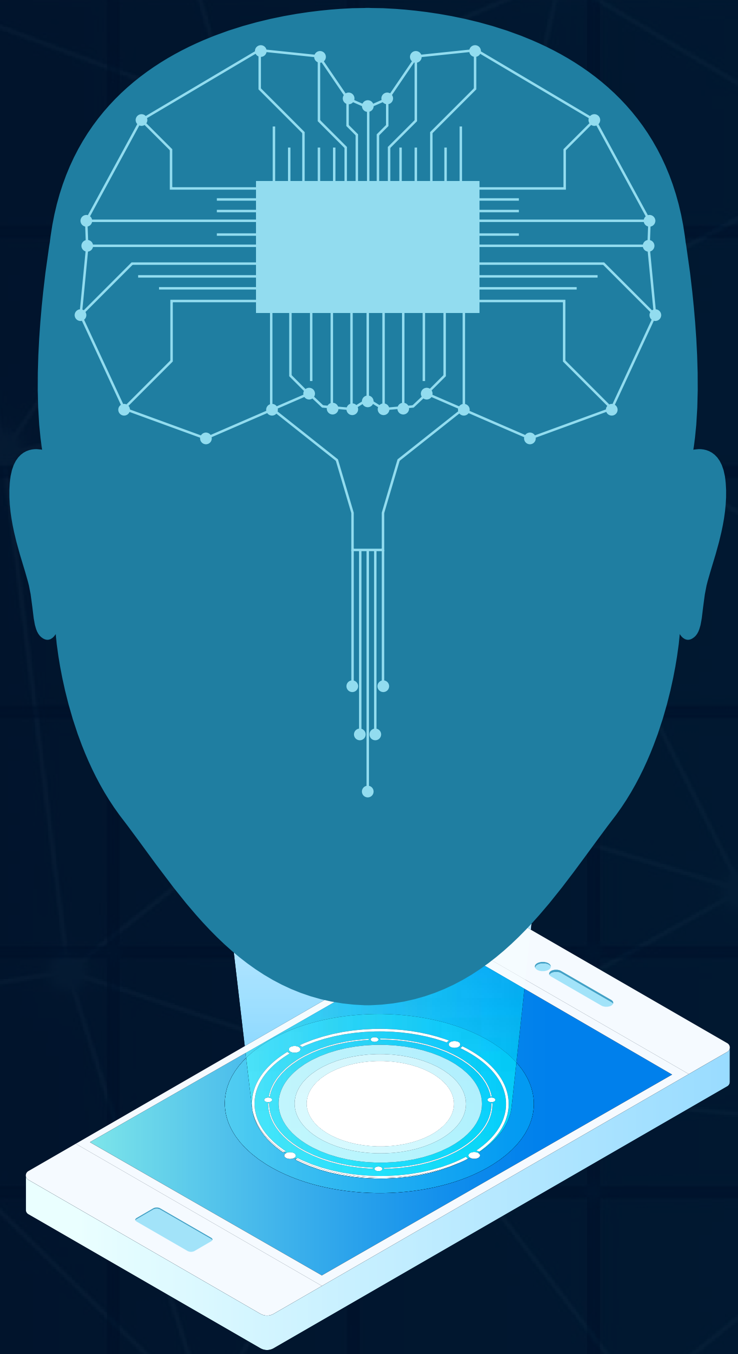


LANGCHAIN COMPONENTS

Presented by:

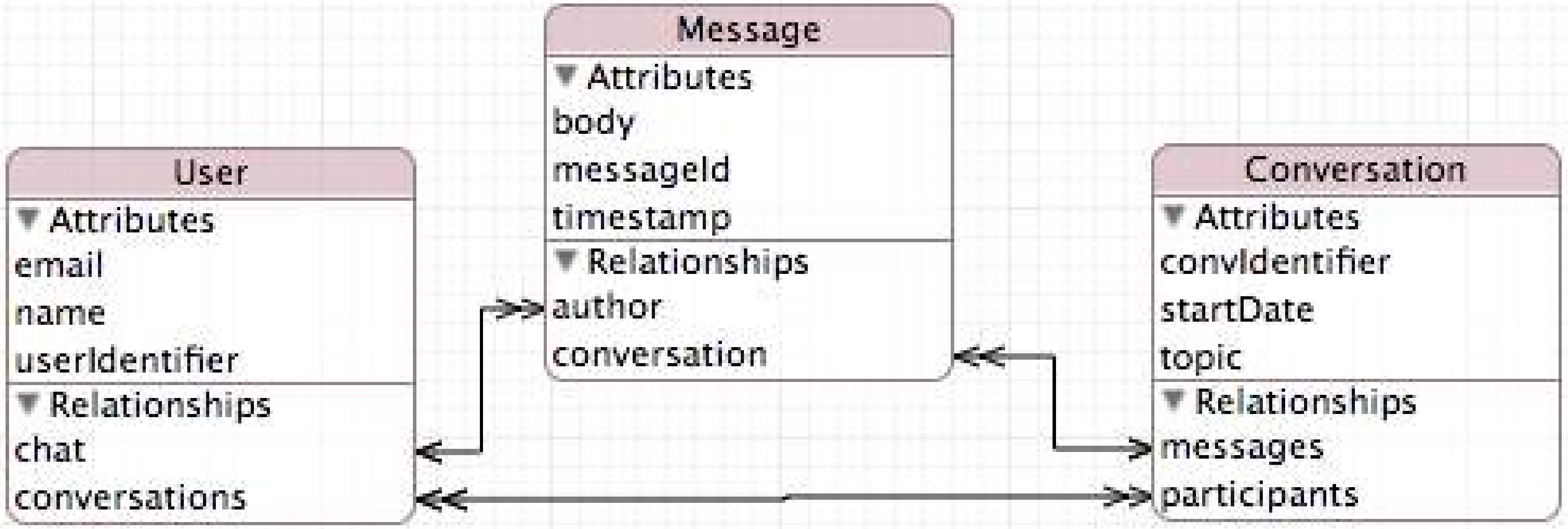
Khristina Pershina

TABLE OF CONTENT

01 CHAT MODELS

- Chat Models are a special type of language model designed for conversations.
- Unlike regular language models that process plain text, Chat Models take chat messages (think back-and-forth dialogue) as input and generate replies as output.

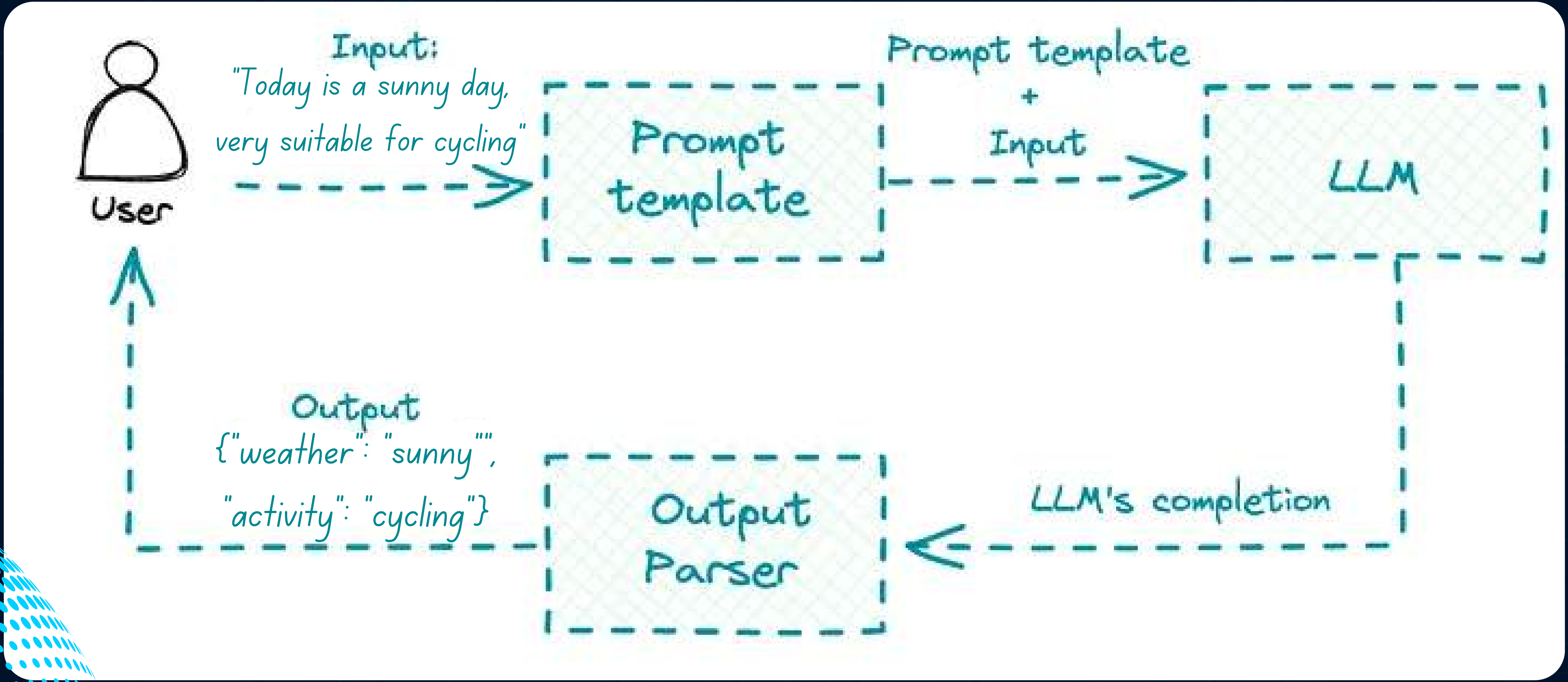
- Modern LLMs are typically accessed through a chat model interface that takes a list of messages as input and returns a message as output.



