ChatGUI

曾为帅 220012931 / 宣典村 2100017415 / 冯子桐 2400017788

A light-weighted desktop software with unified and user-friendly interface to seamlessly interact with multiple large language models (LLMs)

Background

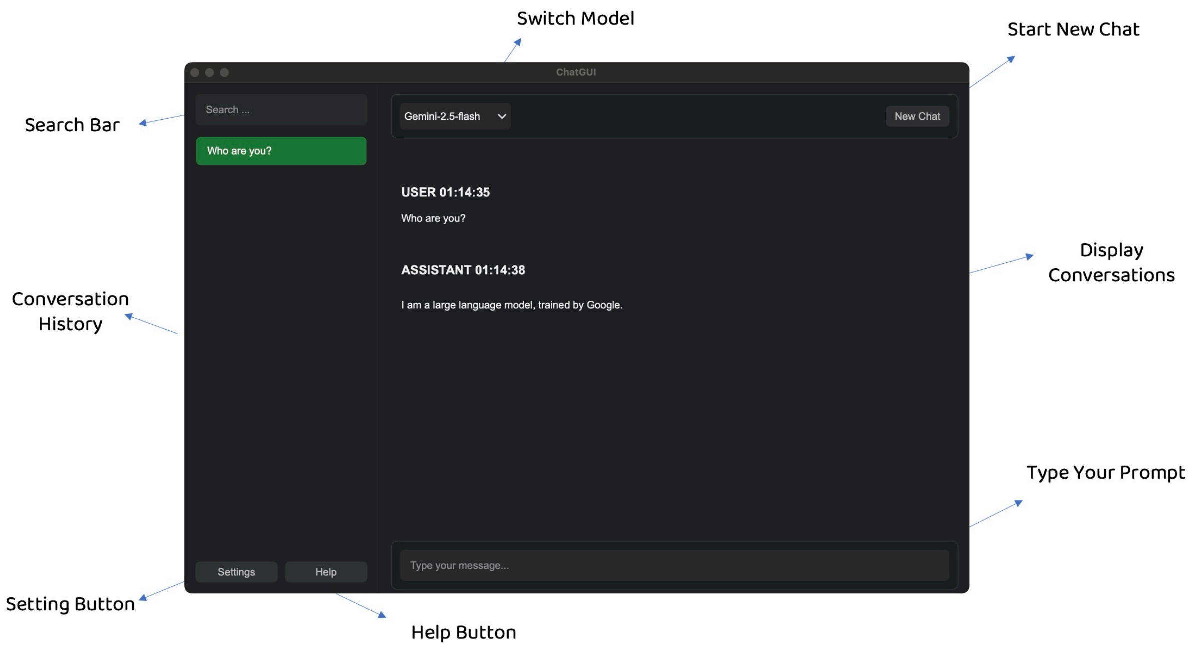
Large language models (LLMs) have gained significant attention for their ability to perform diverse tasks through natural language prompts. These models, trained on extensive datasets, provide an accessible interface for users to obtain results by simply describing their requirements. However, with various companies releasing different LLMs, selecting and utilizing the most suitable model has become a challenge. Performance varies across models—some excel in certain tasks while underperforming in others.

To address this, we propose **ChatGUI**, a lightweight desktop application with a unified, user-friendly interface for seamless interaction with multiple LLMs. Focusing on powerful models like GPT and Gemini, accessible via API keys, ChatGUI enables users to compare outputs across different models efficiently and choose the optimal one for their needs.

Functions

We would introduce our functions based on the *windows* we have. For each window, we would list all the implemented functions

Main Window



USER 01:14:35

Who are you?

ASSISTANT 01:14:38

I am a large language model, trained by Google.

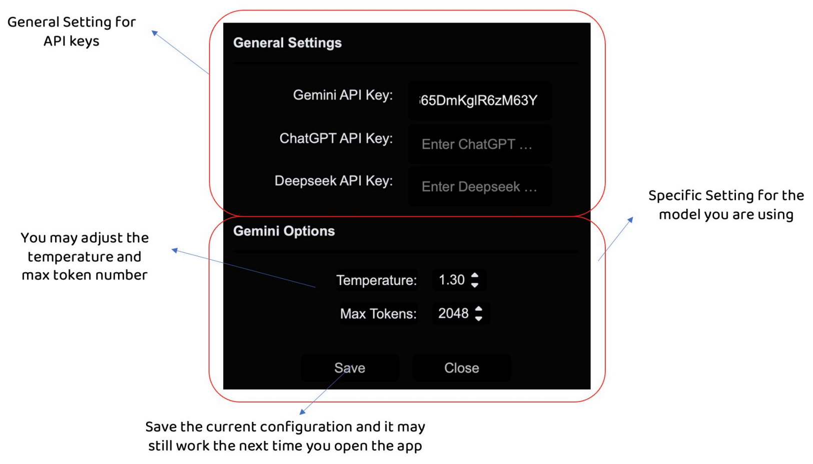
Our main window is a simplified and user-friendly interface which follows the prevailing design of the ChatGPT desktop app. It is mainly made up of the two parts: the history part on the left and the conversation display part on the right. The conversation area possesses the most area as it is the main interaction interface. There are **eight functions** we have implemented and gathered for this window:

