and S. Machani, 'TOTP: Time-Based One-Time Password Algorithm', Internet Engineering Task Force, Request for Comments RFC 6238, May 2011. doi: 10.17487/RFC6238.

[16] 'How Two-Factor Authentication is Changing the Password Security Game | Information Technology Division'. Accessed: Apr. 05, 2025. [Online]. Available: https://itd.sog.unc.edu/knowledge-base/article/how-two-factor-authentication-changing-password-security-game

[17] Bejamas, 'Fed Auth 101: What Is Federated Authentication?', Descope. Accessed: Apr. 05, 2025. [Online]. Available: https://www.descope.com/learn/post/federated-authentication

[18] K. Team, ‘Keycloak’. Accessed: Apr. 05, 2025. [Online]. Available: https://www.keycloak.org/
[19] ‘What is HTTPS?’ Accessed: Apr. 05, 2025. [Online]. Available: https://www.cloudflare.com/learning/ssl/what-is-https/
[20] ‘What is a Certificate Authority (CA)?’, SSL.com. Accessed: Apr. 05, 2025. [Online]. Available: https://www.ssl.com/article/what-is-a-certificate-authority-ca/
[21] E. Havens, ‘GitHub found 39M secret leaks in 2024. Here’s what we’re doing to help’, The GitHub Blog. Accessed: Apr. 05, 2025. [Online]. Available: https://github.blog/security/application-security/next-evolution-github-advanced-security/
[22] ‘Manage sensitive data with Docker secrets’, Docker Documentation. Accessed: Apr. 05, 2025. [Online]. Available: https://docs.docker.com/engine/swarm/secrets/
[23] ‘Git - gitignore Documentation’. Accessed: Apr. 05, 2025. [Online]. Available: https://git-scm.com/docs/gitignore
[24] ‘What is Docker?’, Docker Documentation. Accessed: Apr. 05, 2025. [Online]. Available: https://docs.docker.com/get-started/docker-overview/
[25] ‘React’. Accessed: Apr. 05, 2025. [Online]. Available: https://react.dev/
[26] ‘The Go Programming Language’. Accessed: Apr. 05, 2025. [Online]. Available: https://go.dev/
[27] ‘SQLite Home Page’. Accessed: Apr. 05, 2025. [Online]. Available: https://www.sqlite.org/
[28] ‘SQLite: Begin Concurrent’. Accessed: Apr. 05, 2025. [Online]. Available: https://www.sqlite.org/cgi/src/doc/begin-concurrent/doc/begin_concurrent.md
[29] ‘35% Faster Than The Filesystem’. Accessed: Apr. 05, 2025. [Online]. Available: https://www.sqlite.org/fasterthanfs.html
[30] ‘LiveBench’. Accessed: Apr. 05, 2025. [Online]. Available: https://livebench.ai/#/
[31] ‘Gemini Developer API Pricing | Gemini API’, Google AI for Developers. Accessed: Apr. 05, 2025. [Online]. Available: https://ai.google.dev/gemini-api/docs/pricing
[32] ‘Gemini models | Gemini API’, Google AI for Developers. Accessed: Apr. 05, 2025. [Online]. Available: https://ai.google.dev/gemini-api/docs/models
[33] ‘What is SSL Termination? Definition &amp; Related FAQs | VMware’. Accessed: Apr. 05, 2025. [Online]. Available: https://www.vmware.com/topics/ssl-termination
[34] ‘What is a reverse proxy? | Proxy servers explained’. Accessed: Apr. 05, 2025. [Online]. Available: https://www.cloudflare.com/learning/cdn/glossary/reverse-proxy/
[35] ‘Using server-sent events - Web APIs | MDN’. Accessed: Apr. 05, 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events
[36] ‘The WebSocket API (WebSockets) - Web APIs | MDN’. Accessed: Apr. 05, 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
[37] ‘TextDecoder - Web APIs | MDN’. Accessed: Apr. 05, 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/TextDecoder
[38] ‘useState – React’. Accessed: Apr. 05, 2025. [Online]. Available: https://react.dev/reference/react/useState
[39] ‘react-markdown’, npm. Accessed: Apr. 05, 2025. [Online]. Available: https://www.npmjs.com/package/react-markdown
[40] ‘Write-Ahead Logging’. Accessed: Apr. 06, 2025. [Online]. Available: https://www.sqlite.org/wal.html
[41] P. G. D. Group, ‘PostgreSQL’, PostgreSQL. Accessed: Apr. 06, 2025. [Online]. Available: https://www.postgresql.org/
[42] ‘SQL Injection | OWASP Foundation’. Accessed: Apr. 09, 2025. [Online]. Available: https://owasp.org/www-community/attacks/SQL_Injection
[43] ‘Avoiding SQL injection risk - The Go Programming Language’. Accessed: Apr. 09, 2025. [Online]. Available: https://go.dev/doc/database/sql-injection
[44] ‘bcrypt package - golang.org/x/crypto/bcrypt - Go Packages’. Accessed: Apr. 05, 2025.

