

**MINISTRY OF EDUCATION AND TRAINING**  
**UNIVERSITY OF ECONOMICS AND FINANCE**



**PROJECT REPORT**  
**APPLICATION DEVELOPMENT OF ARTIFICIAL**  
**INTELLIGENCE**

**Major: Information Technology**

**Minor: Artificial Intelligence**

**TOPIC: WEBSITE CHATBOT**

**Supervisor: Nguyễn Sơn Lâm, MSc**

**Student:**

Nguyễn Văn Mạnh	215051286	21D1TH-NT01
Lê Vĩnh Ngọc	215051045	21D1TH-NT01
Nguyễn Cao Bằng	215051639	21D1TH-NT01
Nguyễn Ngọc Thịnh	215101362	21D1TH-NT01
Hồ Đức Nguyên	215051637	21D1TH-NT01

**Ho Chi Minh City, 2024**

**MINISTRY OF EDUCATION AND TRAINING  
UNIVERSITY OF ECONOMICS AND FINANCE**

**PROJECT REPORT  
APPLICATION DEVELOPMEN OF ARTIFICIAL  
INTELLIGENCE**

**Major: Information Technology**

**Minor: Artificial Intelligence**

**TOPIC: WEBSITE CHATBOT**

**Supervisor: Nguyễn Sơn Lâm, MSc**

**Student:**

Nguyễn Văn Mạnh	215051286	21D1TH-NT01
Lê Vĩnh Ngọc	215051045	21D1TH-NT01
Nguyễn Cao Bằng	215051639	21D1TH-NT01
Nguyễn Ngọc Thịnh	215101362	21D1TH-NT01
Hồ Đắc Nguyên	215051637	21D1TH-NT01

**Ho Chi Minh City, 2024**

# TABLE OF CONTENTS

## CONTENTS

TABLE OF CONTENTS .....	i
LIST OF IMAGES .....	iii
ABSTRACTS .....	v
CHAPTER 1 . INTRODUCTION .....	1
1.1 OVERVIEW OF THE TOPIC.....	1
1.2 REASONS FOR CHOOSING THE TOPIC .....	1
1.3 OBJECTIVES OF THE RESEARCH .....	1
1.4 PROJECT STRUCTURE .....	2
CHAPTER 2 . THEORETICAL BASIS .....	4
2.1 HTML.....	4
2.2 Python .....	4
2.3 Javascript.....	5
2.4 visual studio .....	5
2.5 google collab .....	5
2.6 Gemini AI .....	6
2.7 Hugging Face .....	6
CHAPTER 3 . EXPERIMENTAL RESULTS .....	7
3.1 HTML structure for the interface.....	7
3.1.1 Head Part .....	7
3.1.2 Library: .....	7
3.1.3 Body Part .....	7
3.1.4 CSS Styling.....	9
3.1.5 User interaction.....	2
3.1.6 Illustrative interface image .....	3

3.2	GEmini chatbot .....	4
3.2.1	Steps to get Genimi connection url and Create Gemini API.....	4
3.2.2	Results obtained.....	6
3.3	Hugging face model's chatbot .....	6
3.3.1	Steps to get Hugging face's API token.....	7
3.3.2	Steps to get url path to the model "Stable Diffusion 3.5 Large" ...	9
3.3.3	Results obtained.....	11
3.4	Handling feedback from Gemini .....	12
3.5	Handling feedback from Stable Diffusion 3.5 Large.....	13
3.5.1	Step 1: Send an HTTP POST request to the Hugging Face API.	13
3.5.2	Step 2: Wait for the response from the API .....	14
3.5.3	Step 3: Check the response data .....	14
3.5.4	Step 4: Handle errors from the API.....	14
3.5.5	Step 5: Handle exceptions (Error Handling) .....	15
3.5.6	Summary of the Process: .....	15
3.6	Building a QA (Question Answering) chatbot on School Counseling with the "roberta-base" model.....	16
3.6.1	Introduction to the roberta-base model.....	16
3.6.2	Introduction to QA datasets. TDT. FQA_tu_van_hoc_duong on Huggingface.	16
3.6.3	Building and training the model. ....	16
3.6.4	Model Public URL using FastAPI and Ngrok.....	18
3.6.5	Test the model results.....	20
CHAPTER 4 . CONCLUSION.....		23
4.1	CONCLUSION .....	23
4.2	DEVELOPMENT .....	23
REFERENCES .....		25

## LIST OF IMAGES

Figure 3.1.1. Head code .....	7
Figure 3.1.2. Library .....	7
Figure 3.1.3.Code to display video as background for browser .....	8
Figure 3.1.4. Code allows user to choose any chatbot.....	8
Figure 3.1.5. Code to display message and input box.....	8
Figure 3.1.6. Interface code (1).....	9
Figure 3.1.7. Interface code (2).....	9
Figure 3.1.8. Interface code (3).....	1
Figure 3.1.9. Code of the interface frame of the message input box and function buttons in the chatbot .....	2
Figure 3.1.10. Auto scroll effect code.....	3
Figure 3.1.11. chatbot browser interface.....	3
Figure 3.2.1. Google AI Studio interface.....	4
Figure 3.2.2. API key access interface.....	4
Figure 3.2.3. Chosing Google Cloud projects.....	5
Figure 3.2.4. Chose Google Cloud projects .....	5
Figure 3.2.5. API key after created .....	6
Figure 3.2.6. Warning after close API key generated noti box.....	6
Figure 3.3.1. huggingFace main interface.....	7
Figure 3.3.2. Access Token interface.....	8
Figure 3.3.3. Create Token interface.....	8
Figure 3.3.4. Created token announcement.....	9
Figure 3.3.5. Find stable-diffusion 3.5 large model.....	9
Figure 3.3.6. stable-diffusion 3.5 large mode details.....	10
Figure 3.3.7. stable-diffusion 3.5 large model deploy button.....	10
Figure 3.3.8. stable-diffusion 3.5 large mode JavaScript connection code .....	11

Figure 3.6.1. Chatbot test with question 1 .....	20
Figure 3.6.2. Chatbot test with question 2 .....	21
Figure 3.6.3. Chatbot test with question 3 .....	21
Figure 3.6.4. Chatbot test with question 3 .....	22

## **ABSTRACTS**

In recent years, artificial intelligence (AI) has revolutionized the way people interact with technology, especially through chatbots on many websites. However, most chatbot systems are developed for specific purposes, requiring users to switch between multiple platforms, through different websites to meet their diverse needs. This not only makes user interaction inconvenient but also limits the integration capabilities of AI solutions.

With this chatbot integration website project, we aim to develop a web-based platform that integrates multiple chatbots with separate functions, providing a seamless and unified user experience. By leveraging available chatbots such as Gemini or Hugging Face models as well as self-made chatbots, our website will bring users the most comfortable and simple experiences.

The project focuses on designing a website that can effectively manage chatbot conversations, improve response accuracy and ensure a user-friendly interface.

We would like to thank our lecturer, Nguyễn Sơn Lâm, for always caring, understanding and supporting us wholeheartedly in completing this project.

# **CHAPTER 1 . INTRODUCTION**

## **1.1 OVERVIEW OF THE TOPIC**

In the digital age, artificial intelligence (AI) has become an important tool to automate and improve user experience. Chatbots, a popular application of AI, are being widely deployed in many fields such as customer service, education, healthcare, and entertainment. However, integrating different chatbots on the same platform to serve diverse user needs is still quite limited. Therefore, we carried out this project with the aim of creating a chatbot website that integrates many chatbots of many research organizations and different brands.

## **1.2 REASONS FOR CHOOSING THE TOPIC**

- **Urgency of the problem:**

In the context of rapidly developing technology, artificial intelligence (AI) is increasingly widely applied in daily life. Chatbot, one of the typical applications of AI, not only helps improve interaction efficiency but also reduces operating costs for businesses. However, users today often have to access many different platforms or applications to use chatbots for different purposes. This makes it difficult to manage and reduces user experience.

- **Development trend:**

Chat integration platforms such as [poe.com](https://poe.com), [chatGPT.com](https://chatgpt.com) are attracting a lot of attention thanks to their ability to provide diverse and convenient experiences. This is a potential trend, not only solving the problem of dispersion but also opening up opportunities to integrate many advanced AI technologies on the same platform.

- **Applicability:**

The final project product of the group has the potential to become a useful tool for both individual users and organizations. It can be applied in many areas of life to simplify and facilitate life for everyone.

## **1.3 OBJECTIVES OF THE RESEARCH**

- **Overall goal:**



Develop a web platform that integrates multiple chatbots with different features and purposes, to provide a comprehensive solution for users to interact and answer questions with AI, thereby improving efficiency and user experience.

- **Specific objectives:**

- **Build a chatbot integration platform:**

Develop an intuitive and user-friendly web interface that allows users to access and use multiple chatbots on the same platform, one website

Support easy switching between chatbots without interrupting the user experience, saving chat histories, allowing users to proactively review previous conversations.

- **Develop diverse chatbots:**

Create and train specialized chatbots for many specific purposes such as searching for information, answering questions based on specific documents, or guiding students.

Ensure chatbots are able to respond naturally and accurately by using technologies such as Gemini, OpenAI, HuggingFace, and other models.

## **1.4 PROJECT STRUCTURE**

The report structure includes 4 main sections:

- **Chapter 1: Introduction**

Introduce the topic, overview of the topic, reasons for choosing the topic and main objectives of the topic

- **Chapter 2: Basic theory**

The chapter will introduce the programming languages, applications and environments used to create the project.

- **Chapter 3: Experiments and Results**

Explain and describe in detail each step of the project implementation, the final project results and the results obtained after testing.

