IT Talent Training Course

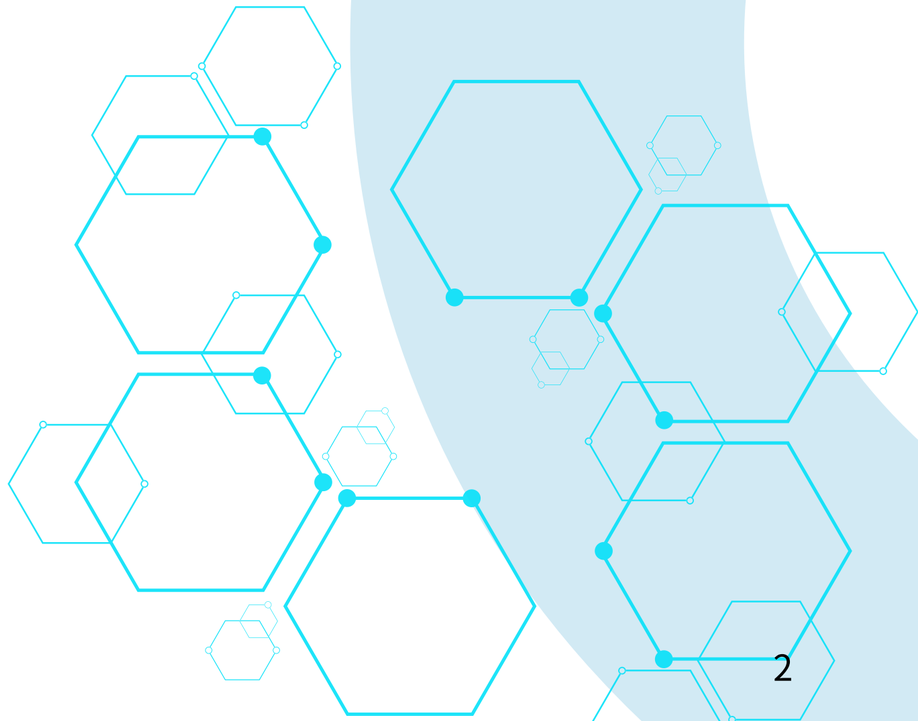
Aug. 2025.

A.I. PROGRAMMING WITH PYTORCH

Instructor :
Daesung Kim

8th Day - Part 01

INDEX



COURSE OBJECTIVES

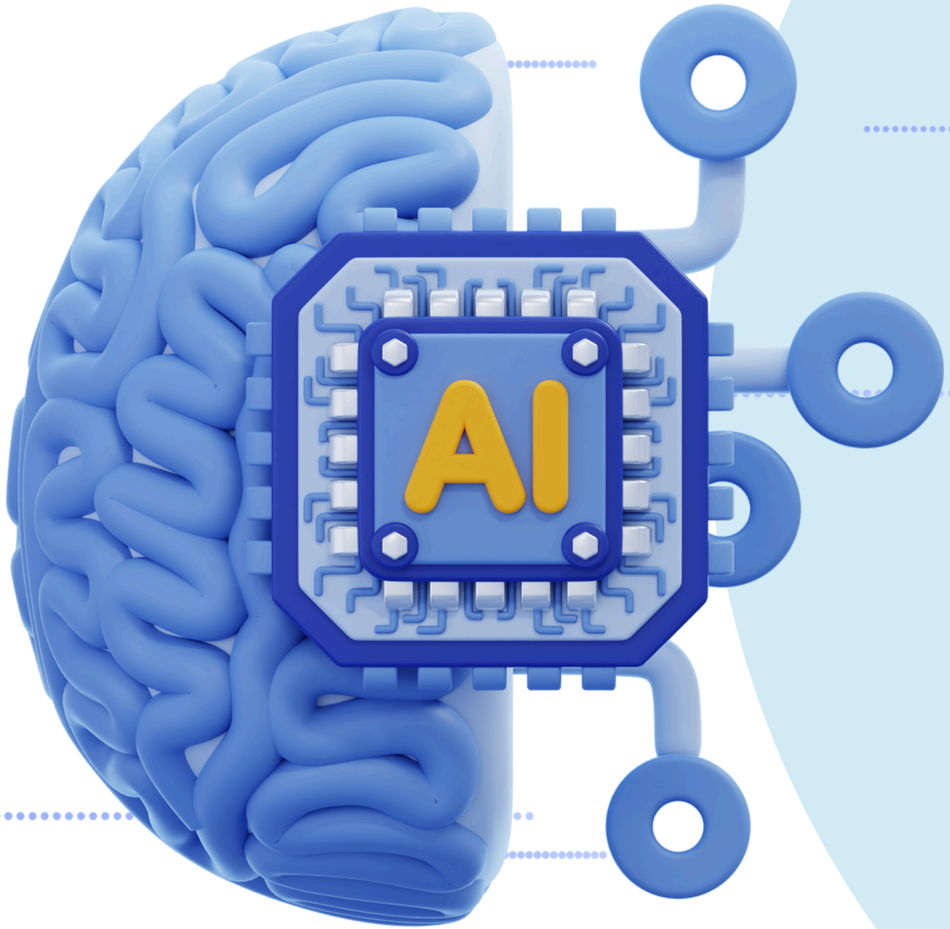
The goal of this morning session is to learn how to turn AI models or data analysis results developed in the Colab environment into **interactive web applications** that end users can use directly. To do this, you will master the core functions of the Streamlit library, which allows you to create web UIs surprisingly easily using only Python.

- You will understand the necessity of Streamlit and its development philosophy.
- You will learn how to run the Streamlit app and interact with users using basic widgets (buttons, text inputs, etc.).
- You will be able to systematically organize the layout of your app using sidebars and columns.
- You will understand the principles of **state management (st.session_state)**, the most important concept of Streamlit, and you will be able to use it to lay the foundation for interactive applications.