02

PROMPT TEMPLATES

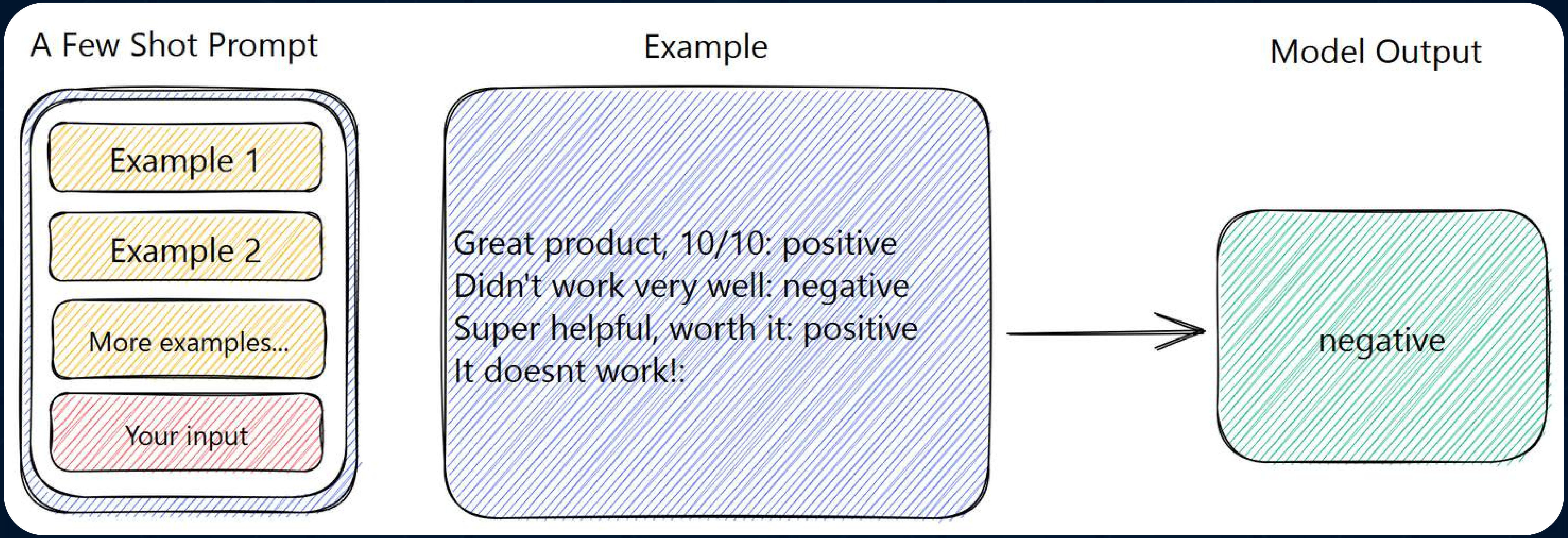
- Prompt templates help to translate user input and parameters into instructions for a language model.
- This guides a model's response, helps it understand the context, and generate relevant and coherent language-based output.



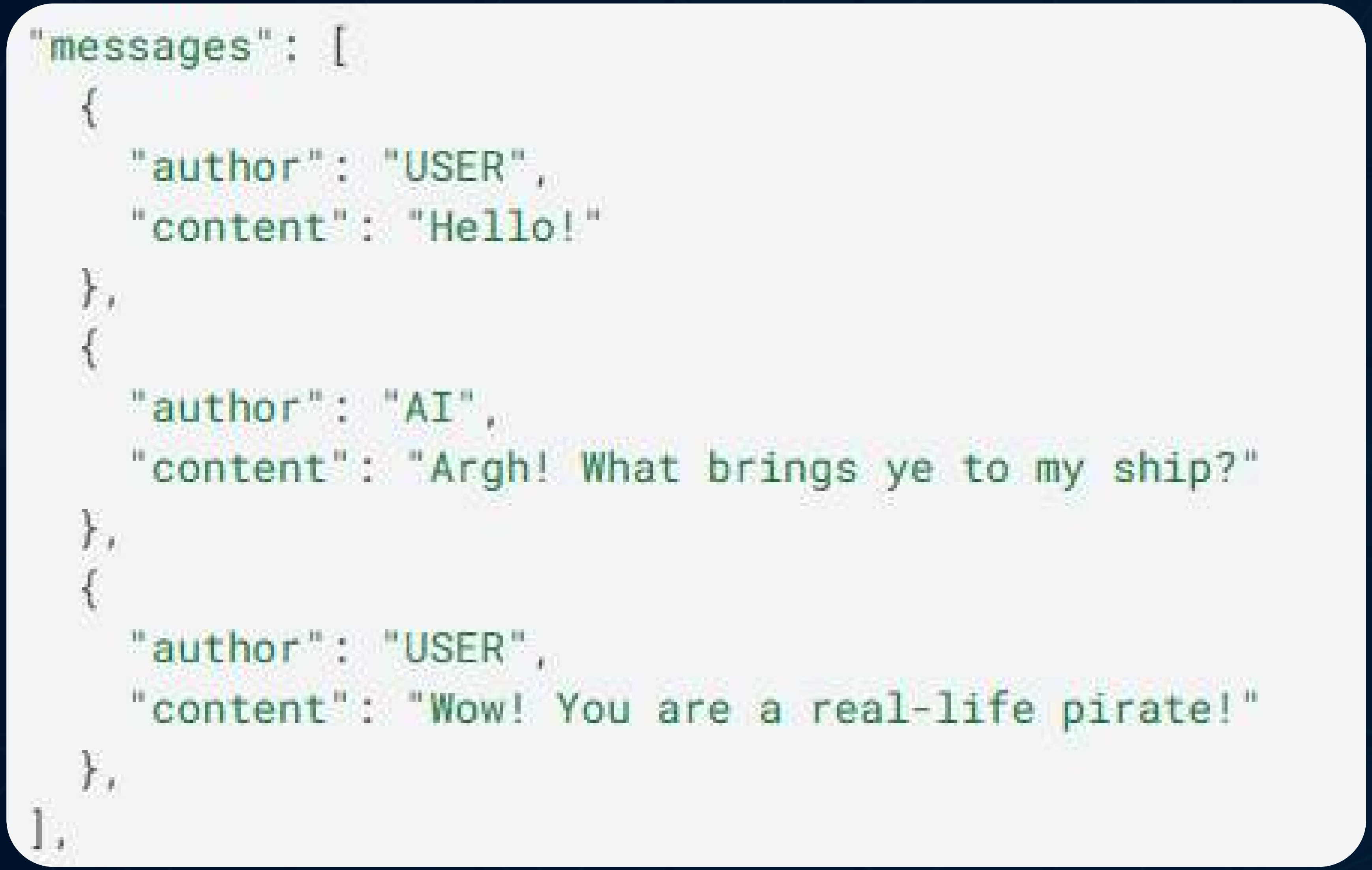
03

N-SHOT PROMPTING

- Zero-shot Prompting -- model receives no examples;
- One-shot Prompting -- model receives one example;
- It is common to include examples as part of the prompt to achieve better performance. This is known as **few-shot prompting.**



04 MESSAGES



```
"messages": [
  {
    "author": "USER",
    "content": "Hello!"
  },
  {
    "author": "AI",
    "content": "Argh! What brings ye to my ship?"
  },
  {
    "author": "USER",
    "content": "Wow! You are a real-life pirate!"
  },
],
```

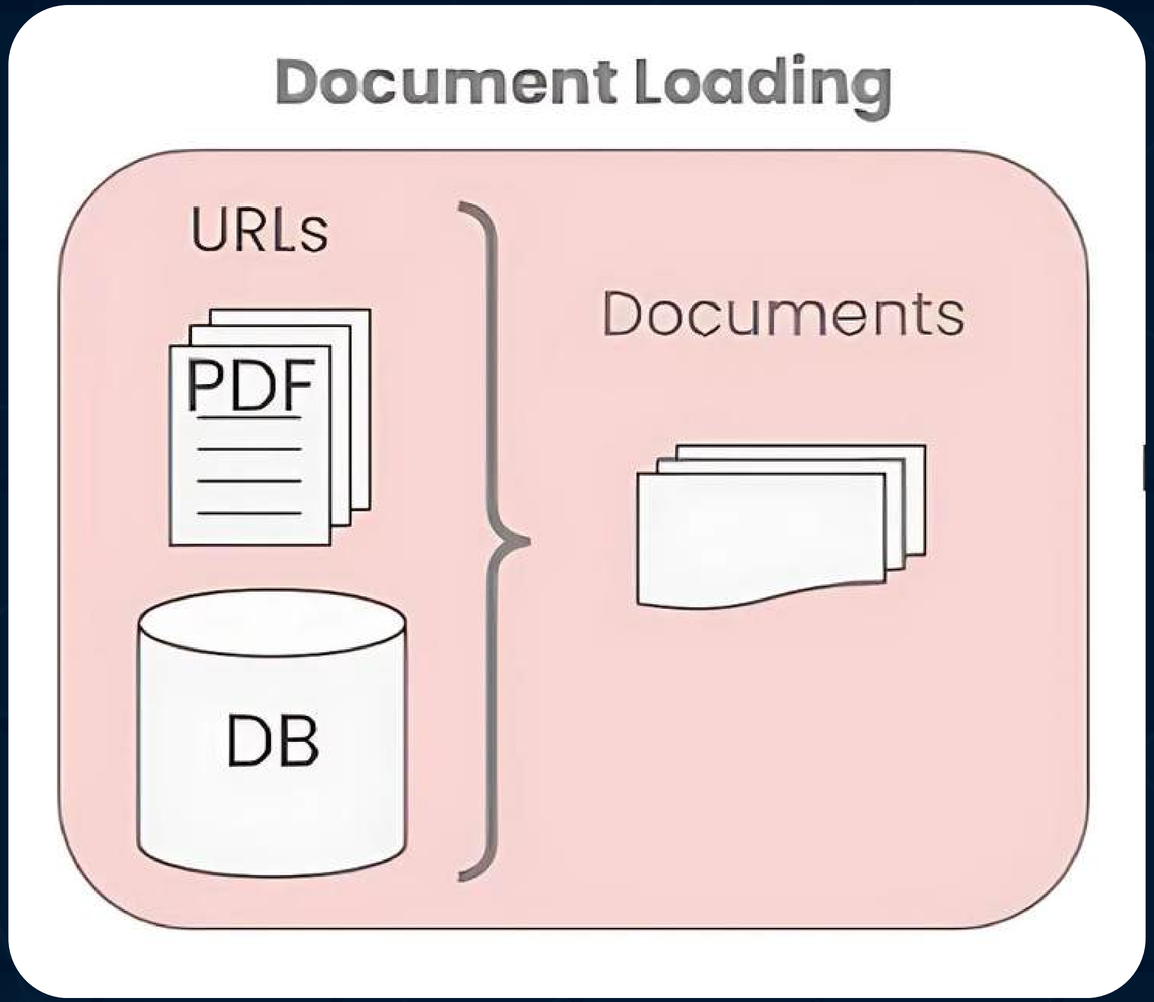
- Messages are the unit of communication in chat models.
- Represent the input and output of a chat model.