- **Chapter 4: Conclusion**

Give conclusions about the project, what has been achieved, current advantages and disadvantages and provide directions for future project development.

## **CHAPTER 2 . THEORETICAL BASIS**

In this project, to complete the product, we used a total of 3 programming languages including HTML, JavaScript and Python, in which:

- HTML: used to create the web interface
- JavaScript: used to write code running inside the web
- Python: used to train the chatbot chosen by our team

In addition, we also used two more environments to complete the project, which are Visual Studio Code and Google Collab, in which:

- Visual Studio Code: A familiar coding environment used to create the web interface and code running the web
- Google Collab: An environment to train the chatbot chosen by the team

The finished product of our team's project will be a professional chatbot running website with a total of 3 chatbots including: chatbot 'Gemini' developed by Google, image-creating chatbot 'Stable Diffusion 3.5 Large' and one chatbot which was trained by ourselves.

### **2.1 HTML**

HTML stands for Hyper Text Markup Language, is the standard markup language for creating Web pages. HTML describes the structure of a Web page consists of a series of elements. These elements tell the browser how to display the content and the label pieces of content such as "this is a heading", "this is a paragraph", "this is a link", etc

### **2.2 PYTHON**

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and

packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

## **2.3 JAVASCRIPT**

JavaScript is a versatile, dynamically typed programming language used for interactive web applications, supporting both client-side and server-side development, and integrating seamlessly with HTML, CSS, and a rich standard library.

JavaScript is a single-threaded language which means it executes one task at a time. It is an Interpreted language which means it executes the code line by line. The data type of the variable is decided at run-time in JavaScript that's why it is called dynamically typed.

## **2.4 VISUAL STUDIO**

Visual Studio is a powerful developer tool that you can use to complete the entire development cycle in one place. It's a comprehensive integrated development environment (IDE) that you can use to write, edit, debug, and build code. Then deploy your app. Visual Studio includes compilers, code completion tools, source control, extensions, and many other features to enhance every stage of the software development process.

With the variety of features and languages support in Visual Studio, you can grow from writing your first "Hello World" program to developing and deploying apps. For example, build, debug, and test .NET and C++ apps, edit ASP.NET pages in the web designer view, develop cross-platform mobile and desktop apps with .NET, or build responsive Web UIs in C#.

## **2.5 GOOGLE COLLAB**

Google Colaboratory is a hosted Jupyter Notebook service that requires no setup to use and provides free access to computing resources, including GPUs and TPUs. Colab is especially well suited to machine learning, data science, and education.

Google Colab, short for Google Colaboratory, is a free cloud service provided by Google that allows users to run Python code real-time in a browser-based environment. It offers a convenient way to write, execute, and share Python code along with its output.

## **2.6 GEMINI AI**

Gemini is the result of large-scale collaborative efforts by teams across Google, including our colleagues at Google Research. It was built from the ground up to be multimodal, which means it can generalize and seamlessly understand, operate across and combine different types of information including text, code, audio, image and video.

Gemini is also our most flexible model yet — able to efficiently run on everything from data centers to mobile devices. Its state-of-the-art capabilities will significantly enhance the way developers and enterprise customers build and scale with AI.

## **2.7 HUGGING FACE**

Hugging Face Transformers is an open-source Python library that provides access to thousands of pre-trained Transformers models for natural language processing (NLP), computer vision, audio tasks, and more. It simplifies the process of implementing Transformer models by abstracting away the complexity of training or deploying models in lower level ML frameworks like PyTorch, TensorFlow and JAX.

The Hugging Face Hub is a platform with over 900k models, 200k datasets, and 300k demo apps (Spaces), all open source and publicly available, in an online platform where people can easily collaborate and build ML together. The Hub works as a central place where anyone can explore, experiment, collaborate, and build technology with Machine Learning.

## CHAPTER 3 . EXPERIMENTAL RESULTS

### 3.1 HTML STRUCTURE FOR THE INTERFACE

#### 3.1.1 HEAD PART

```
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Chat Bot Tui Làm</title>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/animejs/3.2.1/anime.min.js"></script>
  <script src="https://cdn.jsdelivr.net/npm/marked/marked.min.js"></script>
  <link href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.0.0-beta3/css/all.min.css" rel="stylesheet">
  <style>
```

*Figure 3.1.1. Head code*

**<meta charset="UTF-8">**: Ensures character code allows display of Vietnamese and special characters.

**<meta name="viewport" content="width=device-width, initial-scale=1.0">**: Help the interface display correctly.

**<title>Chat Bot Tui Làm</title>**: Set a title for the page, it will be displayed in the browser title bar.

#### 3.1.2 LIBRARY:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/animejs/3.2.1/anime.min.js"></script>
<script src="https://cdn.jsdelivr.net/npm/marked/marked.min.js"></script>
<link href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.0.0-beta3/css/all.min.css" rel="stylesheet">
```

*Figure 3.1.2. Library*

**Anime.js**: Animated effects library, helps create effects, making the interface come alive.

**Marked.js**: Helps display text content more beautifully.

**Font Awesome**: Displays beautiful icons, such as the send message button.

#### 3.1.3 BODY PART

- Video nền:

```

<body>
  <video autoplay muted loop>
    <source src="background1.mp4" type="video/mp4">
  </video>

```

*Figure 3.1.3. Code to display video as background for browser*

The purpose is to create a dynamic background effect, autoplay video and loop.

- Sidebar:

```

<div style="display: flex; flex-direction: column; gap: 10px;">
  <button onclick="switchBot('gemini')" class="bot-btn">Gemini</button>
  <div id="gemini-history" class="bot-history"></div>
  <button onclick="switchBot('textToImage')" class="bot-btn">Text to image</button>
  <div id="textToImage-history" class="bot-history"></div>
  <button onclick="switchBot('newBot')" class="bot-btn">New Bot</button>
  <div id="newBot-history" class="bot-history"></div>
</div>
</div>

```

*Figure 3.1.4. Code allows user to choose any chatbot*

Contains chatbot selection buttons, allowing users to easily select any chatbot they want in the browser.

- Main Part:

```

<div style="flex: 1; display: flex; flex-direction: column;">
  <div id="messages" style="flex: 1; padding: 20px; overflow-y: auto;">
    <div class="welcome-message">
      <h1>Welcome to My Chanel</h1>
      <p>Select a chatbot from the side panel to start chatting</p>
    </div>
  </div>
</div>

```

*Figure 3.1.5. Code to display message and input box*

Contains chatbot selection buttons, allowing users to easily select any chatbot they want in the browser.

### 3.1.4 CSS STYLING

```
<style>
body {
  margin: 0;
  padding: 0;
  font-family: 'Roboto', Arial, sans-serif;
  color: #ffffff;
  overflow: hidden;
  position: relative;
  background-color: #000000;
}

video {
  position: absolute;
  top: 0;
  left: 0;
  width: 100%;
  height: 100%;
  object-fit: cover;
  z-index: -1;
}

.content {
  position: relative;
  z-index: 1;
  padding: 20px;
}

h2 {
  margin-bottom: 30px;
  text-align: center;
  color: #ffffff;
  transition: color 0.3s;
}

.message {
  margin: 10px;
  padding: 10px 15px;
  border-radius: 8px;
  max-width: 75%;
  word-wrap: break-word;
  white-space: pre-wrap;
  display: inline-block;
  vertical-align: top;
  background-color: rgba(205, 198, 198, 0.539);
}

.message.user-message {
  background-color: rgba(205, 198, 198, 0.833);
  align-self: flex-end;
  text-align: right;
}
```

Figure 3.1.6. Interface code (1)

```
.message.bot-message {
  background-color: rgba(205, 198, 198, 0.806);
  align-self: flex-start;
  text-align: left;
}

#messages {
  flex: 1;
  padding: 20px;
  background-color: rgba(80, 79, 79, 0.2);
  color: #000000;
  overflow-y: auto;
  display: flex;
  flex-direction: column;
  gap: 10px;
}

textarea {
  width: 80%;
  padding: 10px;
  margin-bottom: 10px;
  border-radius: 5px;
  border: 1px solid #ddd;
  background-color: #fff;
}

button {
  padding: 10px 20px;
  border: none;
  background-color: #0000006f;
  color: white;
  cursor: pointer;
  border-radius: 5px;
}

#sidebar {
  width: 250px;
  background: #504f4fb3;
  padding: 20px;
  border-right: 1px solid #eaeaea;
  position: relative;
  z-index: 1;
}

#sidebar h2 {
  margin-bottom: 30px;
  color: #ffffff;
  background-color: rgba(24, 23, 23, 0.7);
  padding: 10px;
  border-radius: 8px;
}
```

Figure 3.1.7. Interface code (2)



```

        .bot-btn {
            padding: 10px;
            border: none;
            background: ■ #111111;
            color: □ #ffffff;
            border-radius: 8px;
            cursor: pointer;
        }
        .bot-history {
            padding-left: 20px;
            display: none;
            color: □ #ffffff;
        }
        .welcome-message {
            text-align: center;
            margin-top: 40px;
            color: □ #ffffff;
        }
    }
</style>
</head>

```

*Figure 3.1.8. Interface code (3)*

- Set fonts, colors, and backgrounds for the entire page
- Use flexbox to position elements.
- Separate formatting for user and bot messages, making it easy to differentiate.

### 3.1.5 USER INTERACTION

```
<div style="padding: 20px; border-top: 1px solid #eaeaea;">
  <div style="display: flex; gap: 10px;">
    <textarea id="userInput" placeholder="Type your message..." style="flex: 1;
    font-family: 'Roboto', Arial, sans-serif;
    padding: 15px;
    border-radius: 8px;
    border: 1px solid #eaeaea; resize: none;"></textarea>