[Online]. Available: https://pkg.go.dev/golang.org/x/crypto/bcrypt
[45] P. A. Grassi *et al.*, ‘Digital identity guidelines: authentication and lifecycle management’, National Institute of Standards and Technology, Gaithersburg, MD, NIST SP 800-63b, Jun. 2017. doi: 10.6028/NIST.SP.800-63b.
[46] ‘Password Storage - OWASP Cheat Sheet Series’. Accessed: Apr. 05, 2025. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html
[47] ‘Cloud Compute Instances – Amazon EC2 Instance Types – AWS’, Amazon Web Services, Inc. Accessed: Apr. 05, 2025. [Online]. Available: https://aws.amazon.com/ec2/instance-types/
[48] ‘argon2 package - golang.org/x/crypto/argon2 - Go Packages’. Accessed: Apr. 05, 2025. [Online]. Available: https://pkg.go.dev/golang.org/x/crypto/argon2
[49] Auth0, ‘JSON Web Tokens’, Auth0 Docs. Accessed: Apr. 05, 2025. [Online]. Available: https://auth0.com/docs/
[50] R. Rao, ‘JSON Web Tokens (JWT) are Dangerous for User Sessions—Here’s a Solution’, Redis. Accessed: Apr. 05, 2025. [Online]. Available: https://redis.io/blog/json-web-tokens-jwt-are-dangerous-for-user-sessions/
[51] ‘Cookie theft definition – Glossary | NordVPN’. Accessed: Apr. 05, 2025. [Online]. Available: https://nordvpn.com/cybersecurity/glossary/cookie-theft/
[52] ‘Using HTTP cookies - HTTP | MDN’. Accessed: Apr. 05, 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Cookies
[53] ‘jwt package - github.com/golang-jwt/jwt/v5 - Go Packages’. Accessed: Apr. 05, 2025. [Online]. Available: https://pkg.go.dev/github.com/golang-jwt/jwt/v5@v5.2.1
[54] ‘JSON Web Token (JWT) Signing Algorithms Overview’, Auth0 - Blog. Accessed: Apr. 05, 2025. [Online]. Available: https://auth0.com/blog/json-web-token-signing-algorithms-overview/
[55] samirdal, ‘User experience guidelines for errors - Business Central’. Accessed: Apr. 05, 2025. [Online]. Available: https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/devenv-error-handling-guidelines
[56] ‘http package - net/http - Go Packages’. Accessed: Apr. 05, 2025. [Online]. Available: https://pkg.go.dev/net/http#Flusher
[57] ‘Explore document processing capabilities with the Gemini API’, Google AI for Developers. Accessed: Apr. 05, 2025. [Online]. Available: https://ai.google.dev/gemini-api/docs/document-processing
[58] ‘Policy forbids issuing for name on Amazon EC2 domain - Server’, Let’s Encrypt Community Support. Accessed: Apr. 05, 2025. [Online]. Available: https://community.letsencrypt.org/t/policy-forbids-issuing-for-name-on-amazon-ec2-domain/12692/4
[59] ‘Multi-stage’, Docker Documentation. Accessed: Apr. 05, 2025. [Online]. Available: https://docs.docker.com/build/building/multi-stage/
[60] ‘Docker Networking: Exploring Bridge, Host, and Overlay Modes’. Accessed: Apr. 05, 2025. [Online]. Available: https://www.cloudthat.com/resources/blog/docker-networking-exploring-bridge-host-and-overlay-modes
[61] ‘Volumes’, Docker Documentation. Accessed: Apr. 05, 2025. [Online]. Available: https://docs.docker.com/engine/storage/volumes/
[62] ‘Free Cloud Computing Services - AWS Free Tier’. Accessed: Apr. 05, 2025. [Online]. Available: https://aws.amazon.com/free/?all-free-tier.sort-by=item.additionalFields.SortRank&all-free-tier.sort-order=asc&awsf.Free%20Tier%20Types=*all&awsf.Free%20Tier%20Categories=*all