[Required background knowledge]

- **Basic Python grammar:** Basic knowledge of Python such as variables, functions, conditional statements, lists, dictionaries, etc.
- **Basic usage of terminal (command prompt):** Experience installing packages and executing scripts using the pip command
- **AI modeling experience:** Experience processing data or creating models in a Colab environment (boot camp content to date)

01
STREAMLIT OVERVIEW

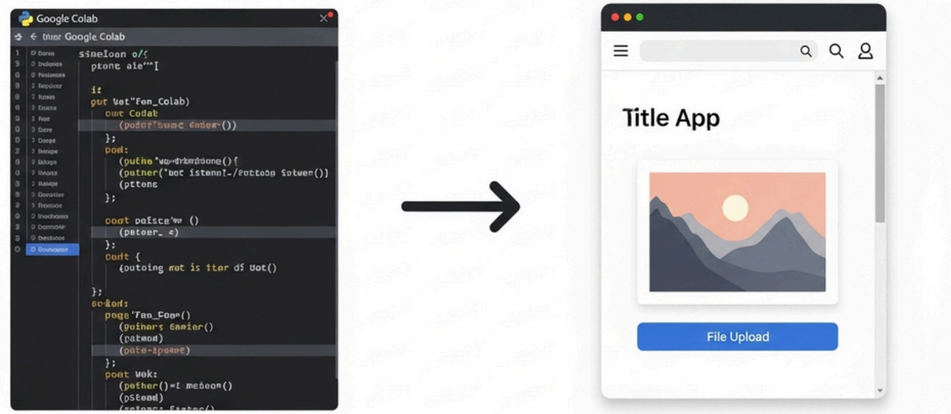


WHY DO DATA SCIENTISTS NEED A WEB FRAMEWORK? - 1

1. Transitioning from Colab to Service: From Ideas to Reality

So far, we have done various tasks such as data analysis, model training, etc. in a great environment called Google Colab. Colab is a great playground for developers, but it is not suitable for showing off our cool creations to friends, family, or future customers who are not developers. They cannot run complex code.

In order for **our AI models to have real 'value', they need to be in the form of a service** that end users can easily access and interact with. For example, if we have created an image classification model, we need a web page where users can upload images and check the results right away.



For Developers

For Users

WHY DO DATA SCIENTISTS NEED A WEB FRAMEWORK? - 2

2. Streamlit's philosophy: Reduce complexity and focus on the core

Traditionally, building web applications required learning various web technologies such as HTML (structure), CSS (design), and JavaScript (behavior). This was a huge burden for AI engineers and data scientists.

Streamlit boldly eliminates this complexity. **The core philosophy** of Streamlit is **"From Python scripts to data apps."** You can create data visualizations, model demos, and interactive tools amazingly quickly with just the Python code you already know.

- **Fast development speed:** You can create a working web app with just a few lines of code.
- **Easy to use:** Use commands like st.button() and st.write() as if you were calling a Python function.
- **Automatic updates:** When you modify and save the code, the web app automatically refreshes to show the changes immediately.

PREPARING THE STREAMLIT LAB ENVIRONMENT - 1

Now let's use Streamlit directly. Since the execution method is a little different in Colab, I recommend doing it in your **local computer environment (such as VS Code)**.

1. Install Streamlit library

First, open Terminal or Command Prompt (CMD) and enter the following command to install Streamlit.

```
# Practice code: Installing Streamlit
pip install streamlit
```

2. Create and run your first Streamlit app

1. Create a Python file named my_first_app.py in a folder of your choice.
2. Write the following code in the file and save it.

PREPARING THE STREAMLIT LAB ENVIRONMENT - 2

```
# Practice code: my_first_app.py

import streamlit as st
import time

# st.title() : Function to write the largest title
st.title("My First Streamlit App in LAOS!")

# st.header() : Function to write subtitle
st.header("Day 8 AM: Streamlit Basics")

# st.write() : Magically output various things such as text, numbers, and data frames
st.write("Hello, world! This app was made only with Python.")

# Example of using button widget
if st.button("Say hello"):
# When the button is clicked, the code below is executed.
st.write("Why hello there! Nice to meet you!")
else:
# When the button is not clicked
st.write("Goodbye")
```

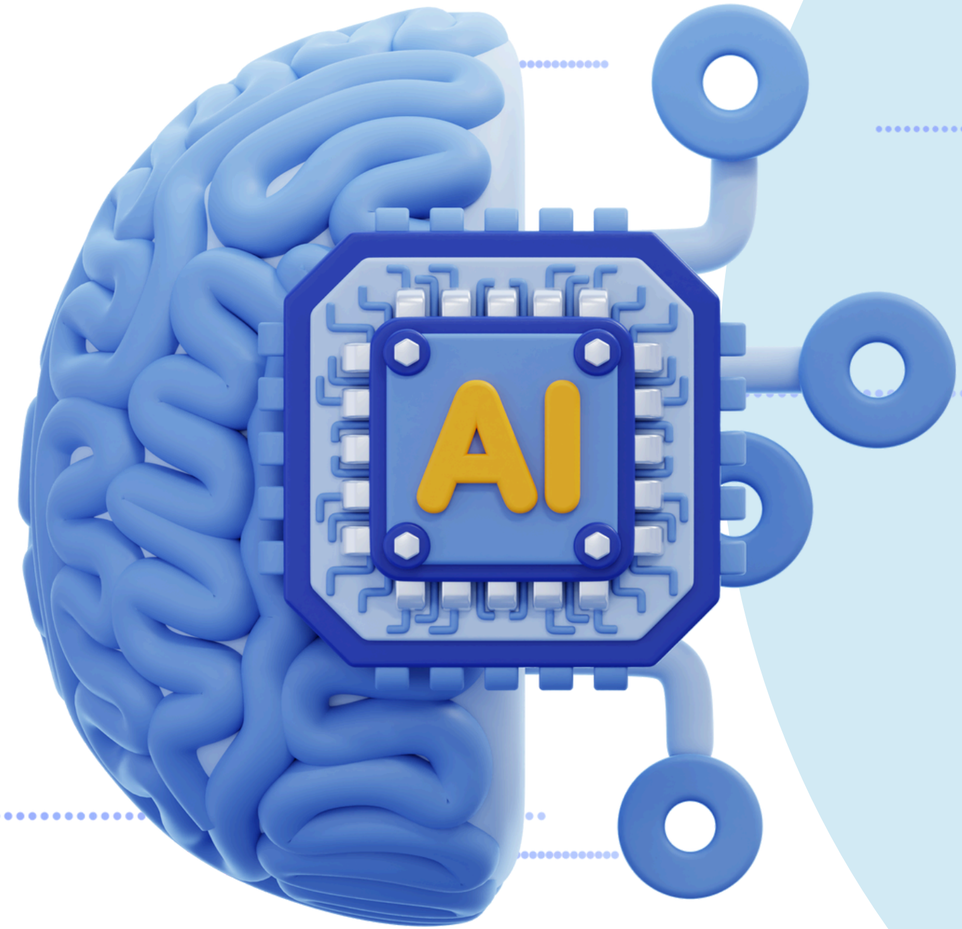
PREPARING THE STREAMLIT LAB ENVIRONMENT - 2