    <button onclick="sendMessage()" style="padding: 15px 20px;
    background: #000000;
    color: #fff;
    font-weight: bold;
    border: none;
    border-radius: 12px;
    font-size: 14px;
    cursor: pointer;
    font-family: 'Roboto', Arial, sans-serif;">
      <i class="fas fa-paper-plane"></i>
    </button>

    <button onclick="createNewChat()" style="padding: 15px 20px;
    background: #333333;
    font-weight: bold;
    color: #fff;
    border: none;
    border-radius: 12px;
    font-size: 14px;
    font-family: 'Roboto', Arial, sans-serif;
    cursor: pointer;">
      New Chat
    </button>
  </div>
</div>
```

*Figure 3.1.9. Code of the interface frame of the message input box and function buttons in the chatbot*

**<div style="display: flex; gap: 10px;">**: Use Flexbox to align elements in rows, creating space between elements (textarea and button).

**placeholder="Type your message..."**: Display text in the chat box to remind the user.

**onclick="sendMessage()"**: Function called to send a message.

**onclick="createNewChat()"**: Function called to start a new chat.

- Dynamic response:

```
function addMessage(sender, text) {
    const messagesDiv = document.getElementById('messages');

    const messageDiv = document.createElement('div');
    messageDiv.className = 'message'; // Đảm bảo lớp đúng
    messageDiv.innerHTML = text;

    // Đặt màu nền và vị trí dựa trên người gửi
    if (sender === 'user') {
        messageDiv.classList.add('user-message');
        messageDiv.style.alignSelf = 'flex-end'; // Tin nhắn người dùng bên phải
    } else {
        messageDiv.classList.add('bot-message');
        messageDiv.style.alignSelf = 'flex-start'; // Tin nhắn bot bên trái
    }

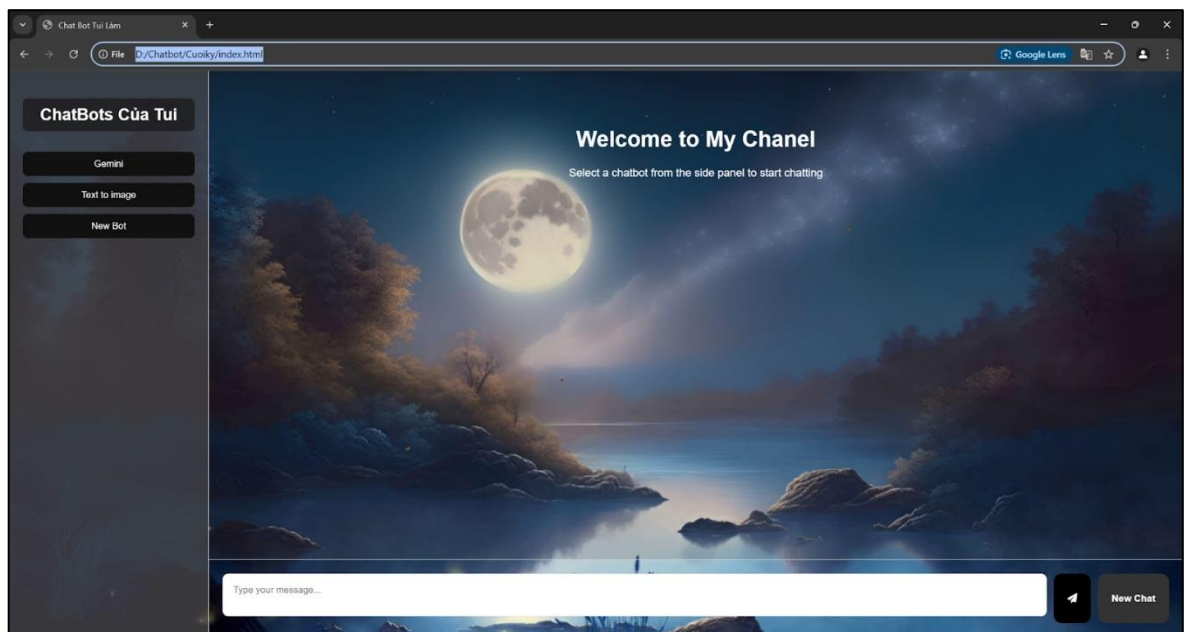
    messagesDiv.appendChild(messageDiv);
    messagesDiv.scrollTop = messagesDiv.scrollHeight;

