

# Streamlit으로 웹 앱 구현하기

## ▼ 참고 강의

1. Streamlit과 LLM 모델을 사용해 서비스를 만드는 방법
  - <https://www.youtube.com/watch?v=ZVmLe3odQvc>
2. Streamlit과 RAG를 활용해 챗봇 서비스를 만들기
  - Youtube 다비드 스튜디오 | <https://youtu.be/s6rX8LLgTFQ?si=TVxY2nJAdUfZhogZ>
  - Youtube 다비드 스튜디오 | [https://youtu.be/YzIGaGfARpl?si=U\\_1HmfPVw4WWxL71](https://youtu.be/YzIGaGfARpl?si=U_1HmfPVw4WWxL71)

## Streamlit

|  <https://streamlit.io/>



[Cloud](#) [Gallery](#) [Components](#) [Generative AI](#) [Community](#) [Docs](#) [Blog](#)

[Sign in](#)

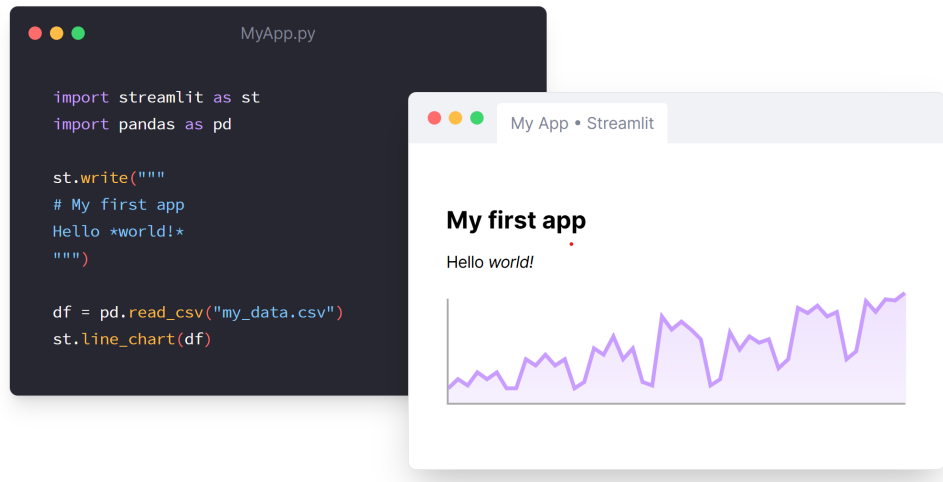
[Sign up](#)

## A faster way to build and share data apps

Streamlit turns data scripts into shareable web apps in minutes.  
All in pure Python. No front-end experience required.

[Try Streamlit now](#)

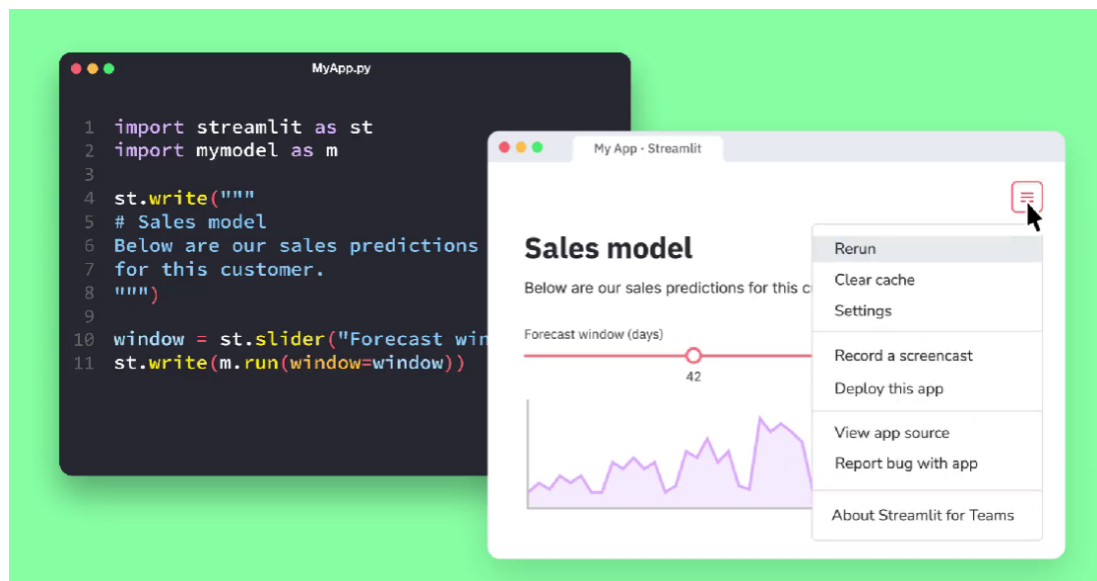
[Deploy on Community Cloud \(it's free!\)](#)