A message typically consists of :

- **Role:** "user", "assistant";
- **Content:** text, multimodal data.
- Additional **metadata:** id, name, token usage etc.

05 DOCUMENT LOADERS

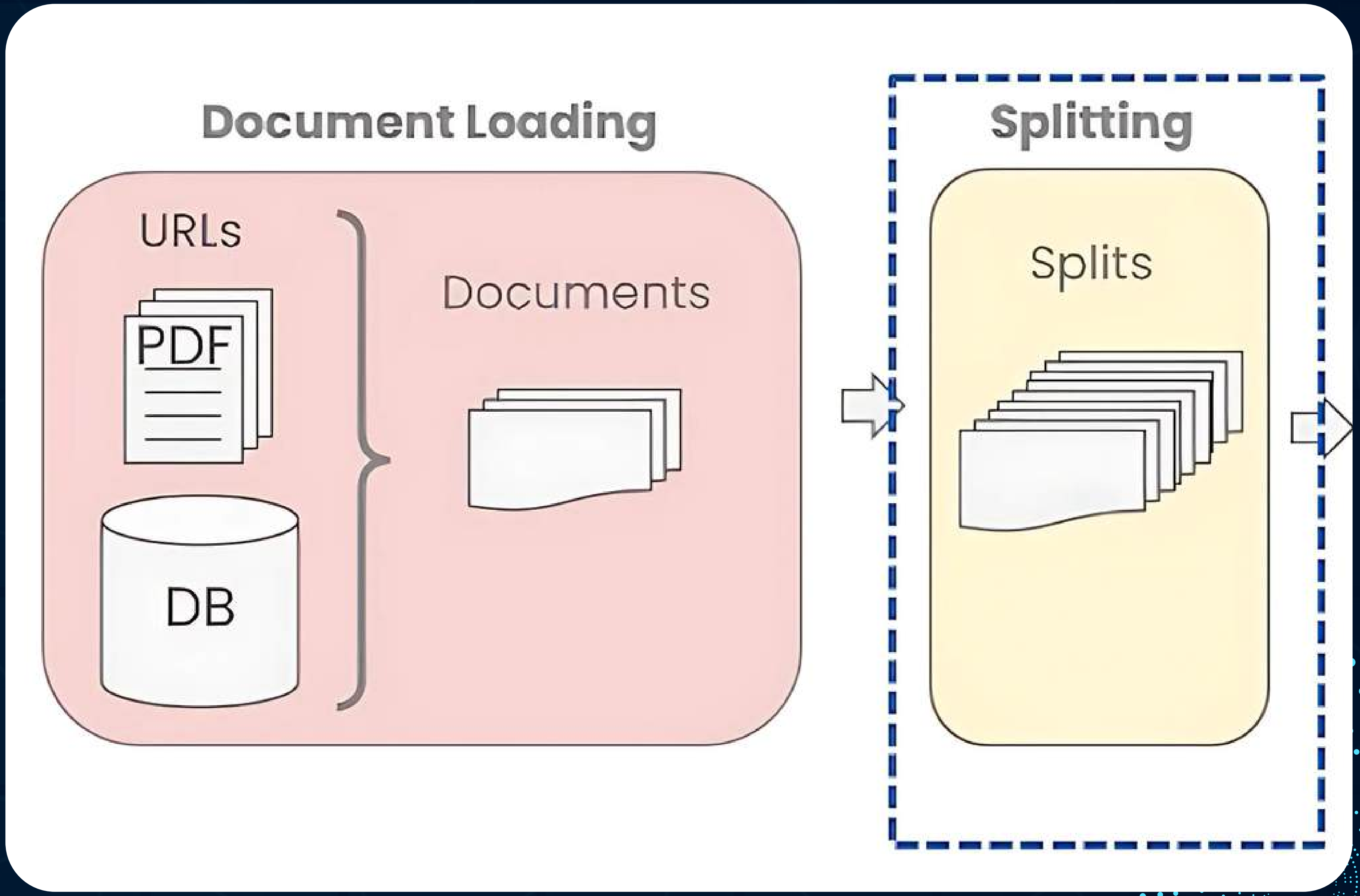
- Document loaders are designed to load document objects.
- LangChain has hundreds of integrations with various data sources to load data from: Slack, Notion, Google Drive, etc.



06 TEXT SPLITTERS

Why split documents?

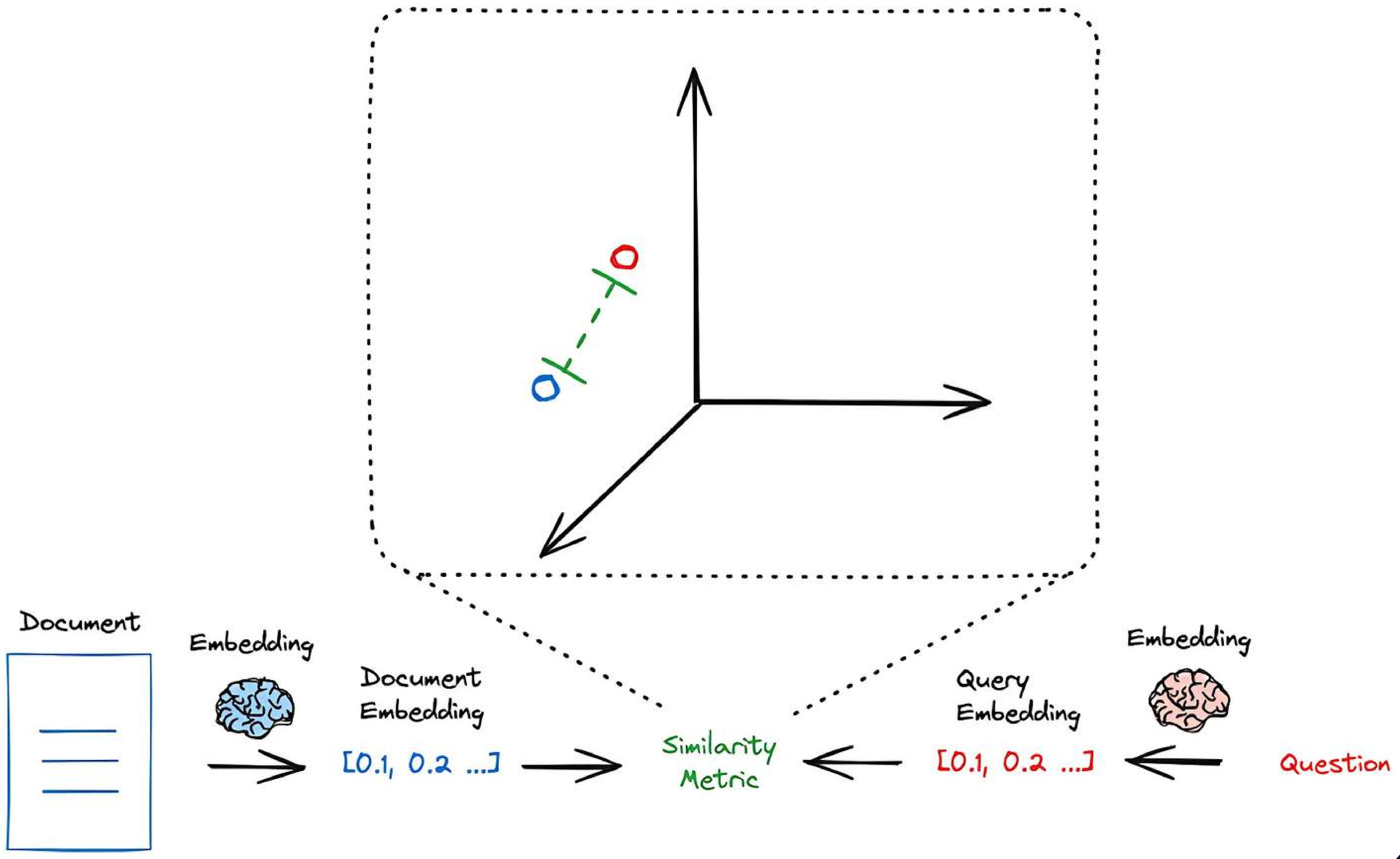
- Handling non-uniform document lengths
- Overcoming model limitations
- Improving representation quality
- Enhancing retrieval precision
- Optimizing computational resources