    anime({
        targets: messageDiv,
        translateY: [20, 0],
        opacity: [0, 1],
        duration: 500,
        easing: 'easeOutCubic'
    });
}
```

*Figure 3.1.10. Auto scroll effect code.*

Add messages to the display, with automatic scrolling so users always see the latest messages..

### 3.1.6 ILLUSTRATIVE INTERFACE IMAGE

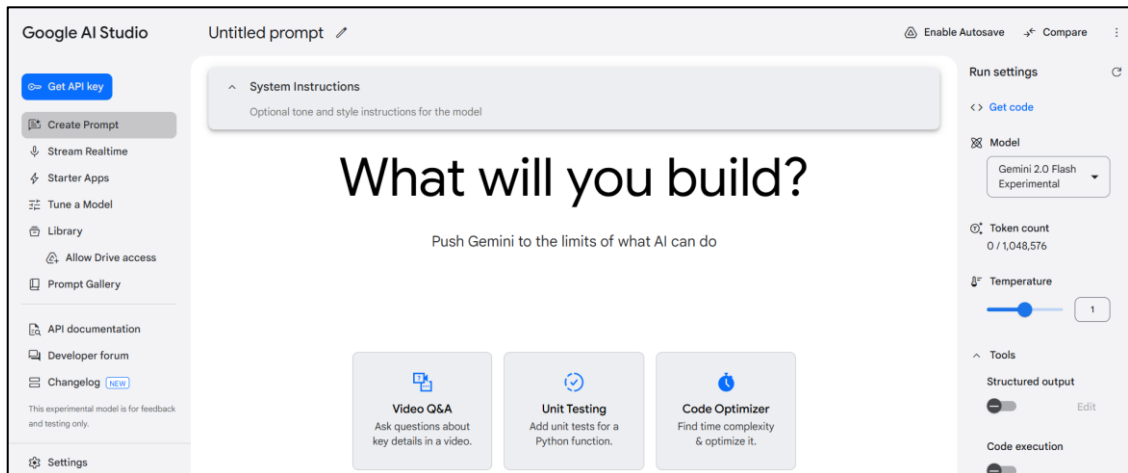


*Figure 3.1.11. chatbot browser interface*

## 3.2 GEMINI CHATBOT

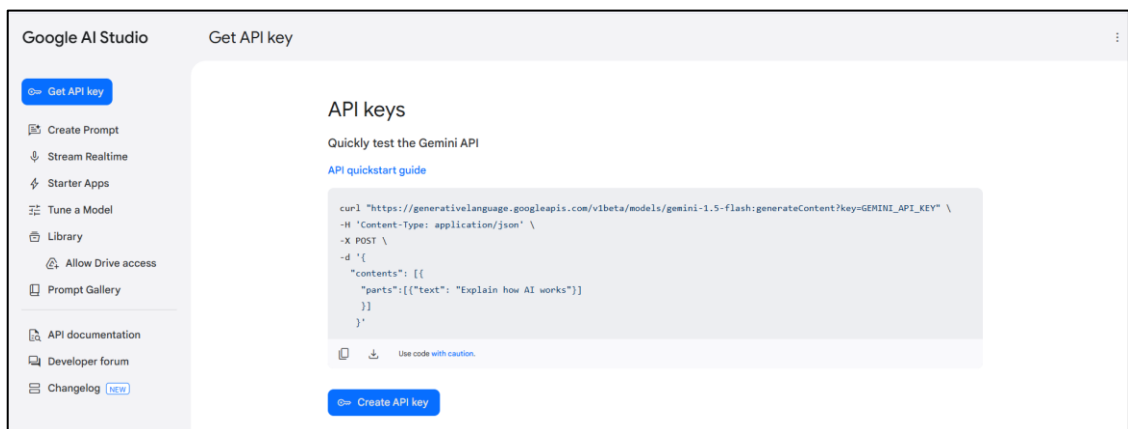
### 3.2.1 STEPS TO GET GENIMI CONNECTION URL AND CREATE GEMINI API

First step, we will access this link: "<https://aistudio.google.com>" to access the Google AI Studio website with the purpose of creating an API key to hook into our project (log in and register to the website with a google account if necessary).



*Figure 3.2.1. Google AI Studio interface*

Next, we will go to the next page: "<https://aistudio.google.com/apikey>" to be able to create an API key (or we can click on the "Get API key" button to quickly access from the initial default website).



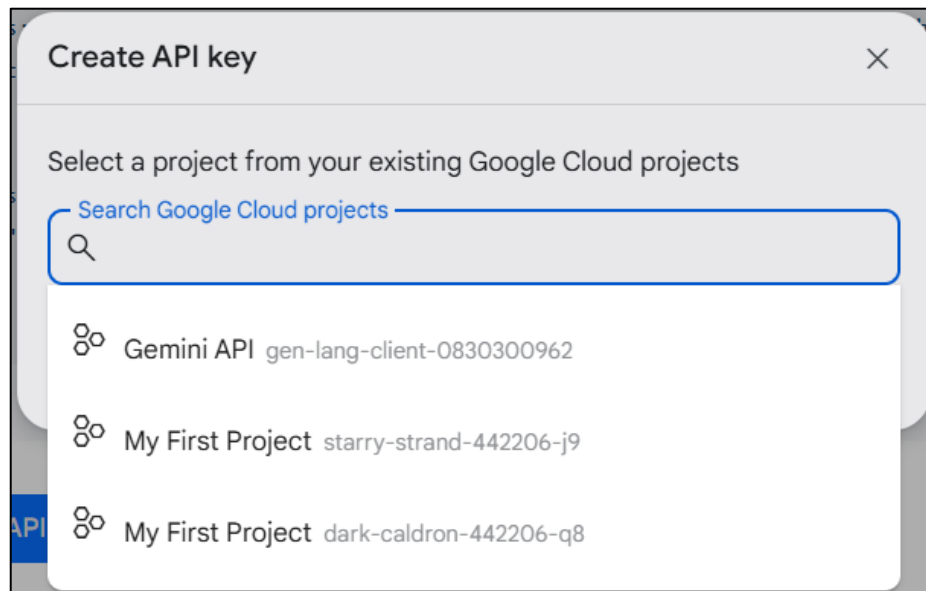
*Figure 3.2.2. API key access interface*

Furthermore, here we can see the url to connect to Google's Gemini in the "API quick start guide" is:

“https://generativelanguage.googleapis.com/v1beta/models/gemini-1.5-flash:generateContent?key=GEMINI\_API\_KEY”.

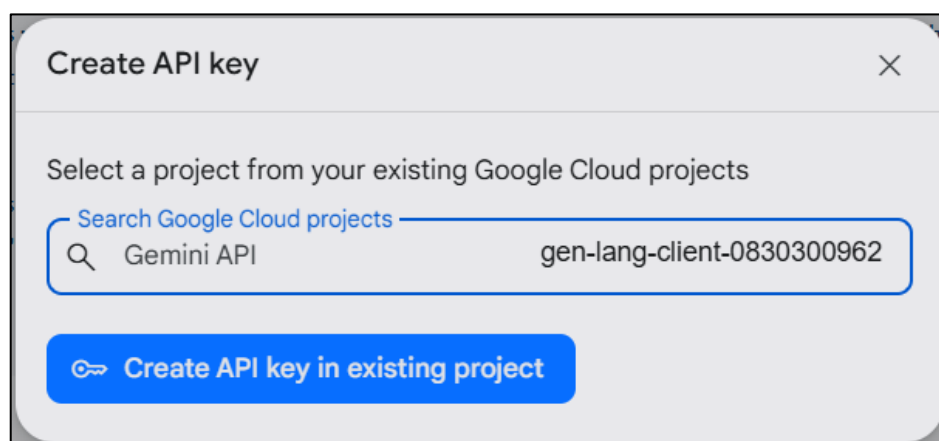
That means we have gone 50% of the way. Next we will continue to create API keys.

Next step, click on the 'create API key' button, a dialog box will appear to select the Project to be able to create the API key.



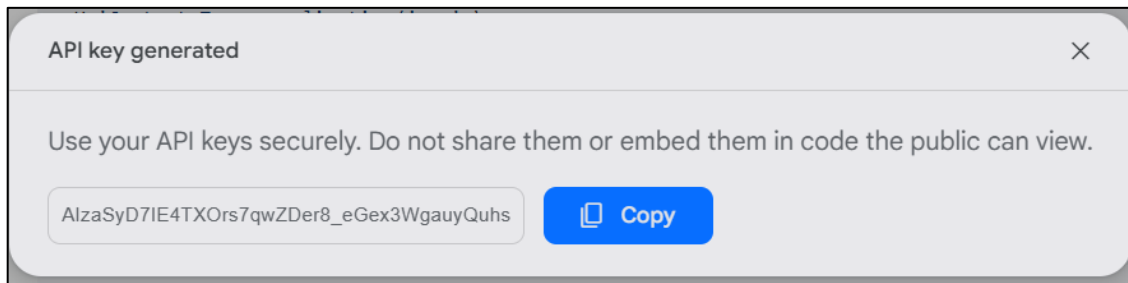
*Figure 3.2.3. Chosing Google Cloud projects*

I will choose "Gemini API" as the main project.



*Figure 3.2.4. Chose Google Cloud projects*

Finally, click on the "Create API key in existing project" button to create an API key.



*Figure 3.2.5. API key after created*

Once completed, we obtained Gemini's "API key" to be able to integrate into the chatbot.

Your API keys are listed below. You can also view and manage your project and API keys in Google Cloud.

Project number	Project name	API key	Created	Plan
...3267	Gemini API	...Quhs	Dec 31, 2024	Paid <a href="#">Go to billing</a> <a href="#">View usage data</a>

Remember to use API keys securely. Don't share or embed them in public code. Use of Gemini API from a billing-enabled project is subject to [pay-as-you-go pricing](#).

*Figure 3.2.6. Warning after close API key generated noti box*

### 3.2.2 RESULTS OBTAINED

After launching each step in section 3.1.1, we have obtained important data to be able to complete hooking the chatbot to the website including:

- Url to connect to Gemini:

“[https://generativelanguage.googleapis.com/v1beta/models/gemini-1.5-flash:generateContent?key=GEMINI\\_API\\_KEY](https://generativelanguage.googleapis.com/v1beta/models/gemini-1.5-flash:generateContent?key=GEMINI_API_KEY)”