3. In Terminal, navigate to the folder containing the file you just created and run the following command:

```
# Practice Code: Running the Streamlit App
streamlit run my_first_app.py
```

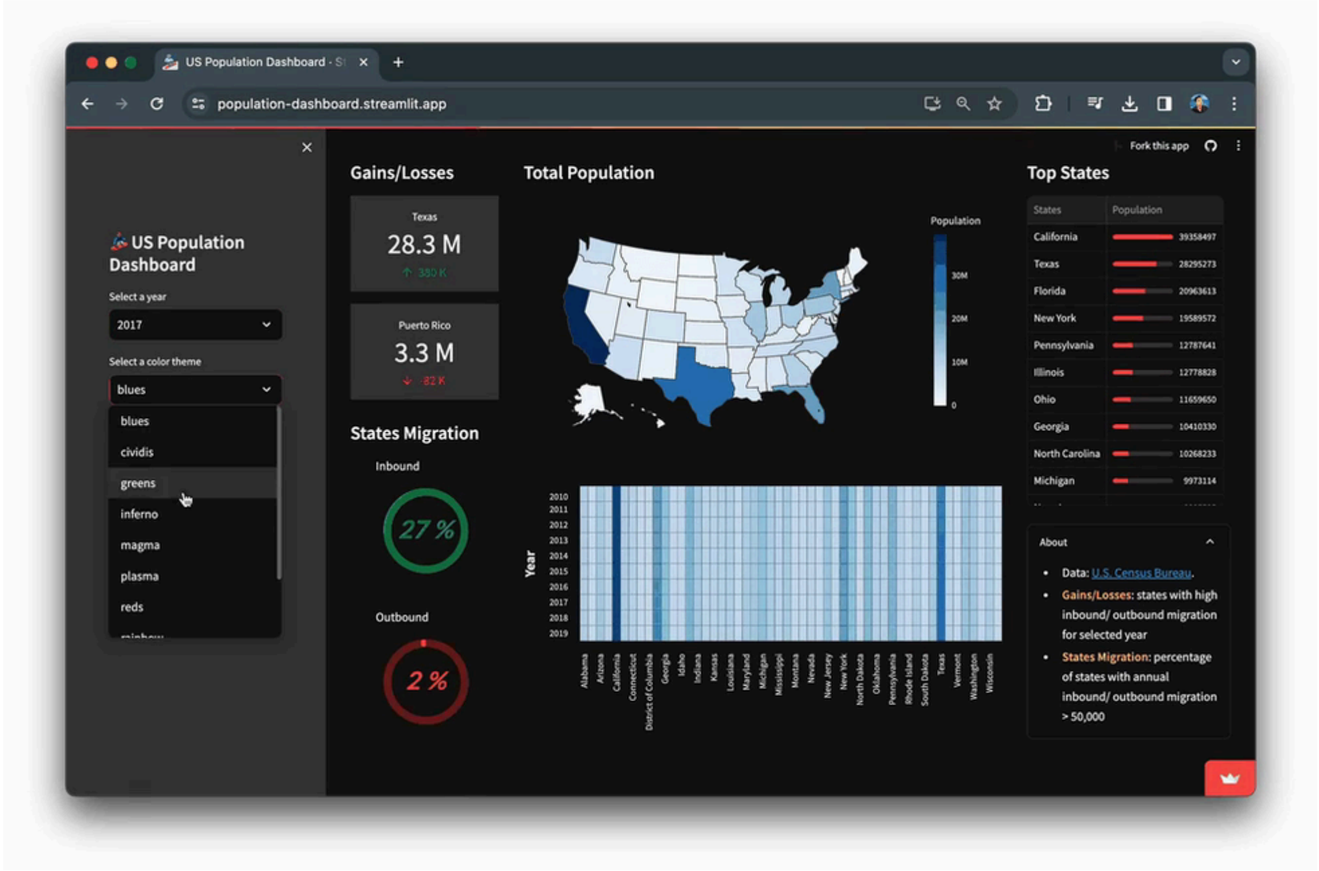
When you run the command, a new tab will open in your web browser, showing your first web application! Try clicking the button to see how it works.

02
USING CORE WIDGETS



WHAT IS STREAMLIT WIDGETS?

A widget is any UI element that the user interacts with. Streamlit provides very intuitive widgets.



USING WIDGETS - 1

1. Text and data display widgets

This is the most basic way to display information in your app.

```
# Practice code: Text-related widgets
import streamlit as st

# Title (largest text)
st.title("Text Widget Demo")

# Header (subheading)
st.header("This is a Header.")

# Subheader (smaller subheading)
st.subheader("This is a Subheader.")

# Versatile output for plain text, markdown, dataframes, etc.
st.write("Plain text: Streamlit is really easy and fun.")
st.write("---") # Horizontal lines (markdown)
st.write("## Markdown headers are also supported!")
st.write("* Bullet 1\n* Bullet 2")

# Display code block
code = """
def hello():
print("Hello, Streamlit!")
"""
st.code(code, language='python')
```