07 EMBEDDING MODELS

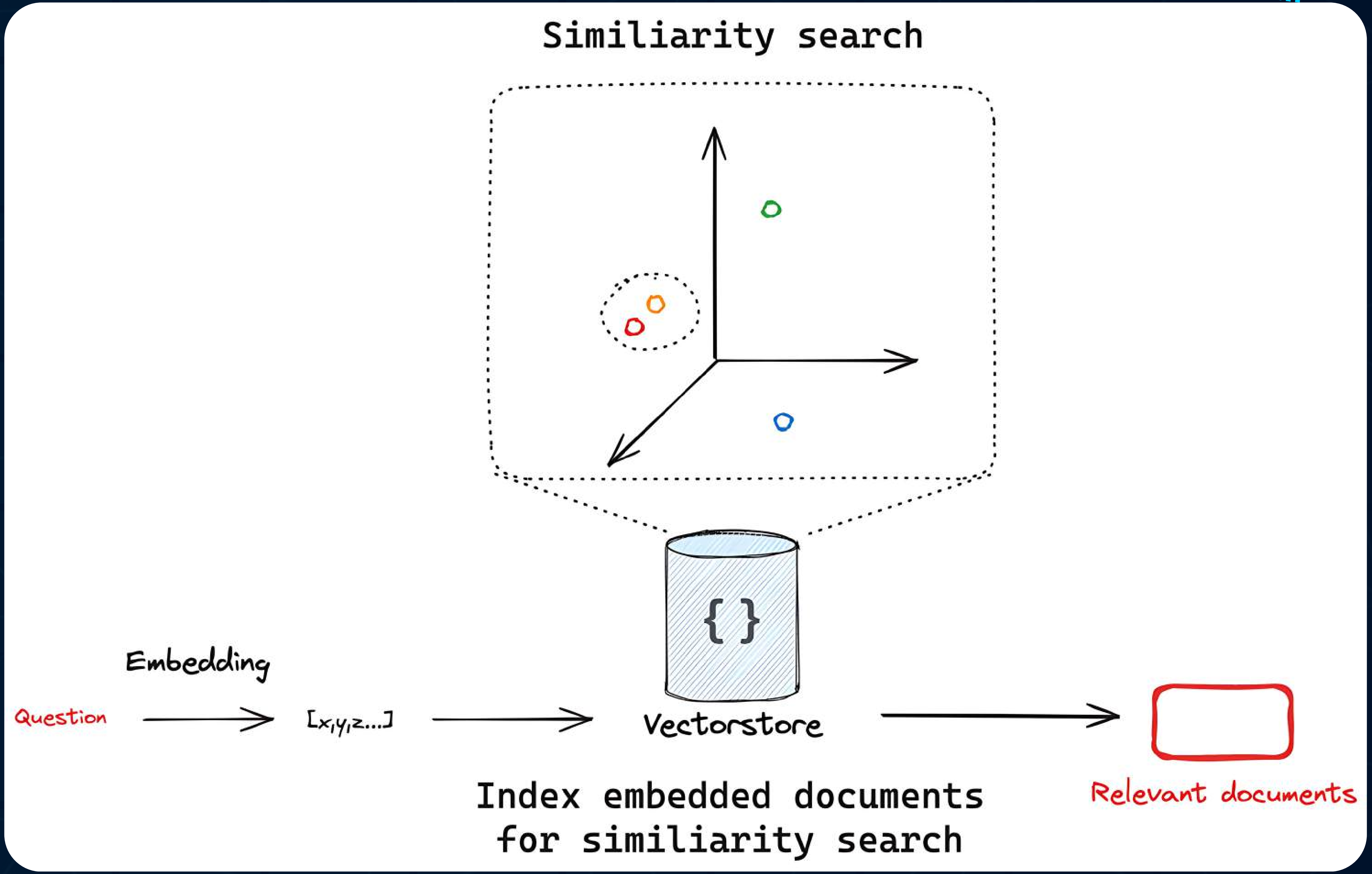
- Embed text as vector;
- Measure similarity between texts;
- Capture the semantic meaning of data that has been embedded.
- Lie at the heart of many **retrieval systems.**

- Cosine SImilarity
- Euclidian Distance
- Dot Product



08 VECOTR STORES

- Specialized data stores that enable **indexing and retrieving** information based on vector representations (embeddings).
- **Search** over unstructured data (text, images, and audio).
- Retrieve relevant information based on **semantic similarity** rather than exact keyword matches.



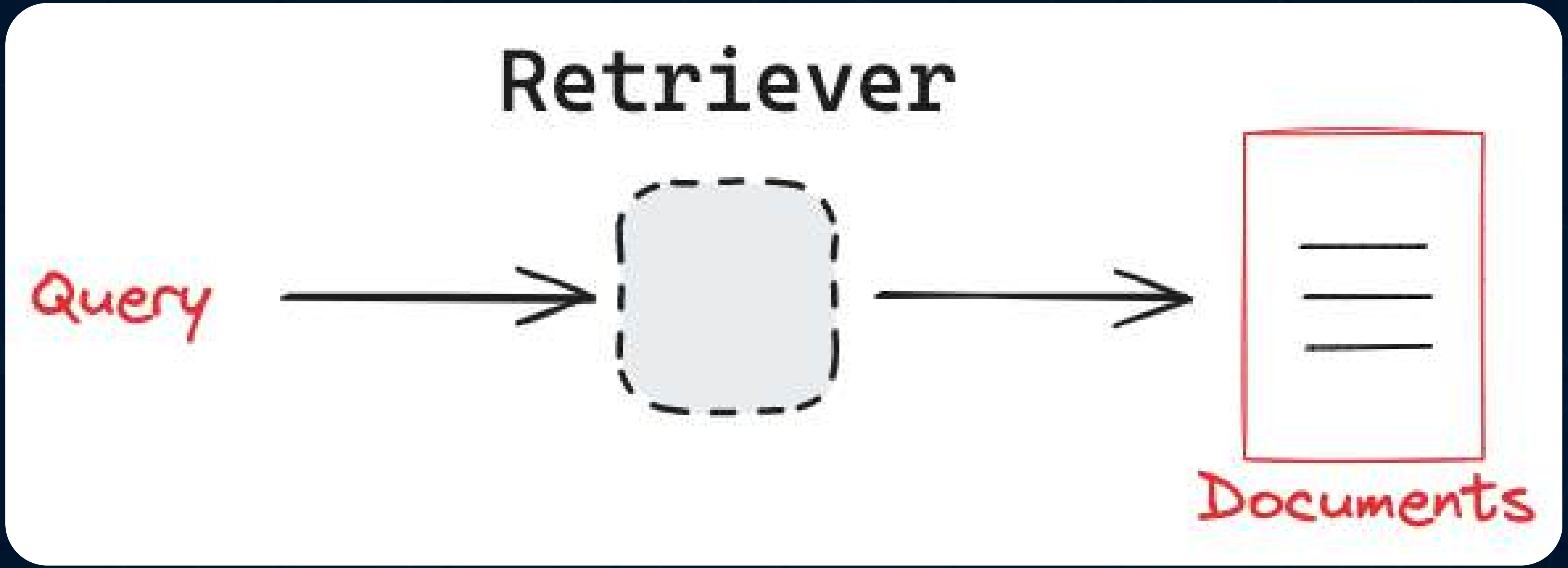
09 INDEXING

- Load and keep in sync documents from any source into a vector store;
- Avoid writing duplicated content into the vector store;
- Avoid re-writing unchanged content;
- Avoid re-computing embeddings over unchanged content;
- Save you time and money
- Improve your vector search results.

10 RETRIEVERS

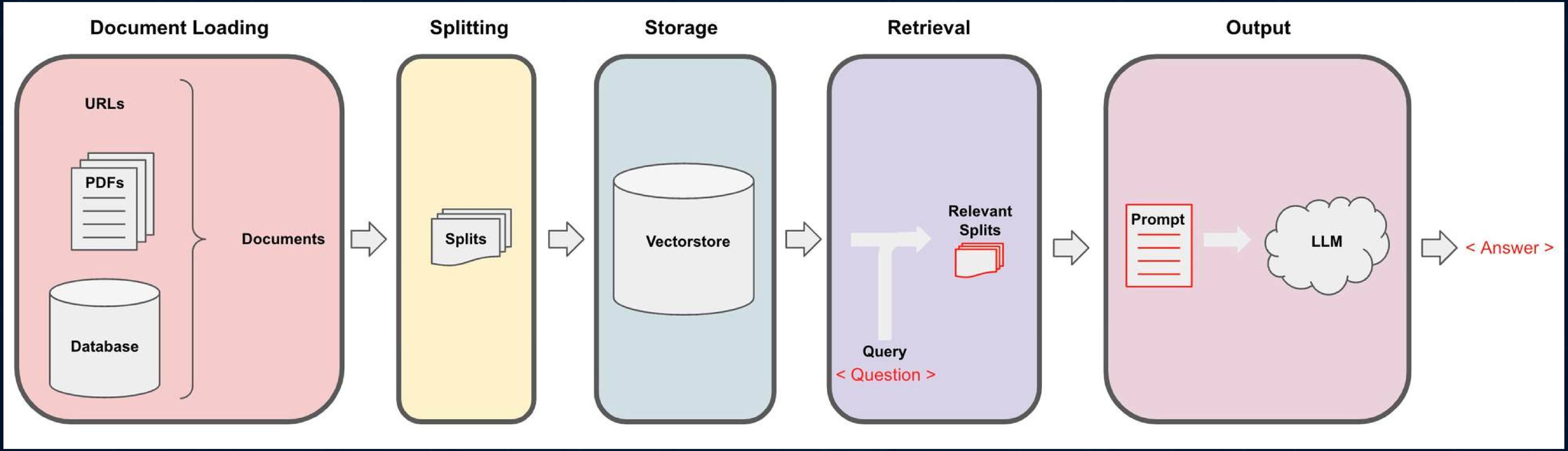
- Types: vectorstores, graph databases, and relational databases.
- Important component in AI application (e.g., RAG).
- Input: A query (string)
- Output: A list of documents.