- Google API key:

“AIzaSyD7IE4TXOrs7qwZDer8\_eGex3WgauyQuhs”

### 3.3 HUGGING FACE MODEL’S CHATBOT

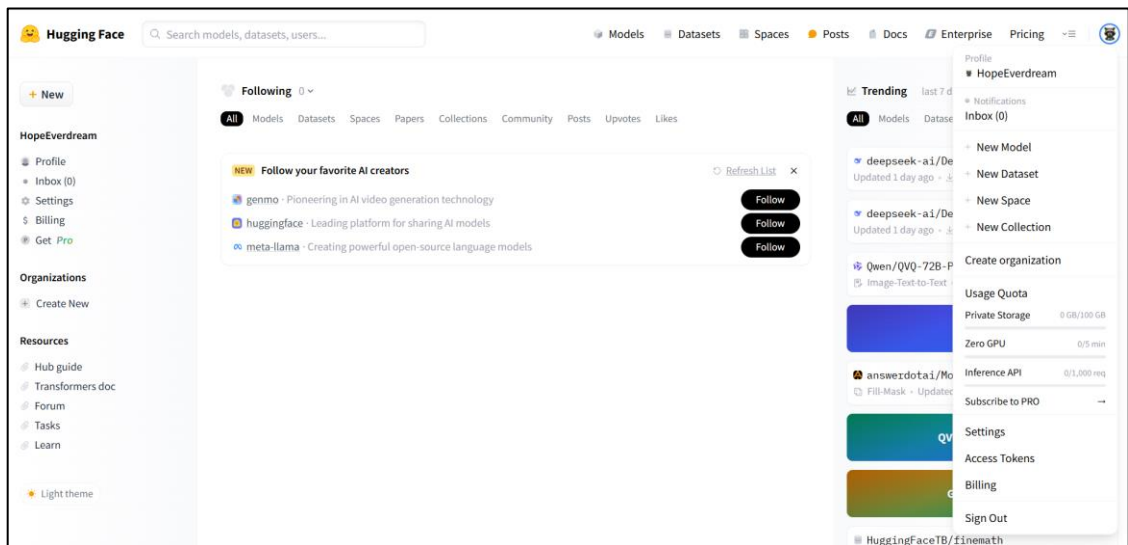
For Huggingface, we will use a model included in Huggingface, the "Stable Diffusion 3.5 Large" model whose main function is to generate images from text.

Stable Diffusion 3.5 Large is a Multimodal Diffusion Transformer (MMDiT) text-to-image model that features improved performance in image quality, typography, complex prompt understanding, and resource-efficiency.

### 3.3.1 STEPS TO GET HUGGING FACE'S API TOKEN

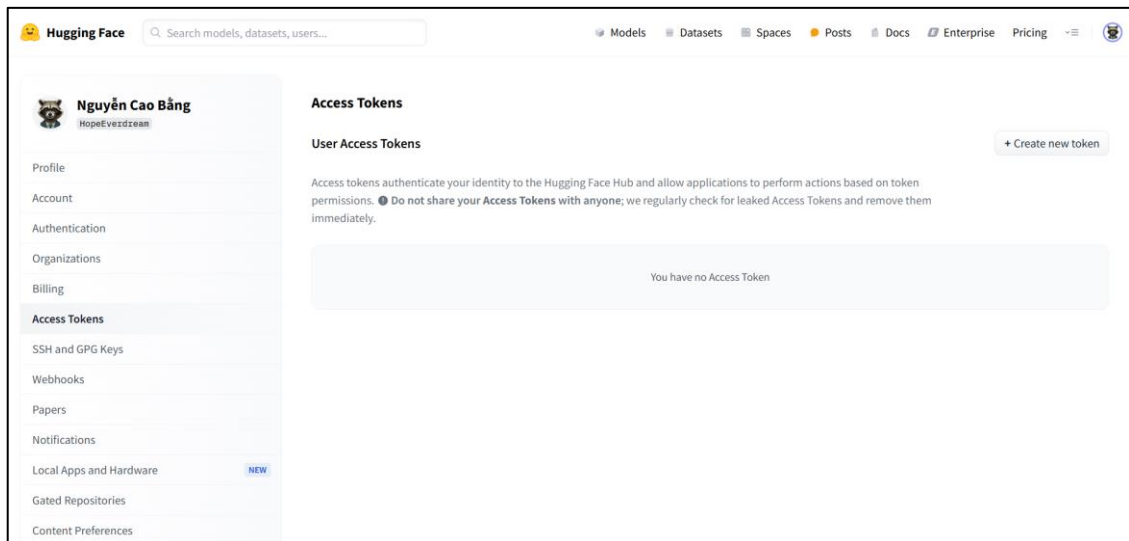
First, we will log in to the huggingface website at the following link: "https://huggingface.co/" (please log in by clicking on the avatar icon in the left corner if necessary).

Next, after logging in, we will select "Access Token" with the purpose of creating an API token to be able to connect to huggingface in general and the model we have chosen in particular.



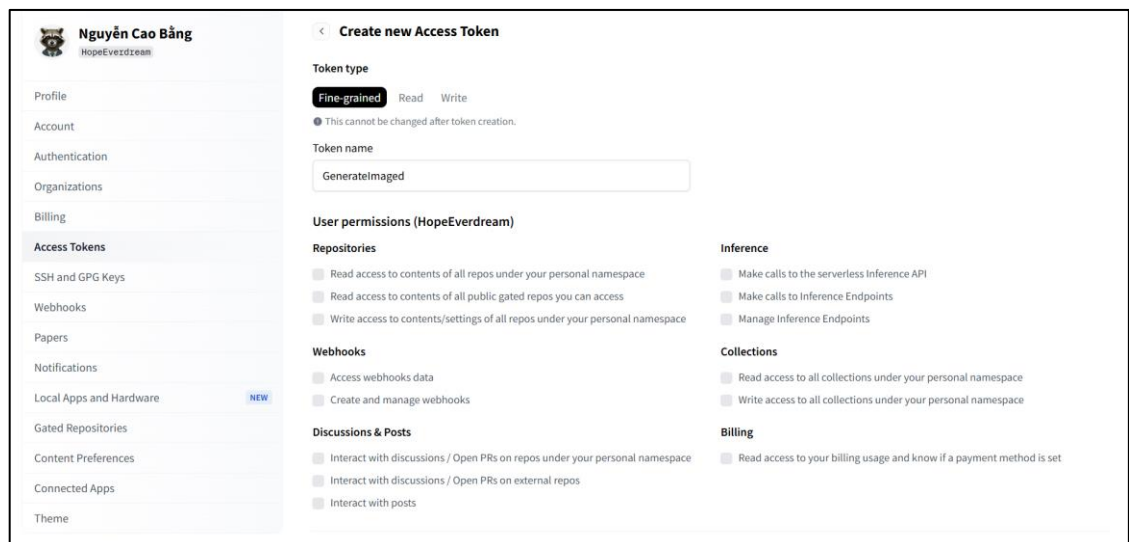
*Figure 3.3.1. huggingFace main interface*

After clicking on the "Access Token" button, a new interface page will appear, at this time if you want to create a new API token, click on the "Create New Token" button.



*Figure 3.3.2. Access Token interface*

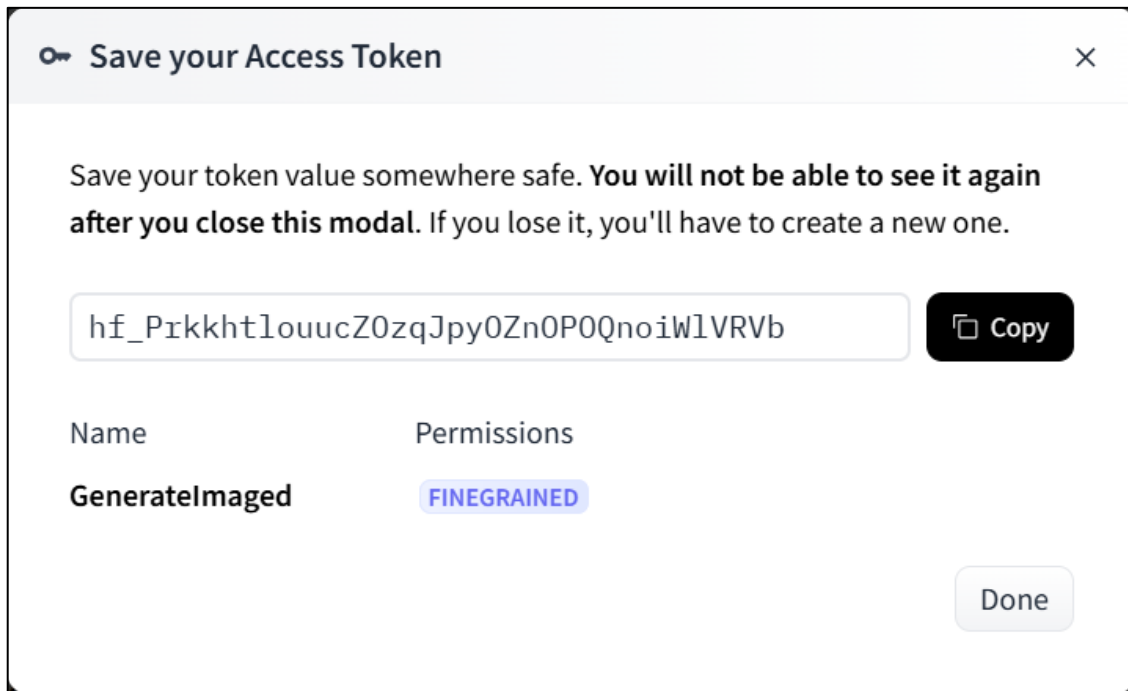
After clicking on the "Create New Token" button, a new interface will appear, here name the token and adjust the settings you want for your token. For the group, we will keep all the settings and then scroll to the bottom of the page to click "Create Token".



*Figure 3.3.3. Create Token interface*

After completing, the group's Token will be displayed on the small interface. As noted, the token will not be viewable again, so please save the token in a safe place.



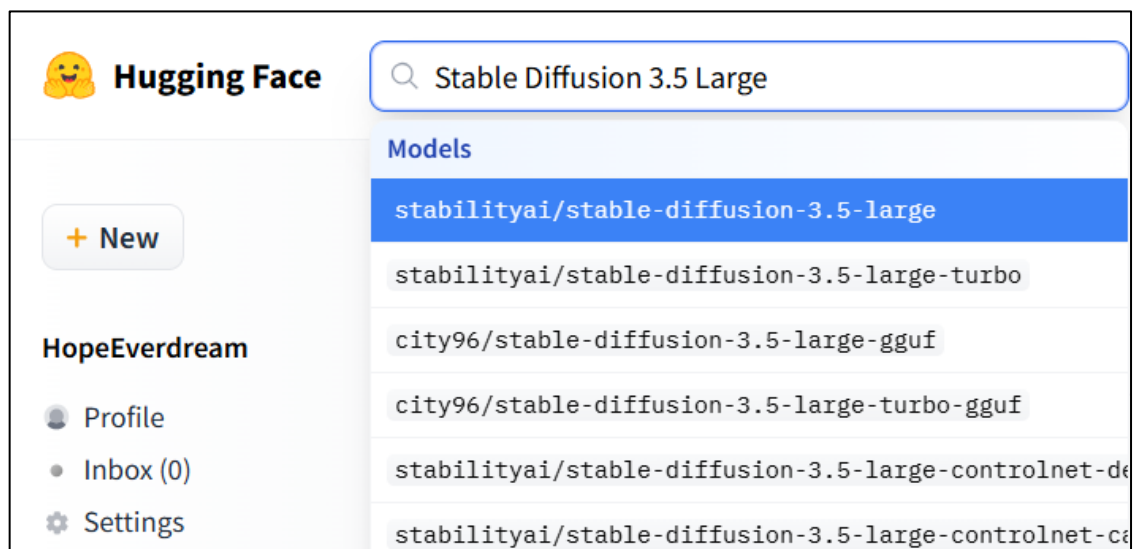


*Figure 3.3.4. Created token announcement*

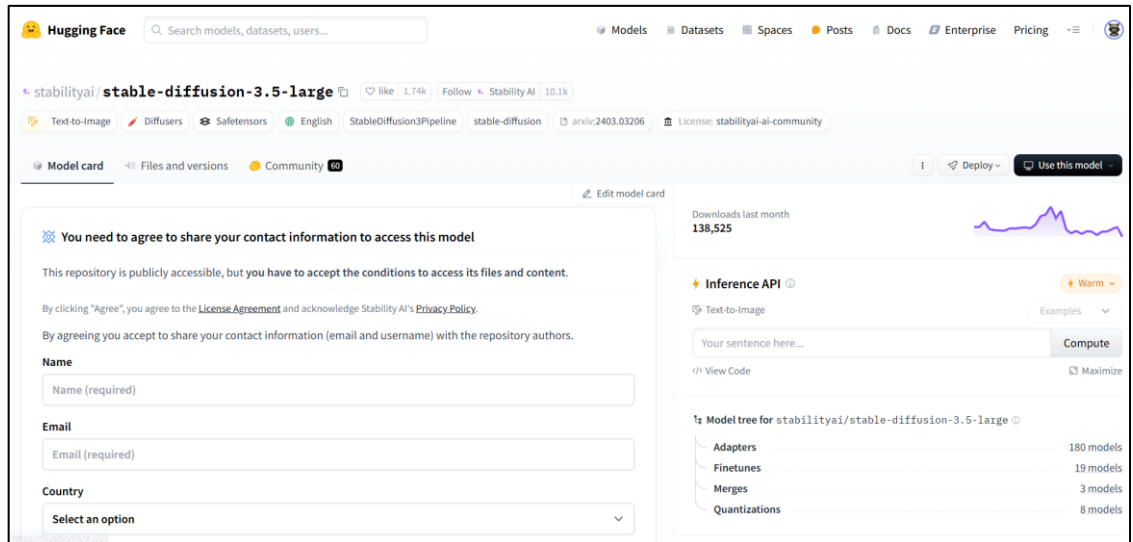
And finally we got the HuggingFace API token.

### 3.3.2 STEPS TO GET URL PATH TO THE MODEL "STABLE DIFFUSION 3.5 LARGE"

First step, go back to the main page at this link: "<https://huggingface.co/>". In the search bar on the top left corner, type in the model "stable diffusion 3.5 large", if the result is: "stabilityai/stable-diffusion-3.5-large" it means it is correct and click on it. Then we will be taken to the website containing information about that model.

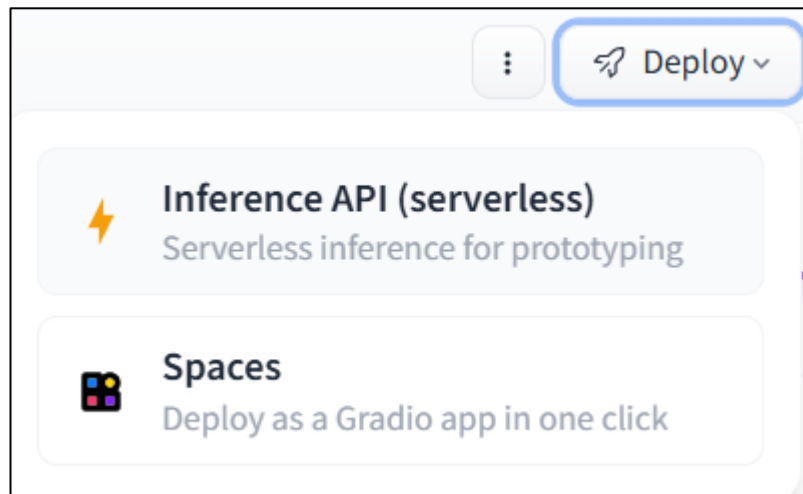


*Figure 3.3.5. Find stable-diffusion 3.5 large model*




*Figure 3.3.6. stable-diffusion 3.5 large mode details*

Next, search on the current website for the "deploy" button and click on it, then select "Inference API", this is where the url leading to the model is located.



*Figure 3.3.7. stable-diffusion 3.5 large model deploy button*

Next, depending on the programming language you are using, choose the appropriate item. Here, because our web creation code is written in HTML and JavaScript, we will choose JavaScript.