Streamlit 홈페이지 화면 캡처

## Streamlit이란?

- **Python**으로 빠르고 간단하게 웹 애플리케이션을 만들 수 있는 오픈소스 프레임워크
- **복잡한 프론트엔드 개발 지식 없이** Python 코드만으로 **인터랙티브 웹앱** 제작 가능
  - 예) 사용자와 상호 작용할 수 있는 웹앱 제작



슬라이더, 버튼, 체크박스 등을 사용해 데이터세트 실시간 조작 가능한 스트림릿으로 만든 웹 앱

- 주로 데이터 사이언티스트, AI 엔지니어, 데이터 분석가가 AI 모델의 결과를 실시간으로 시각화하거나, 대규모 데이터 세트를 분석하여 인사이트를 추출하는 데 사용된다.
  - 이들은 주로 Python을 사용 → Streamlit은 Python으로 개발 가능하기 때문
  - 많은 경우, 프론트엔드 개발에 익숙치 않음 → 프론트엔드 개발 지식 없이도 서비스 배포 가능하기 때문

- 무료 오픈 소스 + 쉬운 사용 방법
  - 빠르게 프로토타입 서비스를 만들어 공유하는데 적절하기 때문

## Streamlit 활용 예시

- 인터랙티브 데이터 대시보드 만들기

```
##### 슬라이더에서 특정 값을 고르면, 그 제공값을 알려주는 웹 앱 ##
####
```

```
import streamlit as st

# 슬라이더 생성
x = st.slider("Select a value")

# 생성된 값 출력
st.write(x, "squared is", x*x)
```



- **pandas DataFrame으로 정리된 데이터 시각화**
  - 사용자가 지표별로 시각화 가능

```
import streamlit as st
import pandas as pd
```

```

import plotly.express as px

def model_performance_visualization():
    # 페이지 제목 설정
    st.title("📊 LLM 모델 성능 비교")

    # 샘플 모델 성능 데이터 ( DataFrame을 불러온다.)
    model_data = pd.DataFrame([
        {"모델": "GPT-3.5", "정확도": 0.85, "추론 속도": 0.7, "메모리 사용량": 0.6},
        {"모델": "GPT-4", "정확도": 0.92, "추론 속도": 0.5, "메모리 사용량": 0.8},
        {"모델": "Claude-3", "정확도": 0.89, "추론 속도": 0.6, "메모리 사용량": 0.7},
        {"모델": "Llama-2", "정확도": 0.82, "추론 속도": 0.8, "메모리 사용량": 0.5}
    ])

    # 성능 지표 선택
    performance_metric = st.selectbox(
        "비교할 성능 지표를 선택하세요",
        ("정확도", "추론 속도", "메모리 사용량")
    )

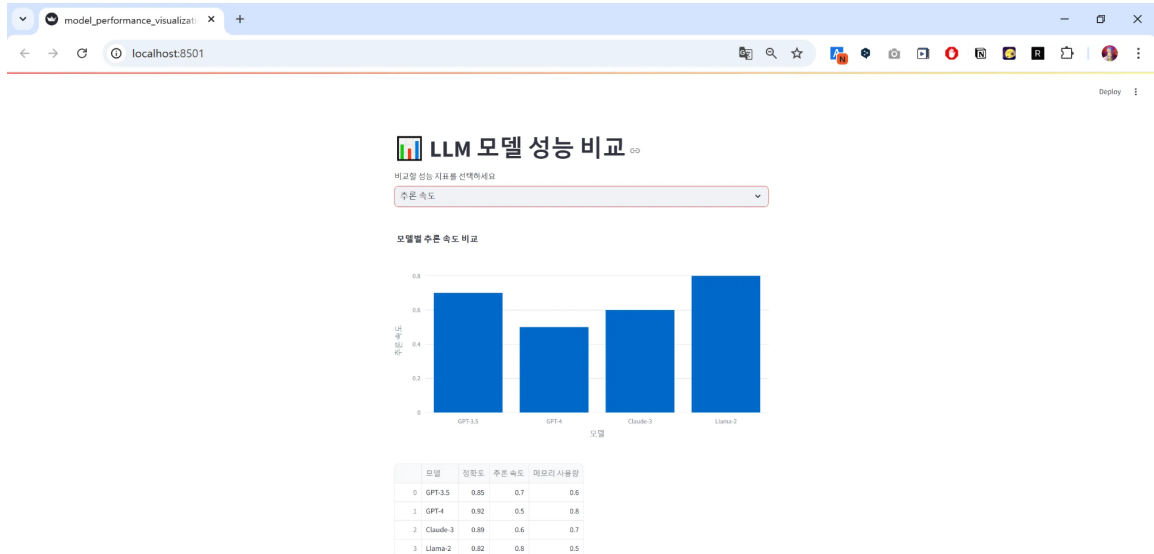
    # 바 차트 생성
    fig = px.bar(
        model_data,
        x="모델",
        y=performance_metric,
        title=f"모델별 {performance_metric} 비교"
    )

    # 차트 표시
    st.plotly_chart(fig)

    # 상세 데이터 테이블
    st.dataframe(model_data)