BM25 and TF-IDF are two popular lexical search algorithms.



11 PRACTICE

Recall the overall workflow for retrieval augmented generation (RAG):

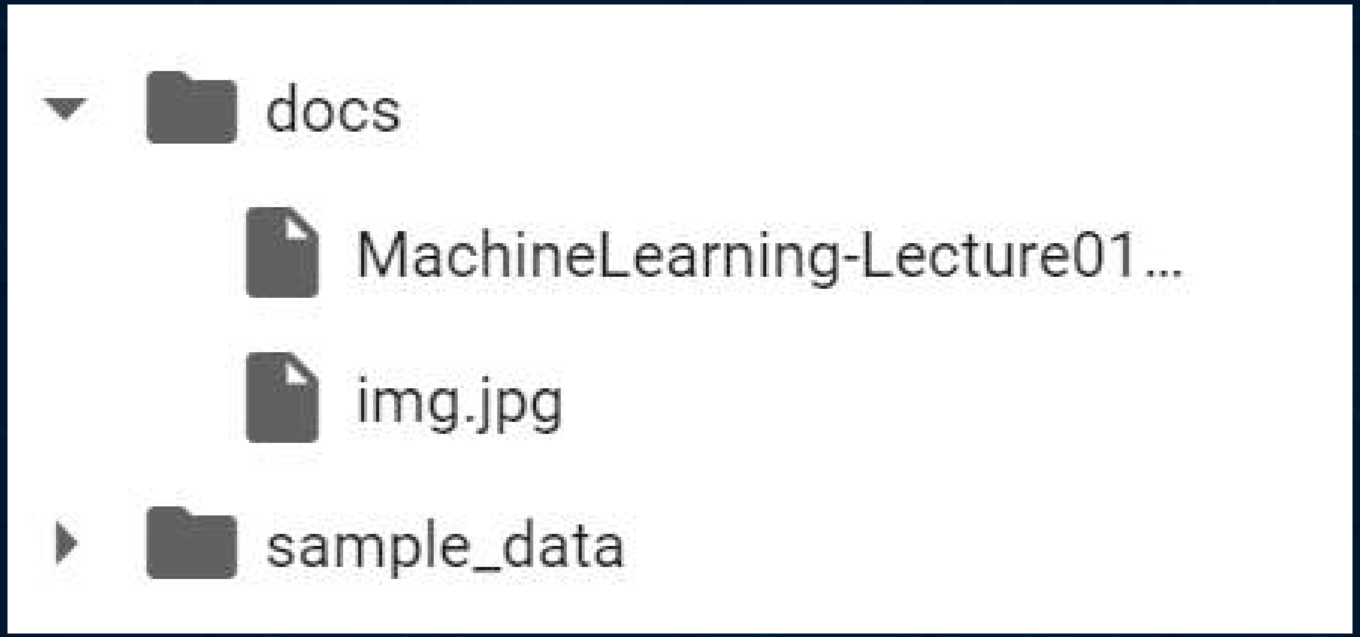


11

PRACTICE

Data is here

First, organize your working space:

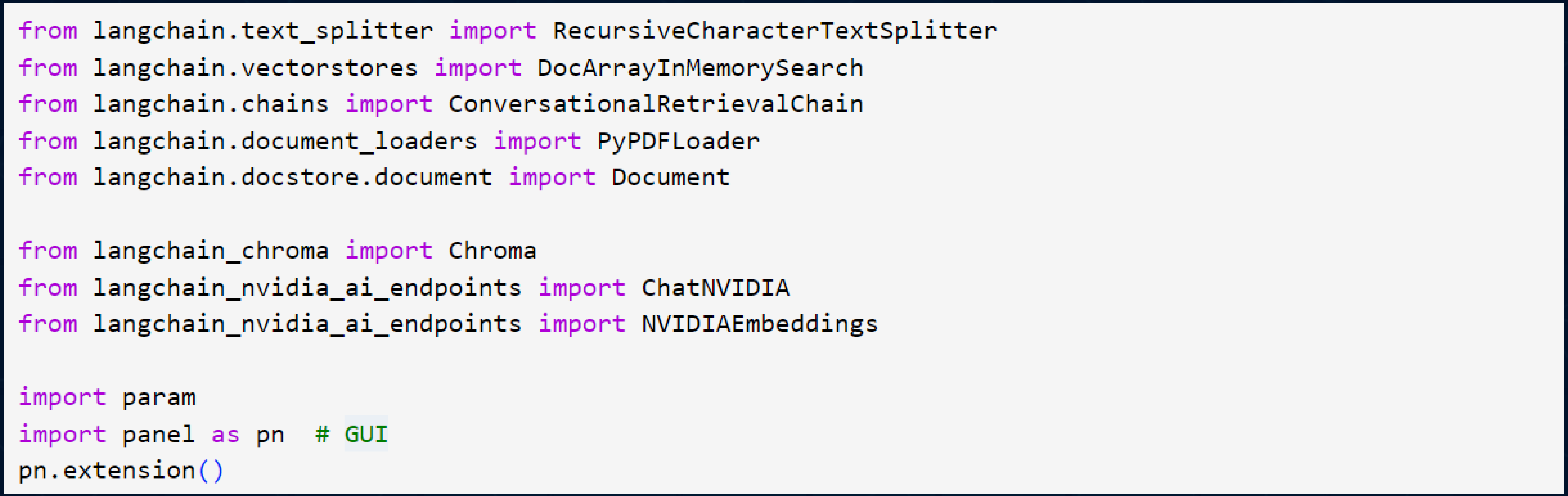

docs
MachineLearning-Lecture01...
img.jpg
sample_data

Then, install dependencies:

```
!pip install langchain-community langchain_ollama langchain_chroma langchain_nvidia_ai_endpoints --quiet
!pip install pypdf docarray protobuf==4.* jupyter_bokeh --quiet
```

11 PRACTICE

Import necessary libraries:



```python
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.vectorstores import DocArrayInMemorySearch
from langchain.chains import ConversationalRetrievalChain
from langchain.document_loaders import PyPDFLoader
from langchain.docstore.document import Document

from langchain_chroma import Chroma
from langchain_nvidia_ai_endpoints import ChatNVIDIA
from langchain_nvidia_ai_endpoints import NVIDIAEmbeddings

import param
import panel as pn  # GUI
pn.extension()
```

11 PRACTICE

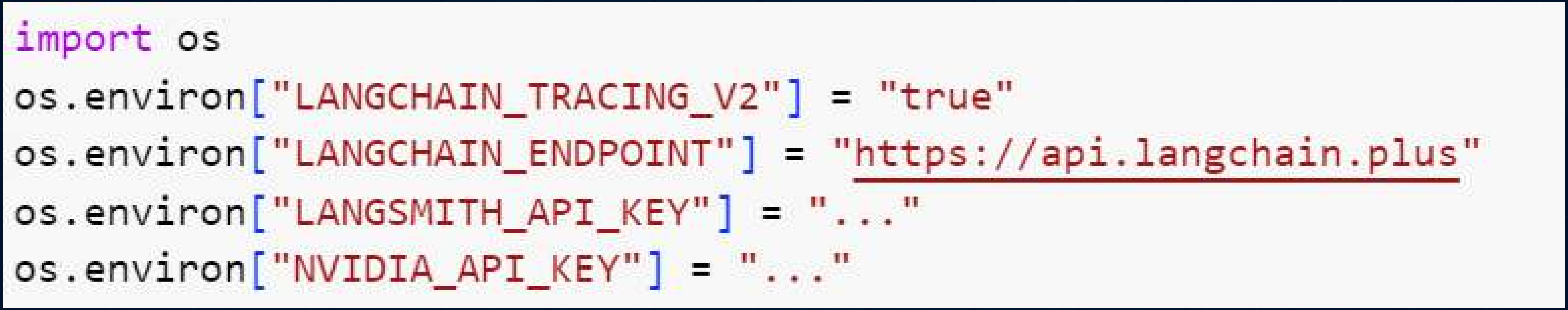
Register your LangChain and NVIDIA API keys.

LangChain:

1. Go to https://smith.langchain.com/
2. Head to the **Settings** page.
3. Scroll to the **API Keys** section. Then click **Create API Key.**
4. **Copy&Paste** into **“LANGSMITH_API_KEY“.**

NVIDIA:

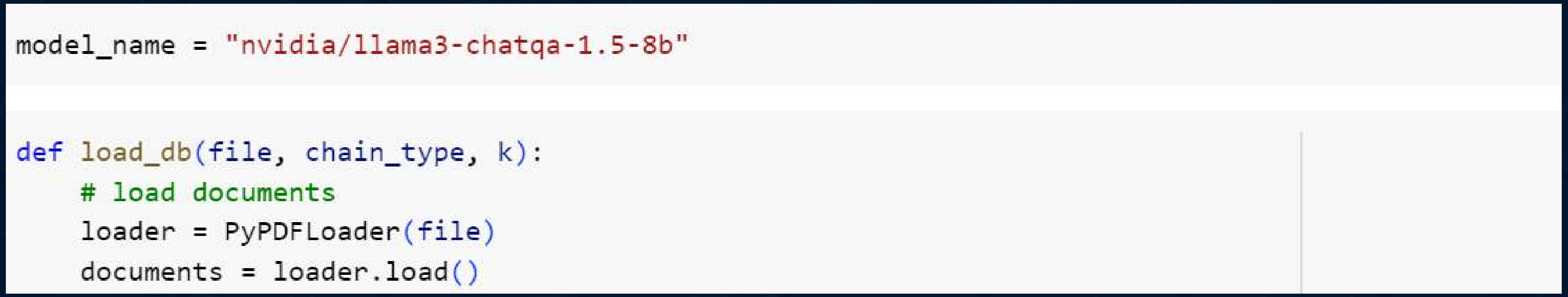
1. Go to https://org.ngc.nvidia.com/setup/api-key
2. Hit “Create API Key” and register with your account.
3. Copy&Paste into **“NVIDIA_API_KEY”.**



```
import os
os.environ["LANGCHAIN_TRACING_V2"] = "true"
os.environ["LANGCHAIN_ENDPOINT"] = "https://api.langchain.plus"
os.environ["LANGSMITH_API_KEY"] = "..."
os.environ["NVIDIA_API_KEY"] = "..."
```