```
Use this model with the • Inference API (serverless) ×

Python JavaScript cURL Manage tokens

Copy

async function query(data) {
  const response = await fetch(
    "https://api-inference.huggingface.co/models/stabilityai/stable-diffusion-3.5-large",
    {
      headers: {
        Authorization: "Bearer hf_XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
        "Content-Type": "application/json",
      },
      method: "POST",
      body: JSON.stringify(data),
    }
  );
  const result = await response.blob();
  return result;
}

query({"inputs": "Astronaut riding a horse"}).then((response) => {
  // Use image
});
```

*Figure 3.3.8. stable-diffusion 3.5 large mode JavaScript connection code*

Here we see what we're looking for, the url code that connects to the "Stable Diffusion 3.5 Large" model:

“https://api-inference.huggingface.co/models/stabilityai/stable-diffusion-3.5-large”

### 3.3.3 RESULTS OBTAINED

After launching each step in section 3.1.1, we have obtained important data to be able to complete hooking the "Stable Diffusion 3.5 Large" model to the website including:

- Model's Url:

“https://api-inference.huggingface.co/models/stabilityai/stable-diffusion-3.5-large”

- HuggingFace's Token:

“hf\_PrkkhtlouucZOzqJpyOZnOPOQnoilVRVb”

### 3.4 HANDLING FEEDBACK FROM GEMINI

- Request details to the API.
- Send an HTTP POST request to the Gemini API.
- Attach API key for authentication.
- Send input data (content from input variable) as JSON.

```
const geminiResponse = await fetch(`${endpoints.gemini}?key=AIzaSyD8t4-eT6YTEpJfAKnYTtLa4wPRX8vk05U`, {  
  method: 'POST',  
  headers: { 'Content-Type': 'application/json' },  
  body: JSON.stringify({  
    contents: [{ parts: [{ text: input }] }]  
  })  
});
```

- Request options: The data is converted to a JSON string.

```
JSON.stringify({  
  contents: [{ parts: [{ text: input }] }]  
})
```

- Waiting for response: Use the await keyword to wait for the API to respond before continuing execution.

```
const geminiResponse = await fetch(...);
```

- Then Process the response:

Parse the response from the API (usually in JSON format) into a JavaScript object.

Save to variable geminiData

Record response data to the dashboard

```
const geminiData = await geminiResponse.json();  
console.log(geminiData);
```

- Check response data
- Check conditions
- If condition is true: Returns the text content from the response.
- If condition is false: Returns a user error message.

```

if (geminiData.candidates && geminiData.candidates[0].content && geminiData.candidates[0].content.parts[0].text) {
    return geminiData.candidates[0].content.parts[0].text;
} else {
    return "Sorry, there was an issue with Gemini's response.";
}

```

- Error handling
- Log error information to the console
- error message: Returns an error message to the user if a problem occurred while sending the request or processing the response.

```

catch (error) {
    console.error('Error:', error);
    return "Sorry, there was an error processing your request.";
}

```

## 3.5 HANDLING FEEDBACK FROM STABLE DIFFUSION 3.5 LARGE

### 3.5.1 STEP 1: SEND AN HTTP POST REQUEST TO THE HUGGING FACE API

First, we need to send an HTTP POST request to the Hugging Face API, specifically to call a model like Gemini (or any other bot you choose), passing in the data you want to process (such as content from the user) and the API key for authentication.

```

const response = await fetch(`${endpoints.gemini}?key=YOUR_API_KEY`, {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json',
    },
    body: JSON.stringify({
        contents: [{ parts: [{ text: input }] }] // Data is sent in JSON format
    })
});

```

Fetch is the method to send the HTTP request. We use POST to send data to the API.

The headers section specifies the data type (here application/json).

The body contains the data you want to send, where input is the user input wrapped in a JSON format.

### 3.5.2 STEP 2: WAIT FOR THE RESPONSE FROM THE API

After the request is sent, we need to wait for the response from the Hugging Face API. This is done using the `await` keyword to ensure the code doesn't continue execution until the response is received.

```
const geminiData = await response.json();
```

`await` waits for the result of `response.json()`, which converts the API response from JSON format into a JavaScript object (`geminiData`).

### 3.5.3 STEP 3: CHECK THE RESPONSE DATA

Once the response is received, we need to check the data to ensure it's valid. If the API returns valid data, we'll extract the necessary information and process it.

```
if (response.ok) {  
  if (geminiData.candidates && geminiData.candidates[0].content && geminiData.candidates  
    const botResponse = geminiData.candidates[0].content.parts[0].text;  
    return botResponse;  
  } else {  
    return "Sorry, there was an issue with Gemini's response.";  
  }  
}
```

`response.ok` checks whether the request was successful (HTTP status 200–299).

The Hugging Face response may contain multiple parts, so we need to check if `candidates[0].content.parts[0].text` exists. If it does, we extract and return that text.

If the response data is not valid, we return an error message.

### 3.5.4 STEP 4: HANDLE ERRORS FROM THE API

If the API doesn't return valid data or there is an issue with the request, we need to handle the error and inform the user.

```
else {  
    const errorData = await response.json();  
    console.error('Error from Hugging Face API:', errorData);  
    return `Error: ${errorData.error || 'Unknown error occurred.'}`;  
}
```

If the response is not successful (response.ok is false), we try to extract error information from response.json() and log it to the console.

Any specific error will be returned to the user.

### 3.5.5 STEP 5: HANDLE EXCEPTIONS (ERROR HANDLING)

There might be unexpected errors, such as network issues, API being down, or misconfiguration. We need to catch these errors and handle them properly.

```
catch (error) {  
    console.error('Error occurred:', error);  
    return "Sorry, there was an error processing your request.";  
}
```

If any errors occur during the request or response processing (such as network errors or unexpected issues in the code), they will be caught by the catch block.

The error is logged to the console, and a generic error message is returned to the user.

### 3.5.6 SUMMARY OF THE PROCESS:

1. **Send request:** User data is sent to the API.
2. **Wait for response:** The code waits for the API response.
3. **Check and process response:** If the data is valid, it's returned; otherwise, an error message is shown.
4. **Handle API errors:** If the API response isn't successful, we handle the error.
5. **Handle exceptions:** Any unexpected errors (network or other) are caught and logged.

## **3.6 BUILDING A QA (QUESTION ANSWERING) CHATBOT ON SCHOOL COUNSELING WITH THE "ROBERTA-BASE" MODEL.**

### **3.6.1 INTRODUCTION TO THE ROBERTA-BASE MODEL.**

“roberta-base” is a transformer model developed by Facebook AI, based on the BERT architecture. This model is designed to handle natural language comprehension tasks and has been pre-trained on a large dataset to improve performance in a variety of NLP tasks such as sentiment analysis, text classification, and questioning.

### **3.6.2 INTRODUCTION TO QA DATASETS. TDT. FQA\_TU\_VAN\_HOC\_DUONG ON HUGGINGFACE.**

Dataset QA. TDT. FQA\_tu\_van\_hoc\_duong is a dataset used in the field of Question Answering in Vietnam, usually related to questions in the field of education. Here is some general information about this dataset:

Here is the link to access the dataset on Huggingface:

“[https://huggingface.co/datasets/BroDeadlines/QA.TDT.FQA\\_tu\\_van\\_hoc\\_duong](https://huggingface.co/datasets/BroDeadlines/QA.TDT.FQA_tu_van_hoc_duong).”

This dataset was posted by a user on Huggingface called " BroDeadlines", this dataset includes 215 frequently asked questions of college students who often ask questions about school problems.

### **3.6.3 BUILDING AND TRAINING THE MODEL.**

In order to build and train, we need to add the necessary libraries for the process of building the following are the necessary libraries:

```
import io
import json
import torch
import uvicorn
import numpy as np
import nest_asyncio
from pyngrok import ngrok
from pydantic import BaseModel
from datasets import load_dataset
from fastapi.responses import JSONResponse
```



```

from tensorflow.keras.models import load_model
from sklearn.preprocessing import LabelEncoder
from fastapi.middleware.cors import CORSMiddleware
from fastapi import FastAPI, File, UploadFile, HTTPException
from transformers import AutoTokenizer,
AutoModelForSequenceClassification, Trainer, TrainingArguments

```

Next is to download the dataset from Huggingface:

```
dataset = load_dataset("BroDeadlines/QA.TDT.FQA_tu_van_hoc_duong")
```

Label the answers

```

answers = dataset['train']['answer']
label_encoder = LabelEncoder()
labels = label_encoder.fit_transform(answers)

```

Encode the question and label it accordingly to each answer in a row of the dataset.