USING WIDGETS - 2

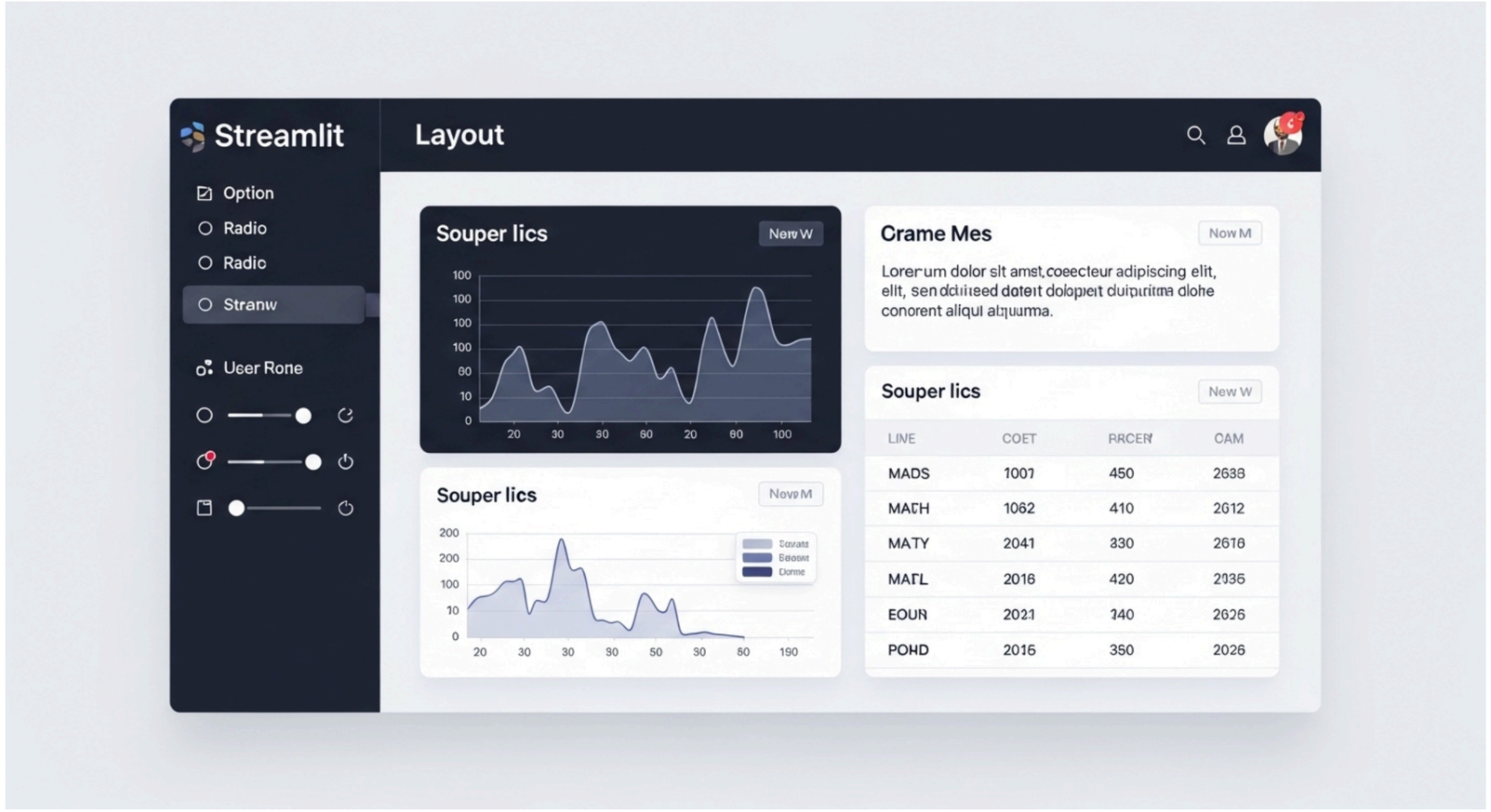
2. User interaction widgets

Widgets that take input from the user or trigger certain actions.

[Code] Day08 - 01_User_Interactive_widget.py

APP LAYOUT - 1

As your information and widgets increase, you will need to organize your screens systematically. Streamlit makes this easy by providing sidebar and column features.



APP LAYOUT - 2

1. st.sidebar: Create a clean side menu

If you add st.sidebar. before any widget that starts with st., that widget will appear in the left sidebar. It is mainly used for selecting options or organizing menus.

```
# Practice code: Using the sidebar
import streamlit as st

# Displaying a title on the main screen
st.title("Using the sidebar and columns")

# Adding widgets to the sidebar
# Using the with st.sidebar: syntax allows you to manage the code more cleanly.
with st.sidebar:
    st.header("Sidebar menu")
    st.write("Select an option here.")
    option = st.selectbox(
        'Which chart would you like to see?',
         ('Line chart', 'Bar chart')
    )
    st.write("Selected option:", option)

st.write(f"The main screen will show '{option}'.")
# (The code for drawing the actual chart is omitted here because it requires data.)
```