```

```
# Streamlit 앱 실행
if __name__ == "__main__":
    model_performance_visualization()
```



## • LLM 챗봇 애플리케이션 배포

- RAG를 사용해 LLM 모델의 한계를 보완하는 챗봇 구현 및 배포
- 웹에서 직접 docx 파일을 업로드
  - 사용자가 제공하는 외부지식과 결합해 그 지식 분야의 전문 챗봇 생성 가능

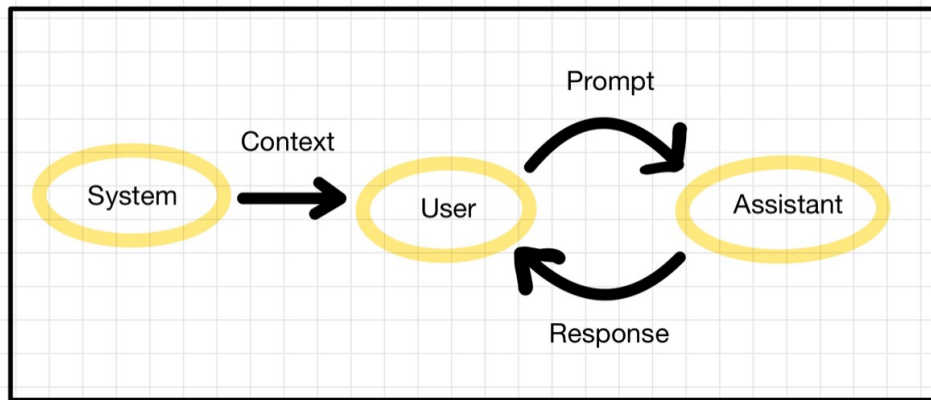
## Streamlit으로 RAG를 사용하는 LLM 어플리케이션 구축하기

### ▼ [참고] Streamlit 챗봇 작동 원리 | System, User, Assistant

#### ▼ 참고

출처: Youtube 다비드 스튜디오 |

<https://youtu.be/s6rX8LLgTFQ?si=TVxY2nJAdUfZhogZ>



- chat GPT와 대화할 때를 생각해봤을 때,
  - 챗봇 → Assistant
  - 사용자 → User
  - 챗봇이 어떤 챗봇인지 상황 설정을 하는 것 → System Prompt
    - 예) '넌 파이썬 전문 개발자야. 파이썬 전문가의 입장에서 해당 코드를 리뷰 해줘. 너는 항상 반말을 하는 챗봇이니까 항상 반말로 친근하게 대답해줘. 영어로 질문 받더라도 무조건 한글로 답변해줘. ...'
    - 보통 대화의 맨 앞에 들어가게 된다.
- 대화가 이어질수록 User, Assistant 간 대화가 계속 왔다 갔다 반복되며 `messages`에 쌓인다.

```
import streamlit as st
```

```
st.session_state.messages.append({"role": "user", "content": prompt})
```

```
st.session_state.messages.append({"role": "assistant", "content": response})
```

## ▼ [참고] Langchain 특징 | 추상화, 표준화, 체이닝

### ▼ 참고

[https://youtu.be/m8tIOcMcwno?si=ioFYR\\_Y7gkdIZfXN](https://youtu.be/m8tIOcMcwno?si=ioFYR_Y7gkdIZfXN)

- **01 추상화**


- 복잡한 작업을 간결하게 표현
- 예)
  - RAG 챗봇을 만들 때, PDF를 읽어와 텍스트를 불러오고 / 텍스트를 쪼개고 임베딩하고 / 벡터 DB에 저장하는 과정들은 하나하나 Python 코드로 짜면 길어지지만, Langchain을 활용하면 매우 간결하게 표현 가능