11 PRACTICE

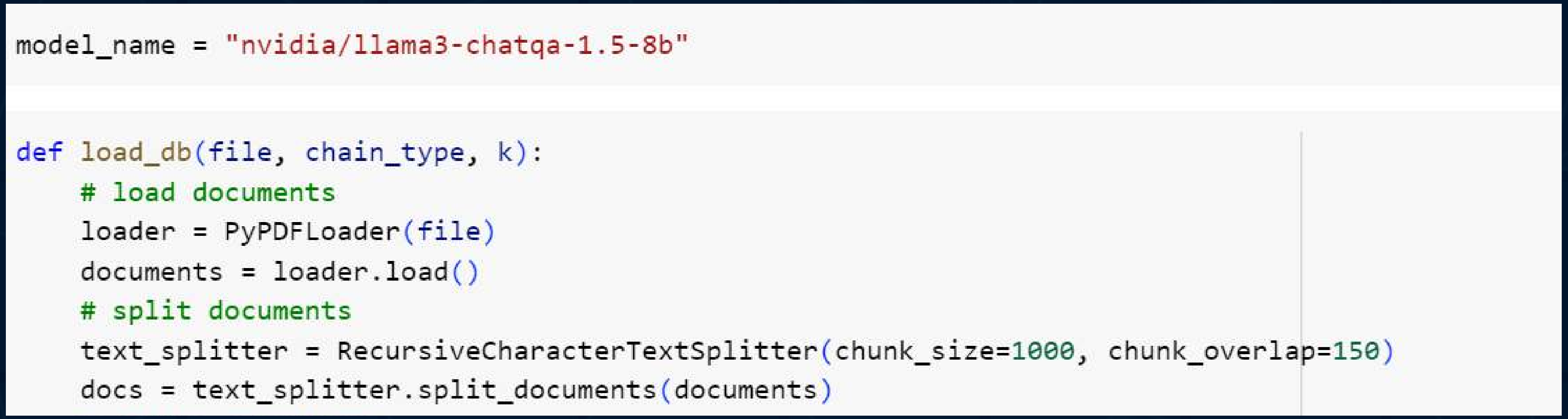
Create a Converstation Retrieval Chain:



```
model_name = "nvidia/llama3-chatqa-1.5-8b"

def load_db(file, chain_type, k):
    # load documents
    loader = PyPDFLoader(file)
    documents = loader.load()
```

11 PRACTICE

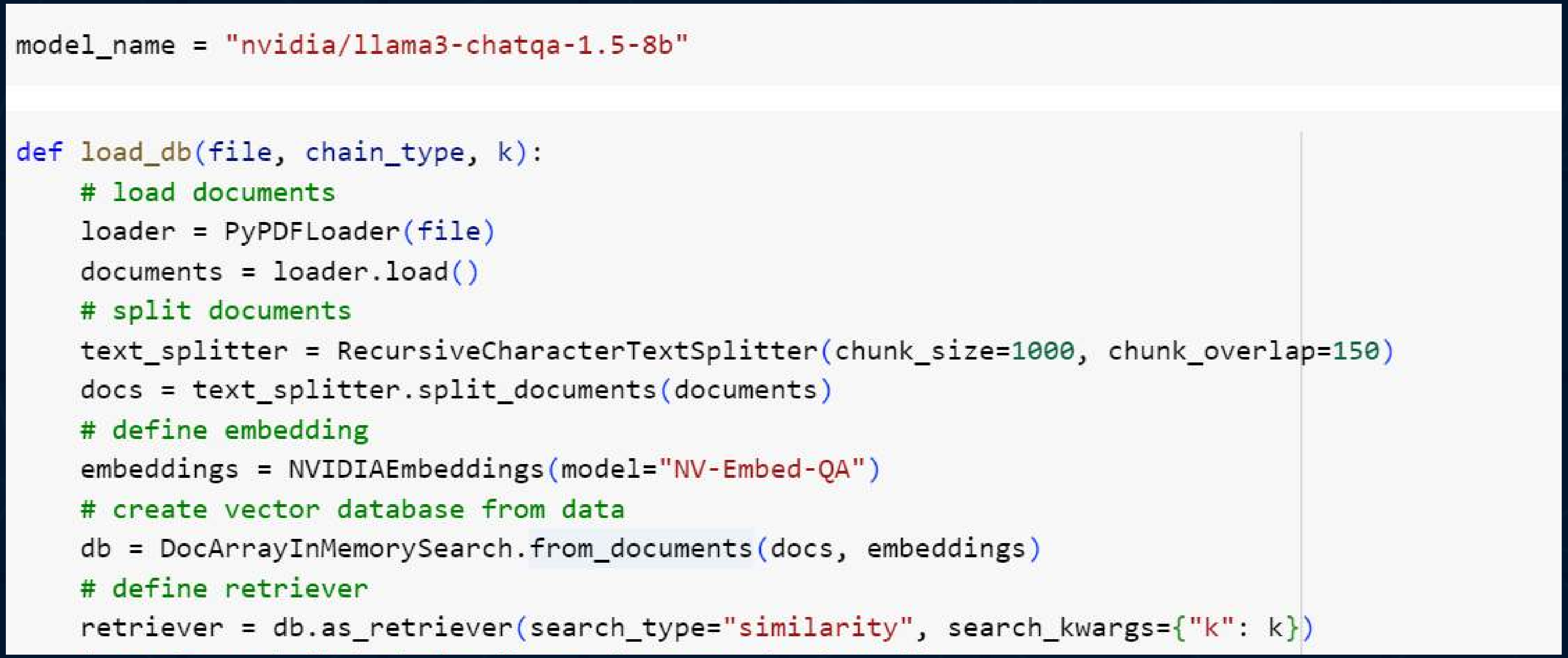
Create a Converstation Retrieval Chain:



```
model_name = "nvidia/llama3-chatqa-1.5-8b"

def load_db(file, chain_type, k):
    # load documents
    loader = PyPDFLoader(file)
    documents = loader.load()
    # split documents
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=150)
    docs = text_splitter.split_documents(documents)
```

11 PRACTICE

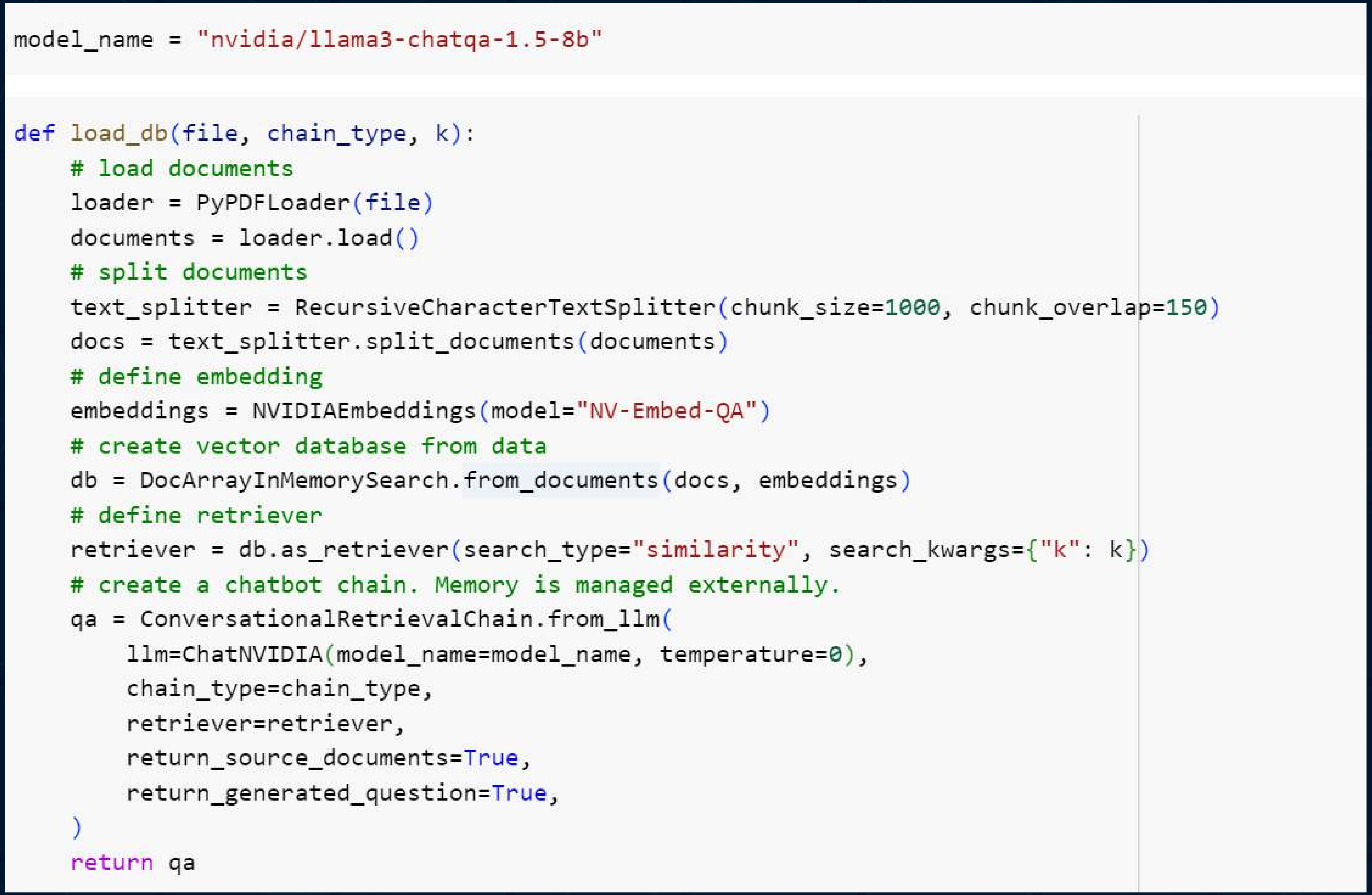
Create a Converstation Retrieval Chain:



```
model_name = "nvidia/llama3-chatqa-1.5-8b"

def load_db(file, chain_type, k):
    # load documents
    loader = PyPDFLoader(file)
    documents = loader.load()
    # split documents
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=150)
    docs = text_splitter.split_documents(documents)
    # define embedding
    embeddings = NVIDIAEmbeddings(model="NV-Embed-QA")
    # create vector database from data
    db = DocArrayInMemorySearch.from_documents(docs, embeddings)
    # define retriever
    retriever = db.as_retriever(search_type="similarity", search_kwargs={"k": k})
```