1. **Search Bar:** Search Bar is a searching area where you may type some content and the app would automatically search all your chat histories (usually the first dozens of words for efficiency) and give the one that matches your requirement.
2. **Conversation History**: Conversation History is an area where your previous chat histories would get stored. You may click on diverse items to switch and check your chat histories.
3. **Setting Button**: Setting Button is a single button which may raise the setting window once you click on it. You may double click it to close it or directly press the 'x' button on the setting window.

4. **Help Button**: Help Button is a single button which may raise the help window once you click on it. You may also double click it to close it or directly press the 'close' button on the help window.
5. **Switch Model**: Switch Model points to the area where you may switch to diverse foundation models. Current application supports:
   - Deepseek-V3
   - Deepseek-R1
   - Gemini-2.0-flash
   - Gemini-2.5-flash
   - gpt-4o-mini
   - etc.

   To be specific, we do not include all the models in the ChatGPT series as we do not have enough budget to debug. Hence, we just use ChatGPT to represent the whole series and use gpt-4o-mini as a representative, though it is easy to adapt to other models in the ChatGPT series.
6. **Start New Chat**: Start New Chat is single button which may clear the conversation area and start a new conversation page once you click on it.
7. **Display Conversations**: We display conversation in a multi-round style. Each round is made up of the title and conversation details. The title is in bold as a way of emphasis. It contains the entity (USER or ASSISTANT) and time for sending the message. The content is properly arranged and the response is rendered by a markdown module to ensure format.
8. **Type Your Prompt**: You may type anything you would like to chat with the agent. If you have not properly set the API key, the setting window would be raised. Once you have pressed the Enter button, the message would be sent by API.

Setting Window


General Setting for API keys
You may adjust the temperature and max token number
Specific Setting for the model you are using
Save the current configuration and it may still work the next time you open the app
Save
Close

General Settings

Gemini API Key: ·65DmKglR6zM63Y
ChatGPT API Key: Enter ChatGPT ...
Deepseek API Key: Enter Deepseek ...

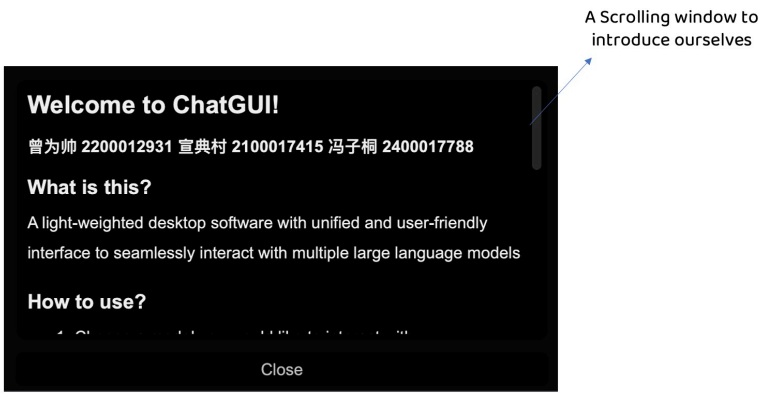
Gemini Options

Temperature: 1.30
Max Tokens: 2048

Our setting window is made up of two parts: the general setting part above and the model-specific part below. There are several functions we have implemented for this window:

1. **General Settings**: General settings would appear no matter which model you are using. It gathers the API key you set for different models, for example, Deepseek or Gemini.
2. **Model-specific Settings**: Model-specific settings would appear depending on which model you are using and it would display the corresponding options. For example, when you are using Gemini series, it will show 'Gemini Options'. The model-specific settings contain temperature and max token number which almost all APIs would share.
3. **Save Button**: Once you click on the save button, current configuration would be saved. This means that you may not need to set the API key once again the next time you open the app.
4. **Close Button**: You may close this window in two ways. The first way is to press the close button. However, the configuration would not be saved if you have not pressed the Save button. The second way is to click on the setting button once again.

Help Window



Welcome to ChatGUI!

曾为帅 2200012931 宣典村 2100017415 冯子桐 2400017788

What is this?