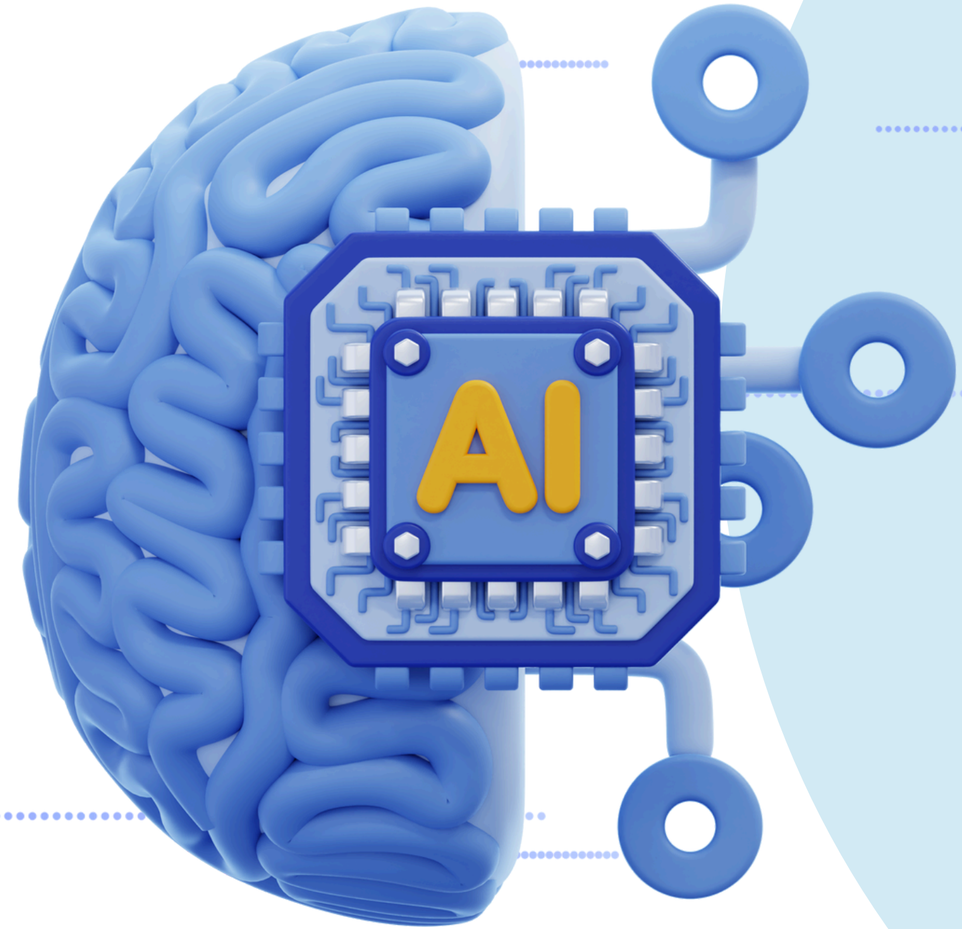
APP LAYOUT - 3

2. st.columns: Split the screen into multiple columns

Using st.columns() you can divide the screen into any number of vertical columns.

[Code] Day08 - 02_Split_screen.py

03
STATE MANAGEMENT



THE CORE OF STREAMLIT: STATE MANAGEMENT

This concept is the most important part you need to understand to properly utilize Streamlit, especially the core principle of the conversational chatbot we will be building later this afternoon.

1. Understanding Streamlit's Re-run Mechanism

The Streamlit app re-runs the entire script from beginning to end every time the user interacts with the widget (e.g. clicking a button, entering text).

This is what makes Streamlit simple, but it introduces a problem: every time the script is re-run, all previous variable values are lost.

Bad example: Creating a counter with a regular variable

```
# Practice code: Bad counter example (not working)
import streamlit as st

st.title("The importance of state management")

# Counter is always initialized to 0 every time the script is run
counter = 0

if st.button("counter +1"):
    counter = counter + 1

st.write(f"Current count: {counter}") # Always displays 0 or 1
```

Run the code and keep pressing the button. The number will never increase and will always show 1. This is because the counter = 0 code is executed again every time you press the button.

THE CORE OF STREAMLIT: STATE MANAGEMENT

2. st.session_state: The app's memory manager.

To solve this problem, Streamlit provides a special 'memory space' called st.session_state. st.session_state works like a dictionary, and the values stored in it are preserved even when the script is re-executed.

Correct example: Creating a counter with st.session_state

```
# Practice code: Correct counter using st.session_state
import streamlit as st

st.title("Correct state management: Session State")

# 1. Check if session_state has 'counter' key, and if not, initialize
if 'counter' not in st.session_state:
st.session_state.counter = 0

# 2. Change the value of session_state when the button is clicked
if st.button("Counter +1"):
st.session_state.counter += 1

# 3. Display the value of session_state on the screen
st.write(f"Current count: {st.session_state.counter}")

# Button to initialize session_state
if st.button("Initialize counter"):
st.session_state.counter = 0
```

Now, if you press the "Counter +1" button, you can see the number increasing normally. In the chatbot we will create in the afternoon, we will use this st.session_state to continuously save the conversation history between the user and the bot.

SUMMARY

Summary

- AI models and data analysis results are truly valuable when they are presented in the form of web applications that end users can easily use.
- Streamlit is a powerful framework that allows you to quickly and easily create interactive web apps using only Python code without complex web technologies.
- You can display information and interact with users using various widgets such as st.title, st.button, st.text_input, st.file_uploader.
- You can systematically organize the layout of your app using st.sidebar and st.columns.
- Since Streamlit re-runs the entire script when interacting, you must manage the state using st.session_state to maintain the values.

Glossary of terms:

- Streamlit: An open source framework for building data science and machine learning web apps in Python.
- Widget: Any UI component that interacts with the user on a web page, such as a button, text box, or slider.
- Layout: A way to visually organize and arrange the content of a web page (e.g. sidebar, column).
- State Management: A technique for maintaining and managing data (state) consistently across application re-runs or across multiple components.
- Session State: A st.session_state object. A memory space that stores and maintains data for the duration of a user's session while the app is open.
- Re-run: A mechanism for re-running an entire Python script from scratch whenever a user interacts with a Streamlit app.

IT Talent Training Course

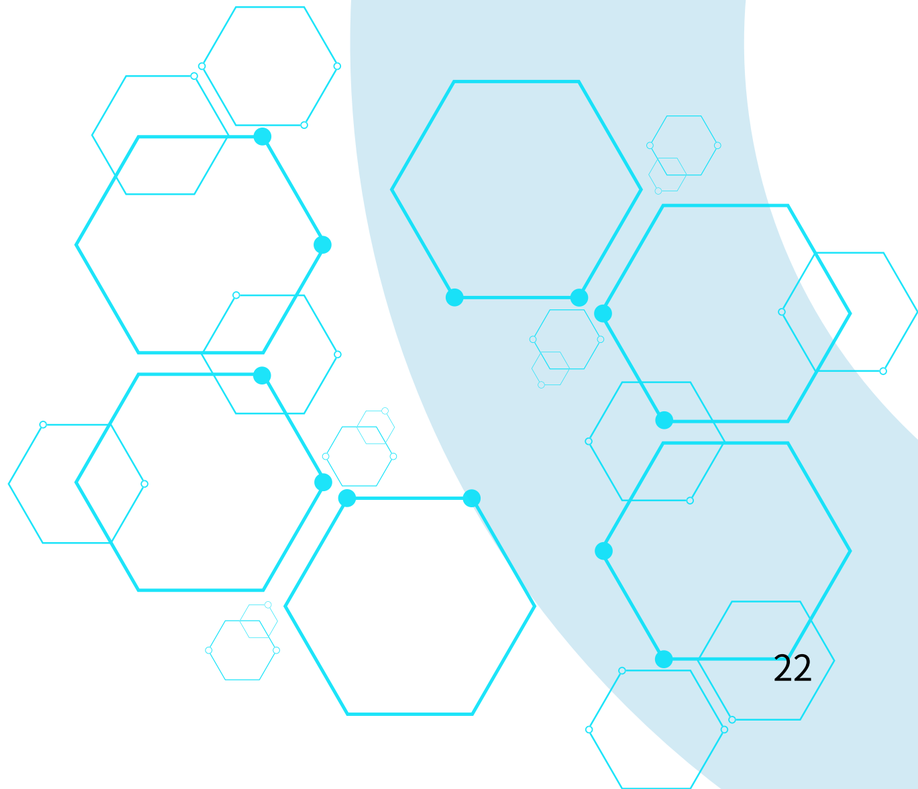
Aug. 2025.