11 PRACTICE

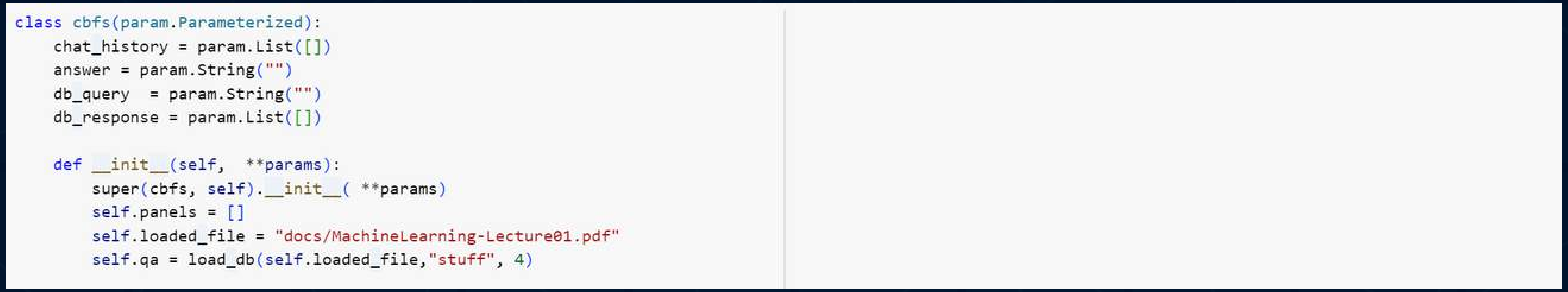
Create a Converstation Retrieval Chain:



```python
model_name = "nvidia/llama3-chatqa-1.5-8b"

def load_db(file, chain_type, k):
    # load documents
    loader = PyPDFLoader(file)
    documents = loader.load()
    # split documents
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=150)
    docs = text_splitter.split_documents(documents)
    # define embedding
    embeddings = NVIDIAEmbeddings(model="NV-Embed-QA")
    # create vector database from data
    db = DocArrayInMemorySearch.from_documents(docs, embeddings)
    # define retriever
    retriever = db.as_retriever(search_type="similarity", search_kwargs={"k": k})
    # create a chatbot chain. Memory is managed externally.
    qa = ConversationalRetrievalChain.from_llm(
        llm=ChatNVIDIA(model_name=model_name, temperature=0),
        chain_type=chain_type,
        retriever=retriever,
        return_source_documents=True,
        return_generated_question=True,
    )
    return qa
```

11 PRACTICE

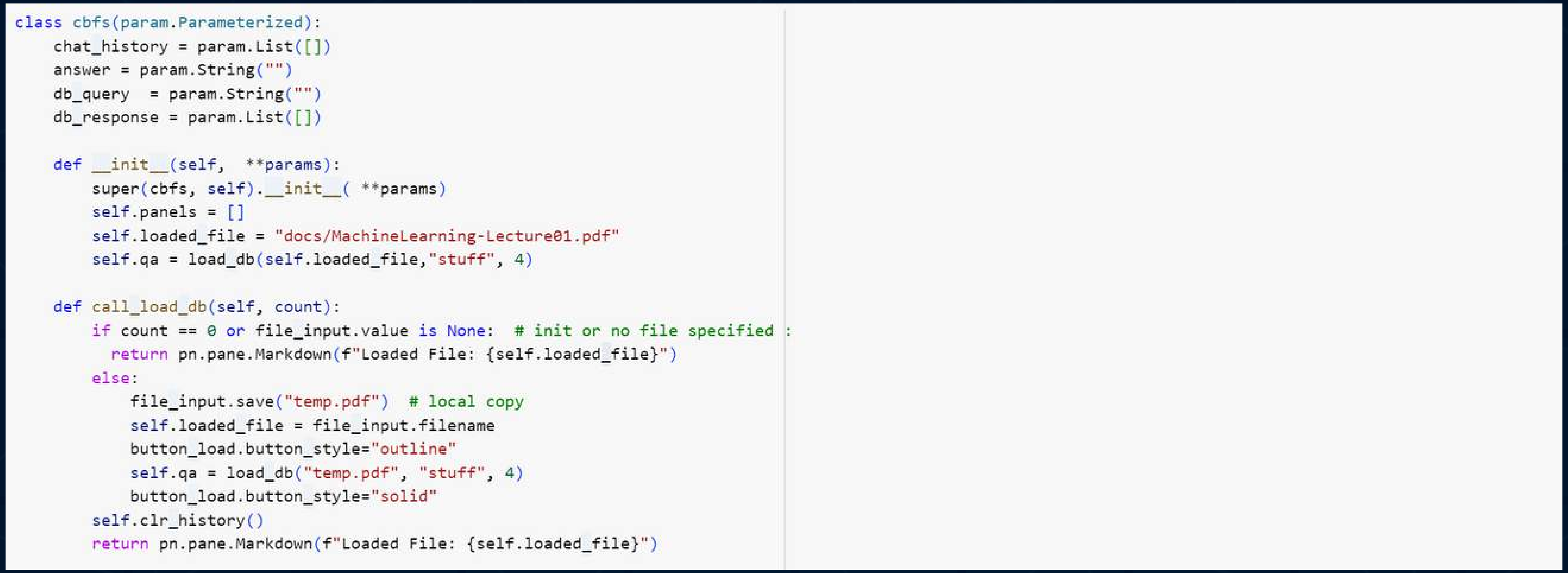
Create UI



```
class cbfs(param.Parameterized):
    chat_history = param.List([])
    answer = param.String("")
    db_query  = param.String("")
    db_response = param.List([])

    def __init__(self,  **params):
        super(cbfs, self).__init__( **params)
        self.panels = []
        self.loaded_file = "docs/MachineLearning-Lecture01.pdf"
        self.qa = load_db(self.loaded_file,"stuff", 4)
```

11 PRACTICE

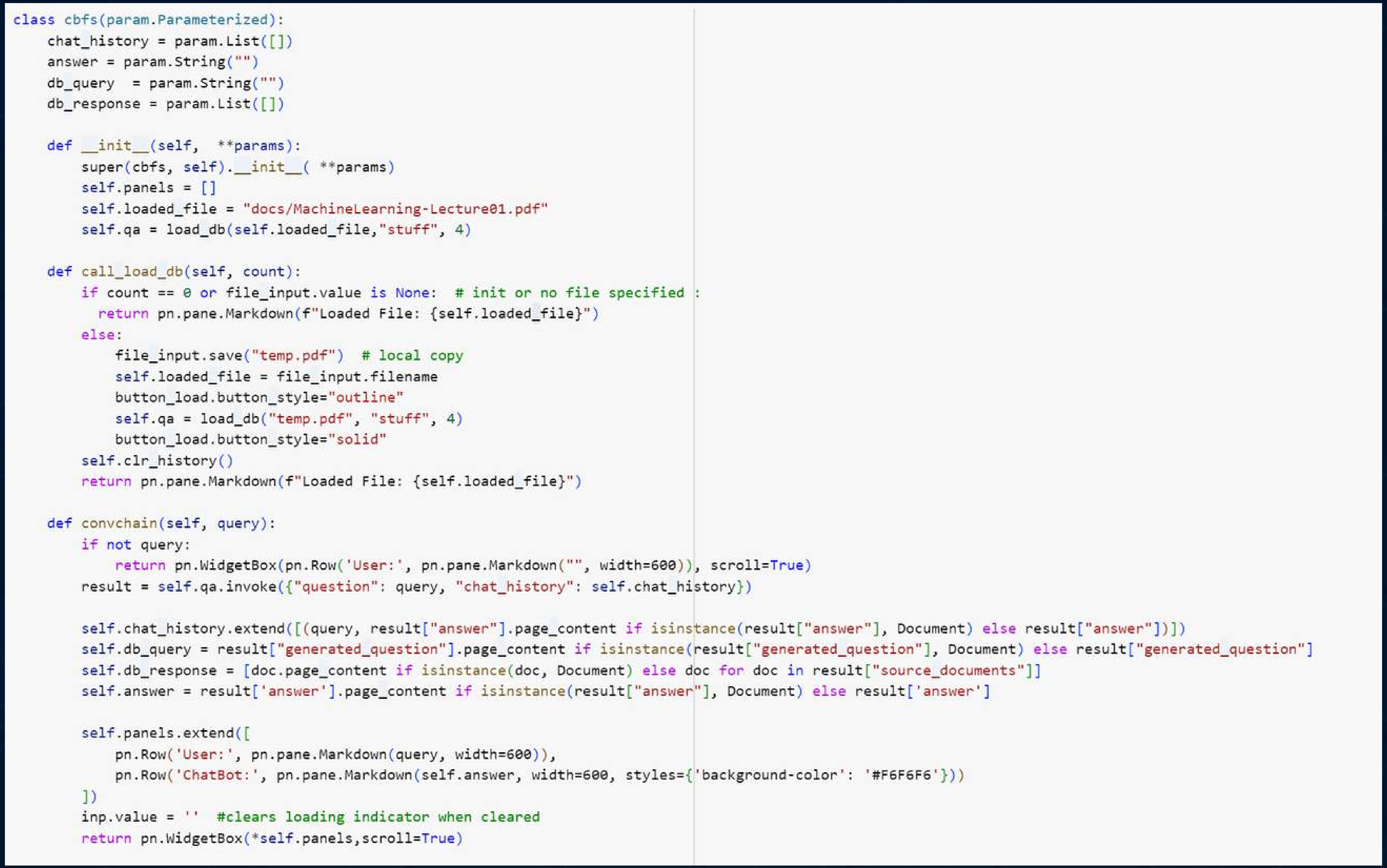
Create UI



```python
class cbfs(param.Parameterized):
    chat_history = param.List([])
    answer = param.String("")
    db_query  = param.String("")
    db_response = param.List([])

    def __init__(self,  **params):
        super(cbfs, self).__init__( **params)
        self.panels = []
        self.loaded_file = "docs/MachineLearning-Lecture01.pdf"
        self.qa = load_db(self.loaded_file,"stuff", 4)

    def call_load_db(self, count):
        if count == 0 or file_input.value is None:  # init or no file specified :
          return pn.pane.Markdown(f"Loaded File: {self.loaded_file}")
        else:
            file_input.save("temp.pdf")  # local copy
            self.loaded_file = file_input.filename
            button_load.button_style="outline"
            self.qa = load_db("temp.pdf", "stuff", 4)
            button_load.button_style="solid"
        self.clr_history()
        return pn.pane.Markdown(f"Loaded File: {self.loaded_file}")
```