```

def preprocess_function(examples):
    inputs = tokenizer(examples['question'], truncation=True,
padding='max_length', max_length=512)
    inputs['labels'] = labels[:len(examples['question'])]
    return inputs

tokenized_datasets = dataset.map(preprocess_function, batched=True)
num_labels = len(set(labels))

```

Load the Model for the sequence classification task with the model you inserted.

```
model = AutoModelForSequenceClassification.from_pretrained("roberta-base", num_labels=len(set(labels)))
```

Start defining the parameters for the model training process using the TrainingArguments class from the Hugging Face Transformers library:

```
training_args = TrainingArguments(
```

```

    output_dir='./results',
    evaluation_strategy="epoch",
    save_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=100,
    report_to="none",
)

```

Create a Trainer object from the Hugging Face Transformers library

```

trainer = Trainer(
    model=model,
    args=training_args,

```

```

train_dataset=tokenized_datasets['train'],
eval_dataset=tokenized_datasets['train'],
)

```

And finally, start training the model:

```

trainer.train()

```

### 3.6.4 MODEL PUBLIC URL USING FASTAPI AND NGROK.

Set up a web application using FastAPI and configure CORS (Cross-Origin Resource Sharing) middleware.

```

app = FastAPI()

origins = [
    "http://127.0.0.1:5500",
    "http://localhost:5500",
    "https://<your-ngrok-url>.ngrok.io"
]
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

```

Define a class called Question, which inherits from BaseModel. This layer will be used to define the structure of the data that the API will receive and return.

```

class Question(BaseModel):
    question: str

```

Create a function to perform the prediction of the answer from the user's input question.

```

def predict(question):

    try:
        print(f"Processing question: {question}") # Ghi log câu hỏi
        inputs = tokenizer(question, return_tensors='pt').to('cpu')
        with torch.no_grad():
            outputs = model(**inputs)

        logits = outputs.logits
        probabilities = torch.softmax(logits, dim=-1)
        answer_index = probabilities.argmax(dim=-1)
        answer_probability = probabilities.max(dim=-1).values.item()

        print(f"Predicted index: {answer_index}, Probability:
{answer_probability}")
        return answer_index.item(), answer_probability,
    except Exception as e:
        print(f"Error in predict: {str(e)}")
        raise

```

Create a predictive return endpoint.

```
@app.post("/predict/")
```

Define a `get_prediction` asynchronous function, which is used to process predictions for a question.

```
async def get_prediction(question: Question):
    try:
        print(f"Received question: {question.question}")
        answer_index, answer_probability = predict(question.question)
        print(f"Predicted index: {answer_index}, Probability: {answer_probability}")

        if isinstance(answer_index, torch.Tensor):
            answer_index = answer_index.item()

        decoded_answer =
label_encoder.inverse_transform([answer_index])
        print(f"Decoded answer: {decoded_answer}")

        answer_text = decoded_answer[0] if hasattr(decoded_answer,
'__iter__') else decoded_answer
        if len(answer_text) > 750:
            chunks = [answer_text[i:i + 750] for i in range(0,
len(answer_text), 750)]
        else:
            chunks = [answer_text]

        return {"answer": chunks}
    except Exception as e:
        print(f"Error: {str(e)}")
        raise HTTPException(status_code=500, detail=str(e))
```

Launch FastAPI and Ngrok.

```
def start_ngrok():

    public_url = ngrok.connect(8000)
    print(f"Lấy link này để đổi: NgrokTunnel + /predict/")
    print(f"Public URL: {public_url}")

def start_fastapi():
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

After launching, ngrok will create a public URL so that website applications can send prediction requests to the model and send feedback back to the website. Formatted.

URL: <https://a3d2-34-127-120-22.ngrok-free.app>.

Then just create a copy link in js to process the request and response in javascript in Visuastudio Code.

```
const ngrokUrl = 'https://a3d2-34-127-120-22.ngrok-free.app/predict/';
```

### 3.6.5 TEST THE MODEL RESULTS.

Model results with input : Thủ tục và cách thức gia hạn đóng học phí như thế nào?

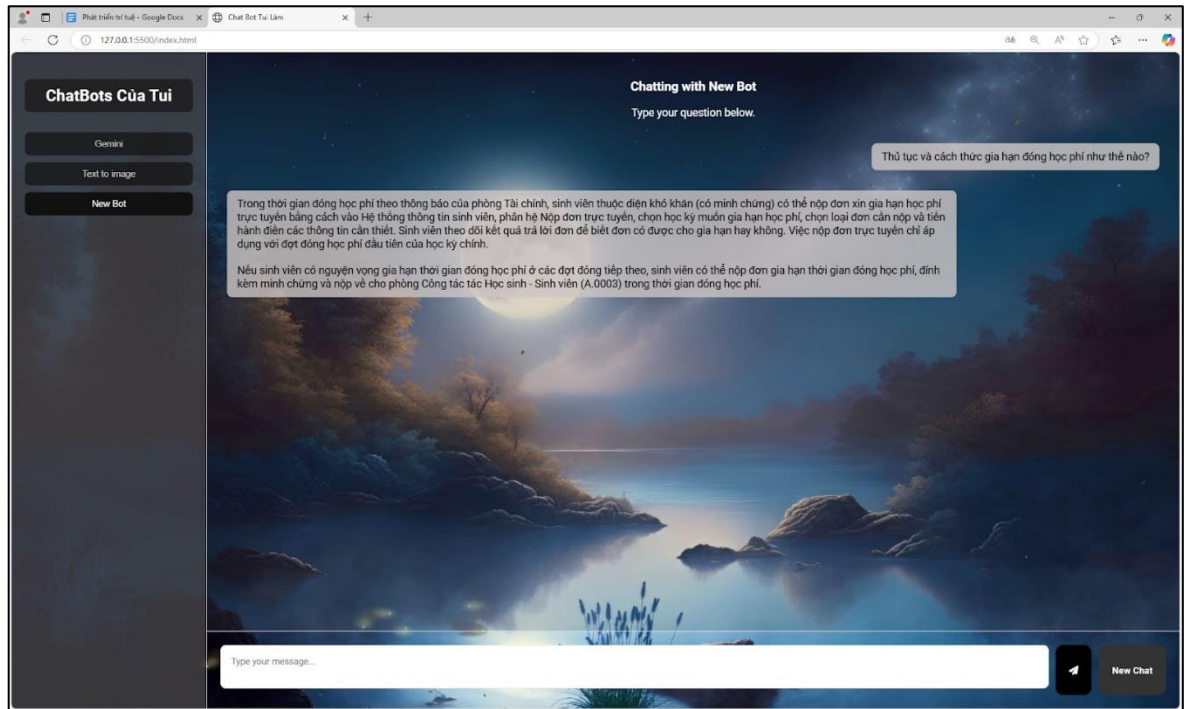
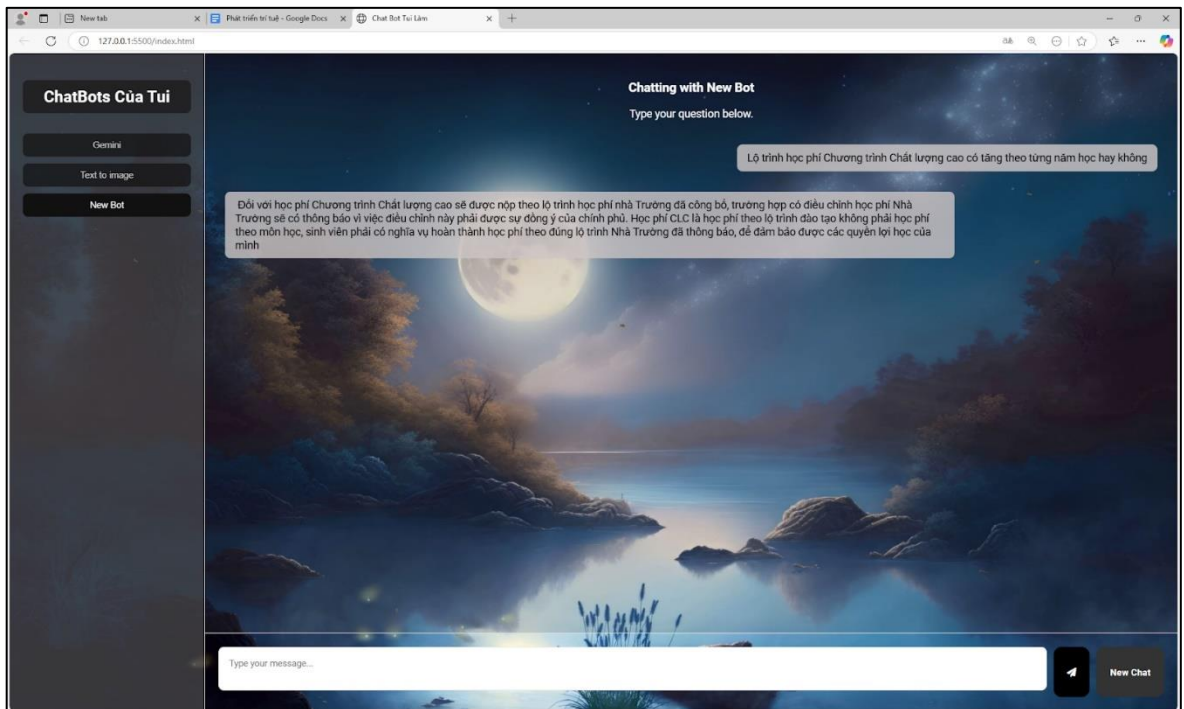