A.I. PROGRAMMING WITH PYTORCH

Instructor :
Daesung Kim

8th Day - Part 02

INDEX

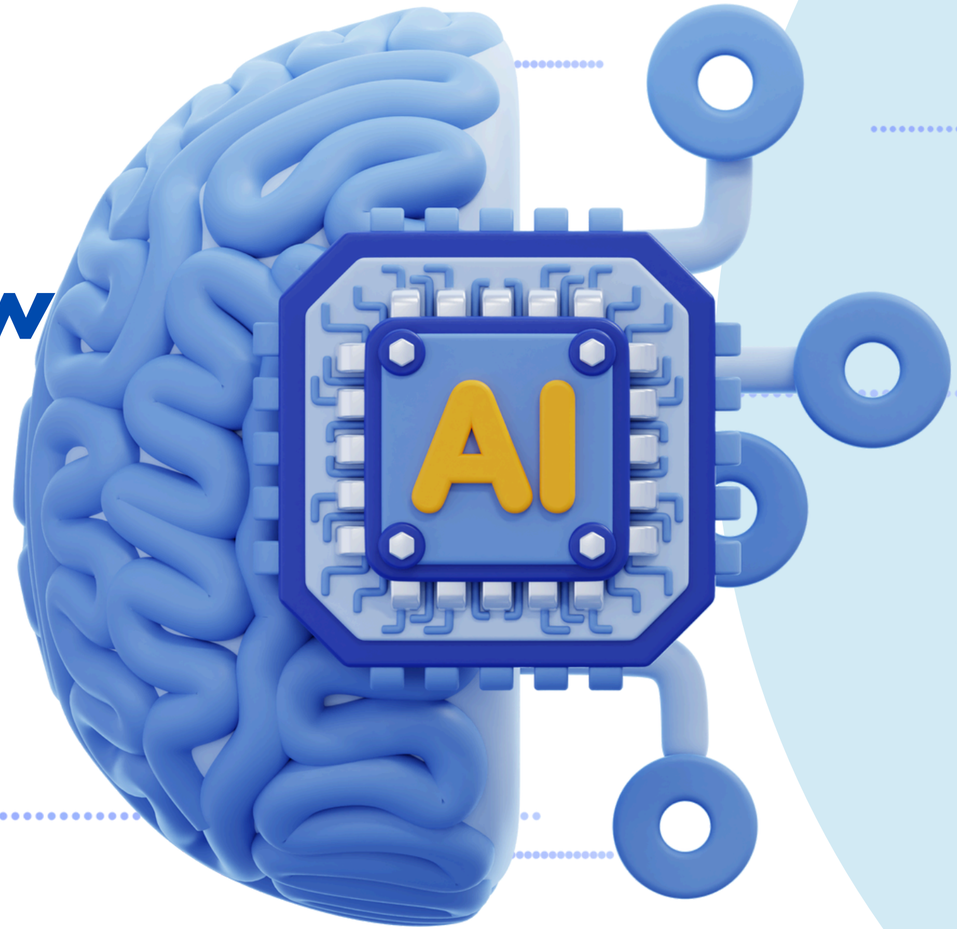


COURSE OBJECTIVES

The goal of this session is to integrate the Streamlit widget usage learned in the morning and the Gemini API usage learned on Day 7 to complete a **conversational AI chatbot application** that interacts with real users from start to finish. Through this process, you will cultivate engineering skills to build a 'living' AI service that responds dynamically to user input, beyond simply creating a model.

- You can build a chatbot interface using Streamlit's conversational widget.
- You can maintain and manage the user's previous conversation history using st.session_state.
- You can implement logic to call the Gemini API based on user input and display the response on the screen in real time.
- Understand the perspective of a service developer who considers the user experience (UX) beyond technical implementation.

01
Chatbot Project Overview



OVERVIEW - 1

1. Linking theory and service

So far, we have been building and testing AI models in a Colab notebook environment. This is a great environment for research or personal learning, but you need a separate ‘window’ to let your end users use your AI models. A chatbot application is one of the most intuitive and effective ways to connect the AI technology we have built (Gemini API) to an easy-to-experience end user experience.

Today’s lab will be your first experience integrating the individual technologies you have learned so far into a single ‘product’. This process is critical to building your ability to connect technical achievements to real-world value.