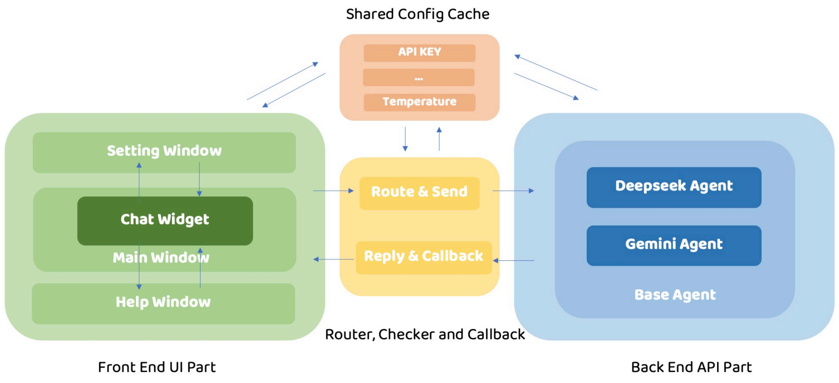
A light-weighted desktop software with unified and user-friendly interface to seamlessly interact with multiple large language models

How to use?

Our help window is a scrolling module where you may know the basic information about our project, our team and some words we'd love to say to users. We consider this window to be an important part of our project as it may help users who uses our app for the first time to know how to use, why they should use and who create such app.

Design and Implementation

In this section, we would introduce the design of the whole pipeline and the implementation of different modules in detail. Our design could be generally summarized into **four parts**, the front-end UI part, the back-end API part, the router part and the config cache part. Each part is packaged well and expose suitable interface for other parts to call, leading to a coherent and highly-modularized working pipeline. In each part, we have maintained a well-designed inheritance strategy, extracting the common features of diverse classes to the most. The overall pipeline can refer to the figure below:



Front-End UI

We implement the front-end UI using Pyside6. As we are designing a simplified and user-friendly interface, we decide to only implement three windows for our app: the main window, the setting window and the help window. The main window is the interface one would interact most so we mainly work on building it. The setting window and help window would be raised if users press button in the main window.

For the main window, the Chat Widget is the core for the front end. It handles the UI display and the connection setup. For UI display, it would setup the interface style and display the messages of users and assistants in a multi-round manner. For connection setups, it would handle the click, type, and scroll operation and call the corresponding functions to process.

Router

As we have multiple models to interact with, we do not allow the front-end part to directly communicate with diverse models, but instead, use a router to send messages and receive response. The router would maintain an activated agent depending on the model you choose. Each time you switch a model, it would delete the previous agent for memory efficiency and initialize the new agent instantly. The router would also do a sanity check before sending the message to the corresponding model. To be specific, it would first check if the API key is properly set for the corresponding model. If it is not, the router would block the message and raise the setting window by sending a callback signal to the front-end UI.

Also, there may be multiple models belonging to the same firm, for example, Deepseek-V3 and Deepseek-R1 all belong to Deepseek. The only difference between their API calling is the model you specify. Hence, we design agents based on their factories. The router, in this way, is also responsible for distinguishing models from diverse factories and find the proper agent to deliver messages.

Back-End API

As different models all support calliing with the OpenAI API, we design an inheritage strategy for agents from diverse firms. The base agent is responsible for setting up clients and sending messages via the OpenAI API. There is a virtual function in the base agent which is used to set up configs like API KEY etc. All the children classes would **ONLY** have to inherit the base agent and implement such virtual function. The reason behind this implementation is that we observe **base_url** is the only factor that differs models when using the OpenAI API.

Config Cache

As config is needed in each part of the pipeline, we design it to be a shared cache. When you set up configs in the setting window, all the options would be saved and they would be later written into the config file when closing the window. Next time you launch the app, it would automatically read the config file and set the value. Along with the config cache, there exists a history buffer for storing the previous conversations. You may click on different conversations on the side bar for checking your chatting history. It would also be automatically saved when you close the window and loaded the next time you launch the app.

Work Division

All the three members contribute equally to this project.

As the overall design is highly-modularized, we follow a divide-and-conquer strategy where three of us maintains a close communication with each other when implementing diverse functions in case that the control interface exposed by one can not be reused by the other.

Conclusion and Reflection

Due to the time constraint, we fail to present you with more briiliant features. We believe that the most disappointing part in our project is that you have to wait until the response by the agent fully arrives before doing any further operations. The reason for this is that we do not allow streaming response and have not implemented a printer effect when displaying the messages. Besides, we find that the cache implementation may consume too many resources and the overall running efficiency is relatively low. And of course, the UI design have much space to improve. As we three are all unfamiliar with UI, we have tried our best to provide a decent interface though it still seems shabby.

However, we believe that the overall experience of our project is precious. We learn knowledge about how to design and implement UI. Also, we are more familiar with the method to call APIs provided by prevailing LLMs. We are exciting to find that the OpenAI API provides a unified interface for all the models and this saves us a lot of time when implementing the back-end functions.

In the end, we want to highlight that the need for such a light-weighted desktop software with unified and user-friendly interface to seamlessly interact with multiple large language models (LLMs) still exists. We are determined to continue working on this tiny but meaningful project as we firmly believe such desktop software would truly make a difference.