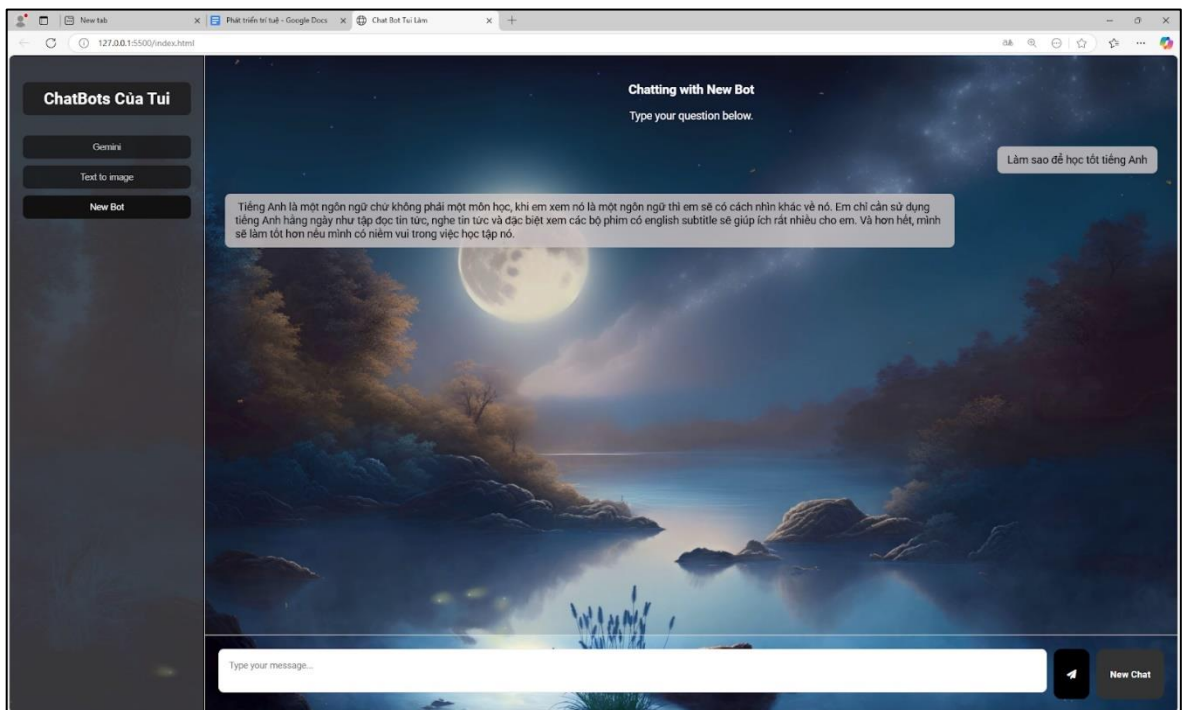
Figure 3.6.1. Chatbot test with question 1

Model Results with Inputs :Lộ trình học phí Chương trình Chất lượng cao có tăng theo từng năm học hay không?



*Figure 3.6.2. Chatbot test with question 2*

Model Results with Inputs :Làm sao để học tốt tiếng Anh?



*Figure 3.6.3. Chatbot test with question 3*

Model Results with Inputs :Nếu sinh viên vắng học Tiếng Anh (có lí do chính đáng) có thể xin học bù được không?

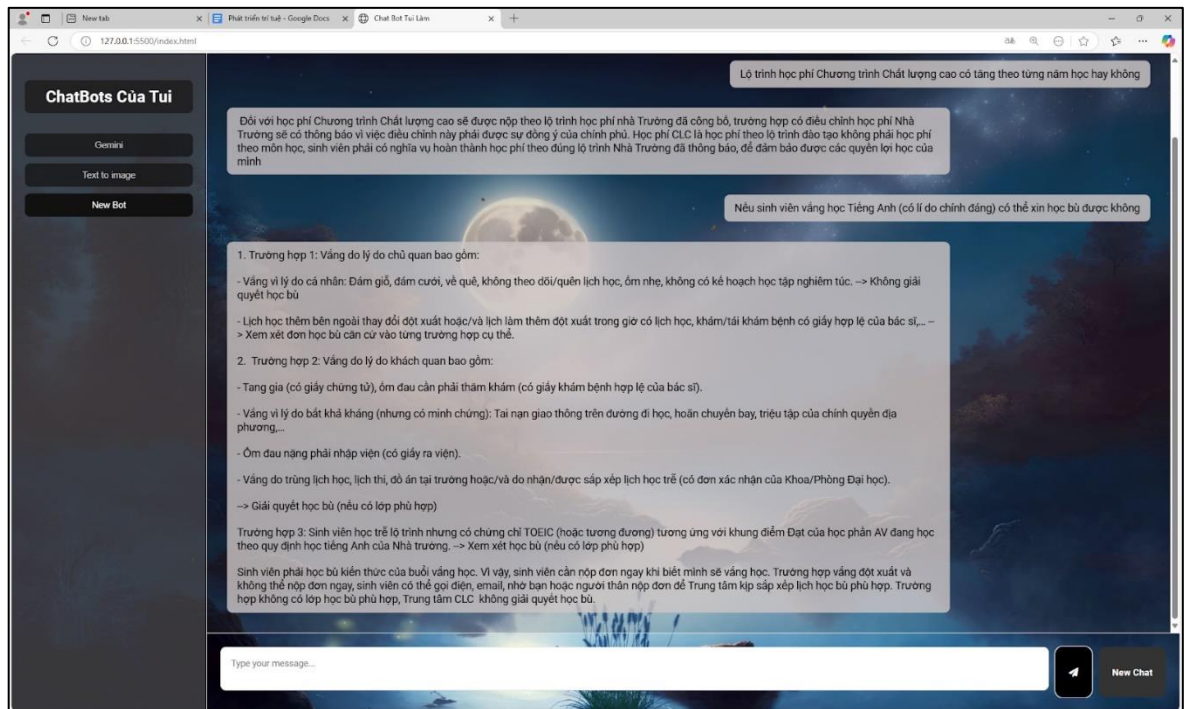


Figure 3.6.4. Chatbot test with question 3

## **CHAPTER 4 . CONCLUSION**

### **4.1 CONCLUSION**

The project has completed the construction of a web platform integrating 3 chatbots, including:

- Gemini: An information support chatbot from Google.
- Stable-diffusion-3.5-large from Hugging Face: Chatbot with strong image creation capabilities.
- Self-trained chatbot: An "academic support" chatbot trained with an available dataset.

Our website has achieved some positive results, including:

- Successfully integrating chatbots on the same interface.
- Ensuring good interaction and stable responses from chatbots.
- Initial evaluation shows that the system meets the basic needs of users.

However, the product still has some limitations:

- Limited scalability: The system has not been optimized to integrate more chatbots.
- Response speed: Sometimes the chatbot responds slowly because it has not been fully optimized.
- User experience: The interface and features as well as the current chatbot are still simple, unable to maximize their potential.

### **4.2 DEVELOPMENT**

Enhance scalability:

- Redesign the Web architecture to easily integrate more chatbots in the future.

Develop more self-trained chatbots:

- Learn, search for datasets and train more chat bots specializing in a fixed topic.
- Develop a more intuitive and friendly interface, supporting many interactions such as voice or images.

- Conduct UX/UI research to optimize the user experience.

Increase the integration of other AI features:

- Add available chatbots, specialized for fields such as education, searching for information online, or suggesting financial information

Deploy and test in real life:

- Collect feedback from many users to adjust and improve the website.



## REFERENCES

- [1] *What is Python? Executive Summary.* (n.d.). Python.org.  
<https://www.python.org/doc/essays/blurb/>
- [2] *W3Schools.com.* (n.d.). [https://www.w3schools.com/html/html\\_intro.asp](https://www.w3schools.com/html/html_intro.asp)
- [3] GeeksforGeeks. (2024, December 18). *Introduction to JavaScript.* GeeksforGeeks. <https://www.geeksforgeeks.org/introduction-to-javascript/>
- [4] Google Colab. (n.d.). *colab.google.* colab.google.  
<https://colab.google/#:~:text=Google%20Colaboratory,Blog>
- [5] Anandmeg. (2024, June 19). *What is the Visual Studio IDE?* Microsoft Learn.  
<https://learn.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2022>
- [6] *Hugging Face Hub documentation.* (n.d.).  
<https://huggingface.co/docs/hub/index>
- [7] Pichai, S. (2024, September 27). Introducing Gemini: our largest and most capable AI model. *Google.* <https://blog.google/technology/ai/google-gemini-ai/#introducing-gemini>
- [8] *RoBERTa.* (n.d.). [https://huggingface.co/docs/transformers/model\\_doc/roberta](https://huggingface.co/docs/transformers/model_doc/roberta)