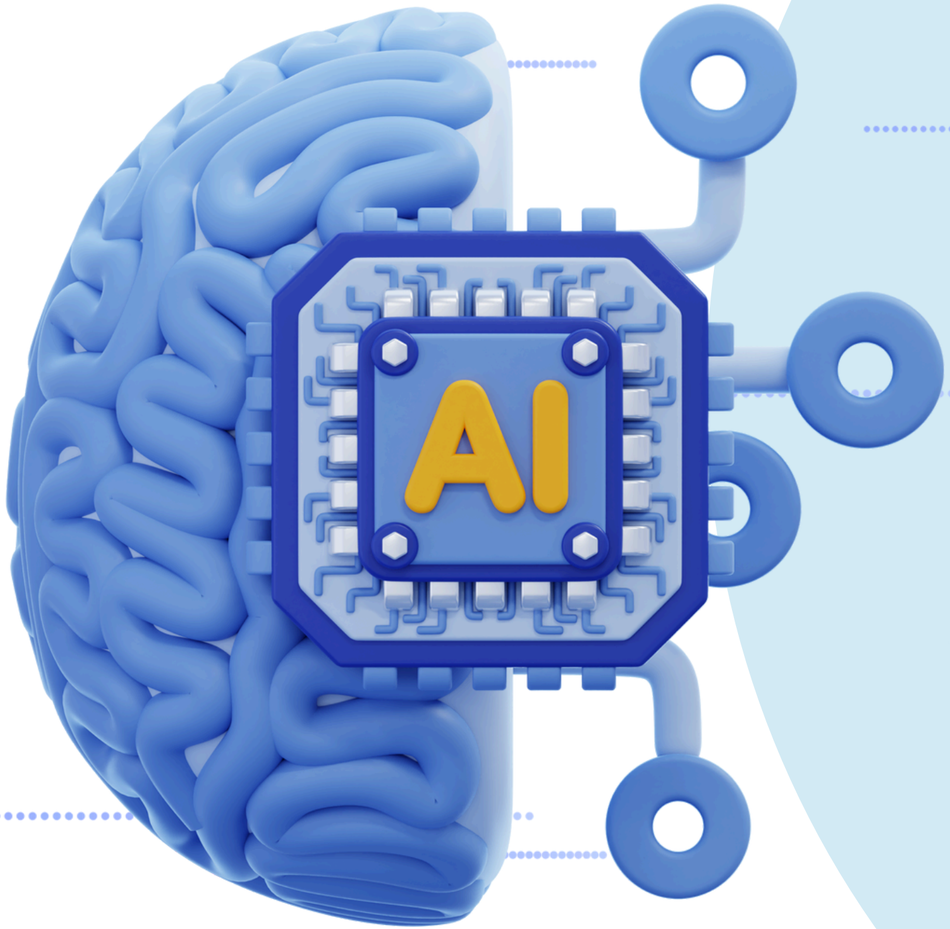
OVERVIEW - 2

2. Understand the project architecture

The chatbot we will create will work in the following flow:

1. **User:** Enter a question in the input field displayed in the web browser and press the 'Submit' button.
2. **Streamlit UI:** Receives user input and passes it to the Python script. At the same time, it displays the user's question beautifully on the screen along with the previous conversation.
3. **State management (st.session_state):** It 'remembers' both the user's question and the previous conversations. This entire conversation history is important information for understanding the context.
4. **Gemini API:** Streamlit passes the entire conversation history it remembered to the Gemini API and asks, "Generate an appropriate answer for the next conversation."
5. **Streamlit UI (Update):** It saves the answer received from Gemini API back to st.session_state and displays it on the screen as the chatbot's response.

02
Chatbot implementation



STEP 1: CONFIGURE THE CHATBOT UI

st.chat_message, st.chat_input

Streamlit provides widgets that allow you to create simple chat UIs.

- **st.chat_message(name):** This function creates a chat bubble. The name argument can be assigned a role such as 'user' or 'assistant' to display a different icon. This function is used with the with statement to display the content written in it in the bubble.
- **st.chat_input(placeholder):** This function creates a text input field fixed to the bottom of the screen. When the user types a message and presses enter, the message content is returned. It returns None until the user types something.

```
import streamlit as st

st.title("Simple Chat UI Example")

# Create a speech bubble with the role 'user'
with st.chat_message("user"):
    st.write("Hello!")

# Create a speech bubble with the role 'assistant'
with st.chat_message("assistant"):
    st.write("Hello! How may I help you?")

# Create an input field to receive user input
if prompt := st.chat_input("Enter your message..."):
    # Display the message entered by the user in the 'user' speech bubble
    with st.chat_message("user"):
        st.markdown(prompt)
```

STEP 2: MANAGE YOUR CONVERSATION HISTORY - 1

st.session_state

st.session_state is a special dictionary object that can store data while the user interacts with the app (while the session is maintained). The values stored in st.session_state will not be lost even if the script is re-run. We will use this property to store the chatbot's conversation history.

Conversation History Management Logic:

1. **Initialization:** When the app is first run, check if there is a list in st.session_state to store the conversation history. If not, create an empty list with the key messages.
2. **Display History:** Whenever the script is run (when the user inputs or refreshes the page), it iterates through the st.session_state.messages list from the beginning to the end and redraws all the conversations on the screen.
3. **Add History:** When the user inputs a new message, it adds the message to the st.session_state.messages list.

STEP 2: MANAGE YOUR CONVERSATION HISTORY - 2

```
import streamlit as st

st.title("Chatbot that remembers conversation history")

# 1. Initialization: If st.session_state does not have 'messages', create an empty list
if "messages" not in st.session_state:
    st.session_state.messages = []

# 2. Show history: Show all previous conversations
for message in st.session_state.messages:
    with st.chat_message(message["role"]):
        st.markdown(message["content"])

# Get user input
if prompt := st.chat_input("Enter a message..."):
    # 3. Add history: Add the message the user entered to the session_state
    st.session_state.messages.append({"role": "user", "content": prompt})
    # Show the added content immediately on the screen
    with st.chat_message("user"):
        st.markdown(prompt)

# (Here we call the Gemini API to get a response, and we also need to add that response to the log)
```

STEP 3: CONNECT TO GEMINI API

Now it's time to implement the logic to receive user input, pass it to Gemini API, and receive the response and display it on the screen.