[63] npearce, ‘Amazon Linux 2 - install docker & docker-compose using “sudo amazon-linux-extras” command’, Gist. Accessed: Apr. 05, 2025. [Online]. Available: https://gist.github.com/npearce/6f3c7826c7499587f00957fee62f8ee9
[64] ‘Managing your personal access tokens’, GitHub Docs. Accessed: Apr. 05, 2025. [Online]. Available: https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/managing-your-personal-access-tokens
[65] ‘Service Name and Transport Protocol Port Number Registry’. Accessed: Apr. 05, 2025. [Online]. Available: https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml?search=https
[66] ‘Create’, Docker Documentation. Accessed: Apr. 05, 2025. [Online]. Available: https://docs.docker.com/docker-hub/repos/create/
[67] ‘Access tokens’, Docker Documentation. Accessed: Apr. 05, 2025. [Online]. Available: https://docs.docker.com/security/for-developers/access-tokens/
[68] ‘Understanding GitHub Actions’, GitHub Docs. Accessed: Apr. 05, 2025. [Online]. Available: https://docs.github.com/en/actions/about-github-actions/understanding-github-actions
[69] ‘Cache management’, Docker Documentation. Accessed: Apr. 05, 2025. [Online]. Available: https://docs.docker.com/build/ci/github-actions/cache/
[70] Castiel, ‘Answer to “What is actually in known_hosts?”’, Stack Overflow. Accessed: Apr. 05, 2025. [Online]. Available: https://stackoverflow.com/a/33243564
[71] ‘GitHub Action with EC2 using SSH’, DEV Community. Accessed: Apr. 05, 2025. [Online]. Available: https://dev.to/raviagheda/github-action-with-ec2-using-ssh-4ej4
[72] ‘Using secrets in GitHub Actions’, GitHub Docs. Accessed: Apr. 05, 2025. [Online]. Available: https://docs.github.com/en/actions/security-for-github-actions/security-guides/using-secrets-in-github-actions
[73] ‘About generative models | Gemini API’, Google AI for Developers. Accessed: Apr. 11, 2025. [Online]. Available: https://ai.google.dev/gemini-api/docs/models/generative-models
[74] ‘Cross Site Scripting Prevention - OWASP Cheat Sheet Series’. Accessed: Apr. 05, 2025. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html
[75] ‘LLM Leaderboard - Compare GPT-4o, Llama 3, Mistral, Gemini & other models | Artificial Analysis’. Accessed: Apr. 05, 2025. [Online]. Available: https://artificialanalysis.ai/leaderboards/models

Appendix

Link to source code

https://github.com/Owen-Choh/SC4052-Cloud-Computing-Project

Link to steps to setup Github Actions

https://github.com/Owen-Choh/SC4052-Cloud-Computing-Project/blob/main/setup%20github%20actions.md

Link to Github Workflow Runs

https://github.com/Owen-Choh/SC4052-Cloud-Computing-Project/actions/workflows/deploy.yml

Link to application

https://ec2-54-179-162-106.ap-southeast-1.compute.amazonaws.com/login

"SimpleChat User Guide and FAQ.pdf" documents how to use the application and some FAQs

https://github.com/Owen-Choh/SC4052-Cloud-Computing-Project/blob/main/documents/SimpleChat%20User%20Guide%20and%20FAQ.pdf