- **02 표준화**

- 코드를 재사용 가능하게 만든다.
- 비슷한 기능이 있으면, 새로 코드를 만들지 않고, 기존에 만들어 놓은 것을 똑같은 형식으로 재사용한다.
- 예)
  - 여러 회사의 모델을 사용하는 경우 → Langchain의 **CHATMODEL** 사용 가능
    - ChatAntropic
    - ChatOpenAI
    - ...
  - Langchain을 사용하면, 각 모델들의 사용법을 모두 각 사이트에 들어가 확인할 필요가 없다.
  - 여러 형식의 문서를 읽어오는 경우 (PDF, docx, drive, ...) → Langchain의 **Document** 사용 가능

- **03 체이닝**

- 컴포넌트를 쉽게 연결할 수 있다. Input을 넣으면 Output이 나오는 형태.
- 주요 컴포넌트
  - Chat model
  - Chat prompt
  - Document Loader
  - Retriever
  - Embedding model
  - ...

-  을 사용해서 쉽게 연결 가능하다

```
from langchain.prompts import PromptTemplate
from langchain.llms import OpenAI
from langchain.chains import LLMChain, SimpleSequentialChain

# 첫 번째 프롬프트 템플릿: 음식 추천
food_prompt = PromptTemplate(
    input_variables=["cuisine"],
    template="오늘의 {cuisine} 요리 추천 메뉴 하나만 알려주세요."
)

# 두 번째 프롬프트 템플릿: 레시피 설명
recipe_prompt = PromptTemplate(
    input_variables=["dish"],
    template="{dish}의 간단한 조리 방법을 3줄로 설명해주세요."
)

# LLM 모델 초기화
llm = OpenAI(temperature=0.7)

# 각각의 체인 생성
food_chain = LLMChain(llm=llm, prompt=food_prompt)
recipe_chain = LLMChain(llm=llm, prompt=recipe_prompt)

# 방법 1: 순차적으로 체인 연결
overall_chain = SimpleSequentialChain(
    chains=[food_chain, recipe_chain],
    verbose=True
)
response = overall_chain.run("한식")

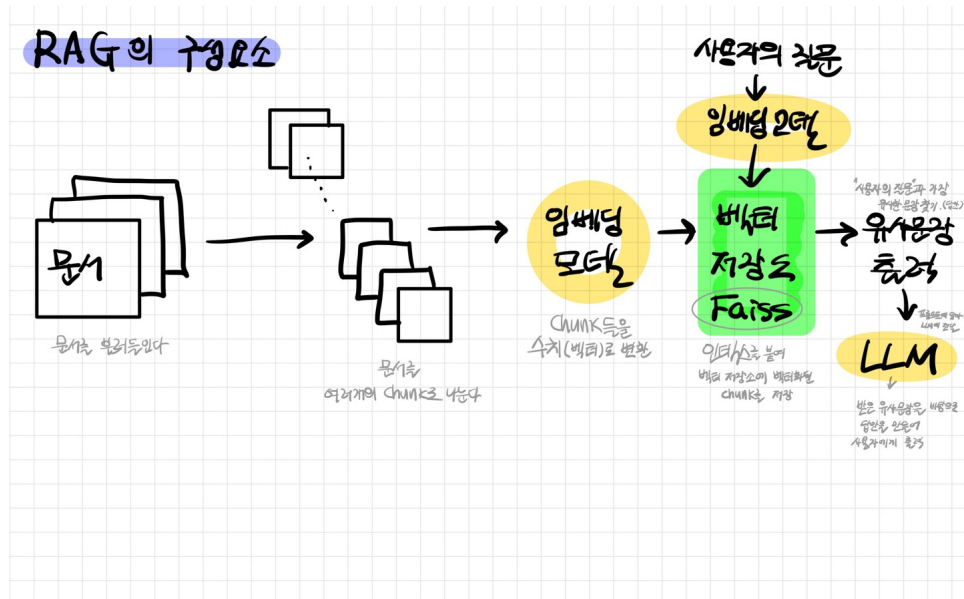
# 방법 2: | 연산자를 사용한 체이닝
```



```
response = food_chain | recipe_chain
result = response.run("한식")
```

## ▼ 참고

- CODE:  
[https://github.com/HarryKane11/langchain/blob/main/streamlit\\_refer.py](https://github.com/HarryKane11/langchain/blob/main/streamlit_refer.py)
- 설명 및 출처: Youtube 모두의 연구소 | <https://www.youtube.com/watch?v=xYNYNKJVa4E>