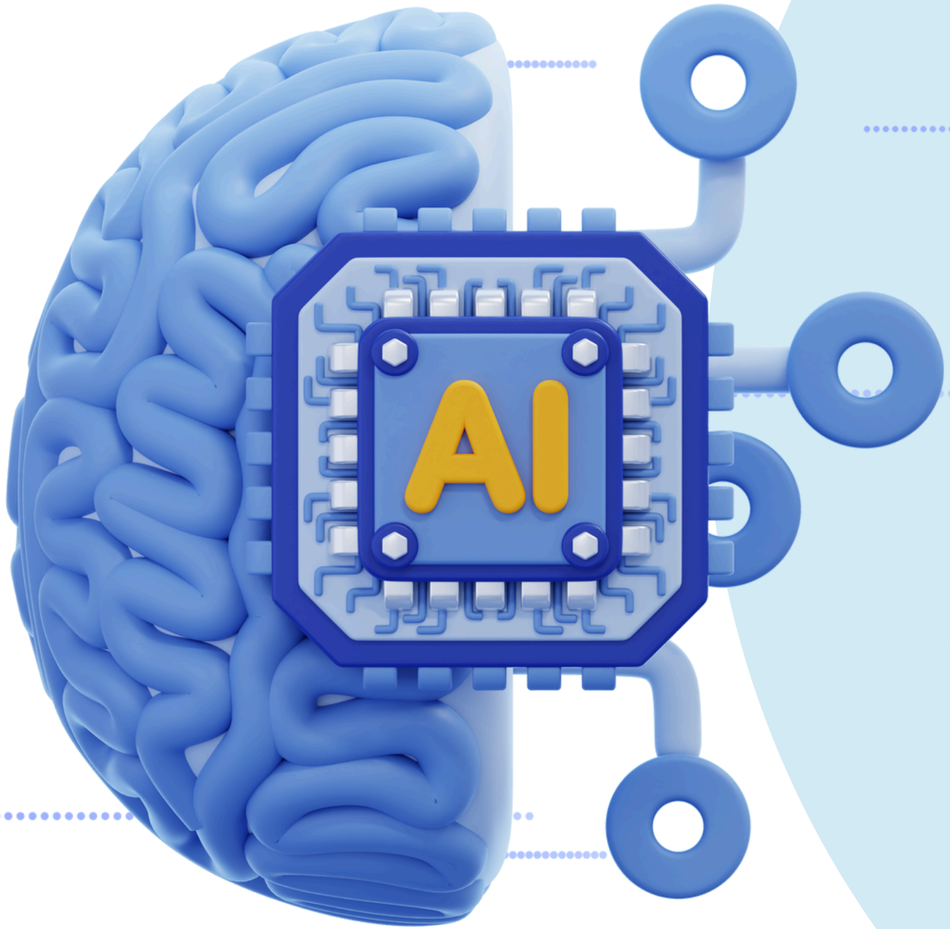
API integration logic:

1. User inputs a message and sends it (st.chat_input).
2. Add the user's message to st.session_state.messages and display it immediately on the screen.
3. Convert the entire conversation history stored in st.session_state.messages into a format that Gemini API can understand. (Gemini prefers to receive the previous history together to understand the context of the conversation.)
4. Call Gemini API (model.generate_content) to request a response.
5. Add the response received from the API (chatbot's answer) to st.session_state.messages and display it on the screen.

With this logic implemented, the chatbot will not simply respond with a short answer like "Hello" -> "Hello", but will understand the flow of the previous conversation and generate an answer that fits the context.

03

Create your own AI chatbot



IMPLEMENTATION - 1

1. Pre-preparation: Setting up the library and API key

Install the required libraries from the terminal.

```
pip install streamlit google-generativeai
```

You need to prepare a Gemini API key. (Refer to the 7th day practice)

In a real deployment environment, it is safe to use st.secrets, but for convenience in this practice, we will use the method of directly entering it in the code or using an environment variable.

IMPLEMENTATION - 2

2. Full code and explanation

[Code] Day08 - 03_Chatbot.py

1. **Initial setup:** Set the page title and icon, and set the Gemini API key. In a real service, it is standard practice to use st.secrets for security.
2. **Initialize model and session:** Load the model with genai.GenerativeModel('gemini-pro'). Start a chat session by calling model.start_chat(history=[]) and save this session object to st.session_state.chat so that the context of the chat is maintained even when the script is re-executed.
3. **Render UI:** Loop through the previous chat history stored in st.session_state.chat.history and draw everything on the screen using st.chat_message. The history object in the Gemini library has role (user or model) and parts (content) as properties.
4. **Handle user input:** Get user input with st.chat_input.
5. **API call and response handling:** Call the Gemini API by passing the user's message (prompt) to the st.session_state.chat.send_message() function. The response is stored in the response variable.
6. **Streaming response:** If you use the stream=True option, Gemini's response will be sent in real time, word by word, rather than all at once. If you receive each piece through the for chunk in response: syntax and concatenate them, you can improve the user experience by giving a typing-like effect. st.empty() is used to create an empty space and continuously update the contents of that space.

IMPLEMENTATION - 3

3. Improving code from a user experience (UX) perspective

Beyond simply implementing functions, it is the virtue of a good engineer to think about how to make the service more convenient and enjoyable for users.

1. **Add an initial message:** You can add an initial message so that the chatbot talks to the user first. When initializing st.session_state.chat, try adding a system message.
2. **Loading spinner:** You can use st.spinner to display a message like "Thinking..." so that the user doesn't get bored while waiting for the API response.
3. **Provide an input example:** You can use the placeholder of st.chat_input to show examples of what questions the user can ask. (Example: "Tell me what PyTorch is")
4. **Reset conversation button:** You can create a "Start a new conversation" button in st.sidebar and add a function to initialize st.session_state.chat again when the button is pressed.

SUMMARY

Summary

- We built a conversational UI using Streamlit's st.chat_message and st.chat_input widgets.
- To overcome the limitations of Streamlit (re-running scripts on interaction), we learned how to manage the conversation history persistently using st.session_state.
- We implemented a complete chatbot application logic that passes user input and previous conversation history (context) to Gemini API, receives responses streamed in real time, and displays them on the screen.
- Through this project, we gained practical experience in integrating multiple technologies to create a real-world working AI service, and understood the importance of user experience.

Glossary of terms:

- st.session_state: A built-in object of Streamlit to store and share data while the user's browser session is maintained. It is essential for state management because its value is maintained even when the script is re-executed.
- st.chat_input: A widget that is fixed at the bottom of the screen and receives text input from the user.
- st.chat_message: A container that displays messages from the user or assistant in a style (speech bubble) that is distinguished by role.
- State Management: A technology that maintains data consistency when the application is re-executed or multiple components interact. It is a very important concept in web applications, especially Streamlit.
- User Experience (UX): The overall experience that a user feels and thinks while using a system, product, or service. It aims to increase user convenience and satisfaction beyond the implementation of functions.