## 01 문서를 어떻게 불러올 것인지?

```
from langchain.document_loaders import Docx2txtLoader
```

## 02 문서를 어떻게 chunk로 조각낼 것인지?

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
```

```
def tiktoken_len(text):
    # token개수를 기준으로 text를 Chunk로 split해주기 위한 함수
    tokenizer = tiktoken.get_encoding("cl100k_base")
```

```
tokens = tokenizer.encode(text)
return len(tokens)
```

```
def get_text_chunks(text):
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=900, # 900개 토큰을 기준으로 chunk를 자른다.
        chunk_overlap=100, # chunk들이 맥락을 어느정도 파악할 수
        있도록 100개의 토큰을 앞뒤로 겹치도록 자른다.
        length_function=tiktoken_len # chunk를 세는 기준
    )
    chunks = text_splitter.split_documents(text)
    return chunks
```

- 문자 개수 기준 - token 개수

### 03 chunk들을 어떻게 수치화 할 것인지?

- 한글 특화 임베딩 모델 - Huggingface 모델 사용

```
from langchain.embeddings import HuggingFaceEmbeddings

embeddings = HuggingFaceEmbeddings(
    model_name="jhgan/ko-sroberta-multitask", # 사용할 모
    델 이름 (HuggingFace 모델 허브에서 모델을 불러옴)
    model_kwargs={'device': 'cpu'}, # 모델을 실행할 디바이
    스 설정 ('cpu' 또는 'cuda' 등)
    encode_kwargs={'normalize_embeddings': True} # 임베
    딩 결과를 정규화할지 여부 (True로 설정하면 임베딩을 정규화)
```

### 04 벡터로 수치화 된 chunk들을 어디에, 어떻게 저장할 것인지?

- faiss
  - 저장 기간을 짧지만, 임시로 저장 + 로컬에 구축하는 것 보다 조금 더 빠른 속도

```
from langchain.vectorstores import FAISS
```

```
vectoradb = FAISS.from_documents(text_chunks, embeddings)
```

## 05 유사한 문장을 어떻게 찾을 것인지

```
from langchain.chains import ConversationalRetrievalChain

conversation_chain = ConversationalRetrievalChain.from_llm(
    llm=llm, # 이전에 정의한 LLM 인스턴스를 전달 (대화형 응답 생성을 담당)
    chain_type="stuff", # 체인의 유형 ('stuff'는 응답을 생성할 때 관련된 정보를 모두 사용. 즉, 검색된 모든 정보를 모델에게 넘기고, 그 정보를 바탕으로 응답을 생성)
    retriever=vectordb.as_retriever(search_type='mmr', verbose=True), # 검색기 정의
    # retriever: 벡터 저장소를 검색하여 관련 정보를 반환하는 역할 (MMR 방식으로 검색, verbose=True로 디버깅 출력 활성화)
    # MMR: 검색 결과에서 중복을 최소화하면서 가장 관련성 높은 문서를 선택하는 방법
    memory=ConversationBufferMemory(memory_key='chat_history', return_messages=True, output_key='answer'),
    # memory: 대화의 기록을 저장하고 불러오는 메모리 관리 객체
    # memory_key: 메모리에서 대화 기록을 찾을 때 사용하는 키
    # return_messages: True로 설정하면 메시지들을 반환
    # output_key: 대화를 나눈 내용 중 답변에 해당하는 것만 history에 담겠다는 의미
    get_chat_history=lambda h: h, # 대화 기록을 가져오는 함수 (그대로 반환)
    return_source_documents=True, # 응답과 함께 관련된 원본 문서도 반환
    verbose=True # 디버깅을 위해 자세한 로그를 출력
)
```

## 06 어떤 LLM을 사용할 것인지

```
from langchain.chat_models import ChatOpenAI
```

```
llm = ChatOpenAI(openai_api_key=openai_api_key, model_name
='gpt-3.5-turbo', temperature=0)
# ChatOpenAI 인스턴스를 생성하고 OpenAI API 키와 모델명을 전달
# openai_api_key: OpenAI API 키 (환경 변수 등으로 관리)
# model_name: 사용할 모델 이름 ('gpt-3.5-turbo' 사용, GPT-3.5
모델)
# temperature: 생성되는 응답의 창의성 정도 설정 (0이면 결정적이고 정
확한 응답, 1이면 더 창의적인 응답)
```