11 PRACTICE

Create UI

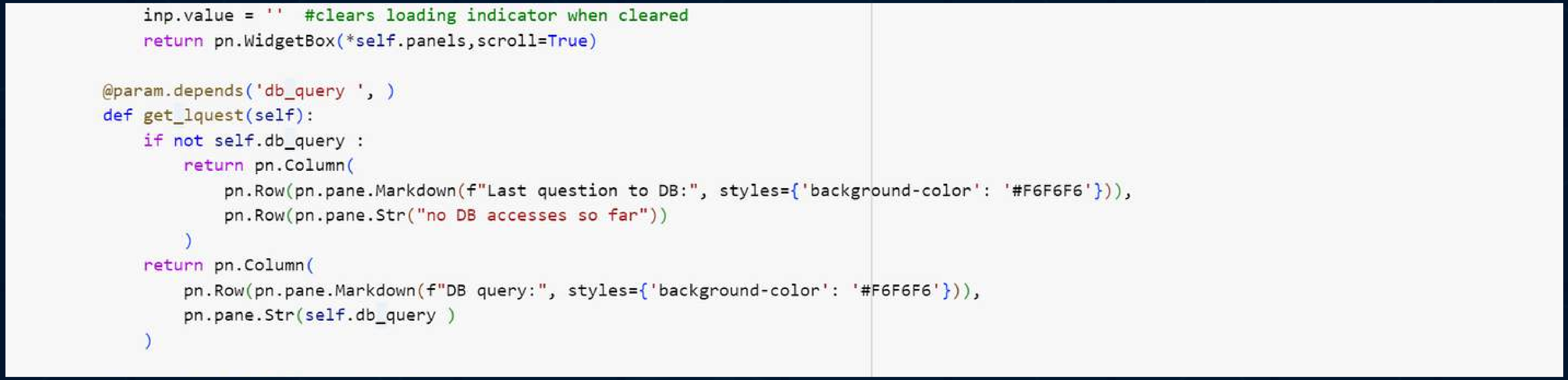


```python
class cbfs(param.Parameterized):
    chat_history = param.List([])
    answer = param.String("")
    db_query  = param.String("")
    db_response = param.List([])

    def __init__(self,  **params):
        super(cbfs, self).__init__( **params)
        self.panels = []
        self.loaded_file = "docs/MachineLearning-Lecture01.pdf"
        self.qa = load_db(self.loaded_file,"stuff", 4)

    def call_load_db(self, count):
        if count == 0 or file_input.value is None:  # init or no file specified :
          return pn.pane.Markdown(f"Loaded File: {self.loaded_file}")
        else:
            file_input.save("temp.pdf")  # local copy
            self.loaded_file = file_input.filename
            button_load.button_style="outline"
            self.qa = load_db("temp.pdf", "stuff", 4)
            button_load.button_style="solid"
        self.clr_history()
        return pn.pane.Markdown(f"Loaded File: {self.loaded_file}")

    def convchain(self, query):
        if not query:
            return pn.WidgetBox(pn.Row('User:', pn.pane.Markdown("", width=600)), scroll=True)
        result = self.qa.invoke({"question": query, "chat_history": self.chat_history})

        self.chat_history.extend([(query, result["answer"].page_content if isinstance(result["answer"], Document) else result["answer"])])
        self.db_query = result["generated_question"].page_content if isinstance(result["generated_question"], Document) else result["generated_question"]
        self.db_response = [doc.page_content if isinstance(doc, Document) else doc for doc in result["source_documents"]]
        self.answer = result['answer'].page_content if isinstance(result["answer"], Document) else result['answer']

        self.panels.extend([
            pn.Row('User:', pn.pane.Markdown(query, width=600)),
            pn.Row('ChatBot:', pn.pane.Markdown(self.answer, width=600, styles={'background-color': '#F6F6F6'}))
        ])
        inp.value = ''  #clears loading indicator when cleared
        return pn.WidgetBox(*self.panels,scroll=True)
```

11

PRACTICE

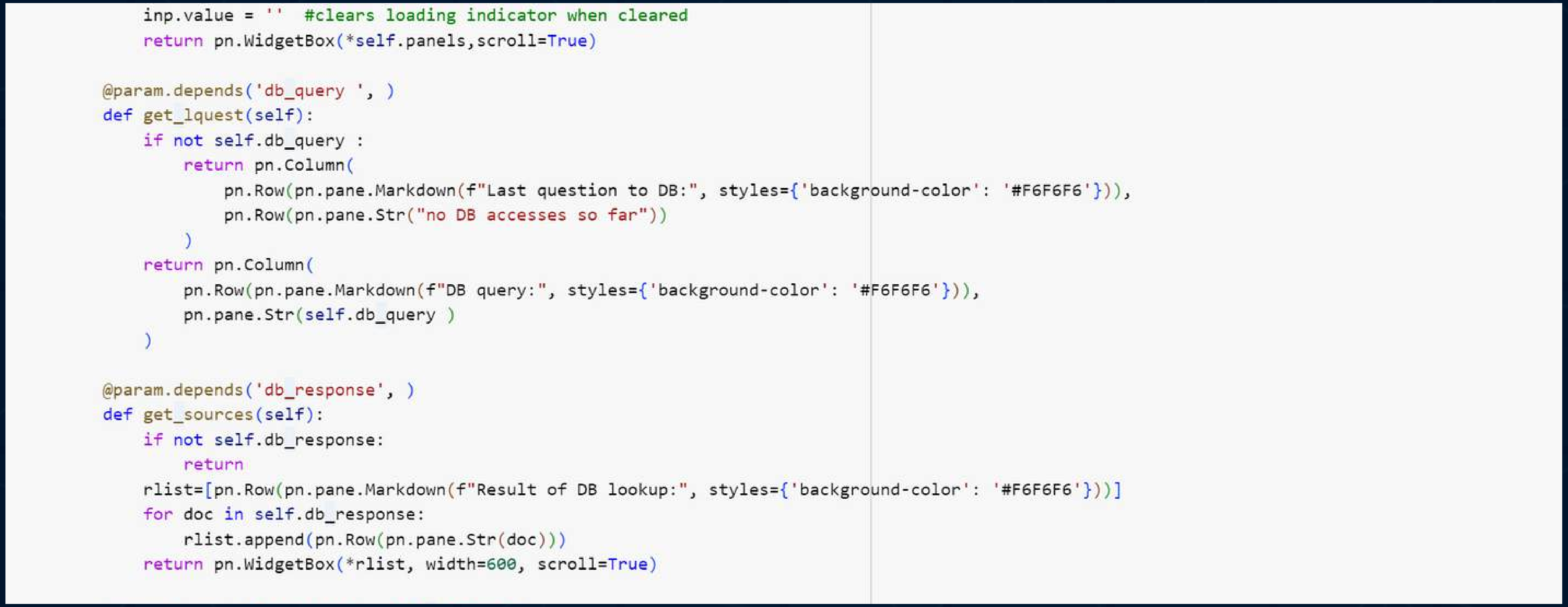
Create UI



```
        inp.value = ''  #clears loading indicator when cleared
        return pn.WidgetBox(*self.panels,scroll=True)

    @param.depends('db_query ', )
    def get_lquest(self):
        if not self.db_query :
            return pn.Column(
                pn.Row(pn.pane.Markdown(f"Last question to DB:", styles={'background-color': '#F6F6F6'})),
                pn.Row(pn.pane.Str("no DB accesses so far"))
            )
        return pn.Column(
            pn.Row(pn.pane.Markdown(f"DB query:", styles={'background-color': '#F6F6F6'})),
            pn.pane.Str(self.db_query )
        )
```

11 PRACTICE

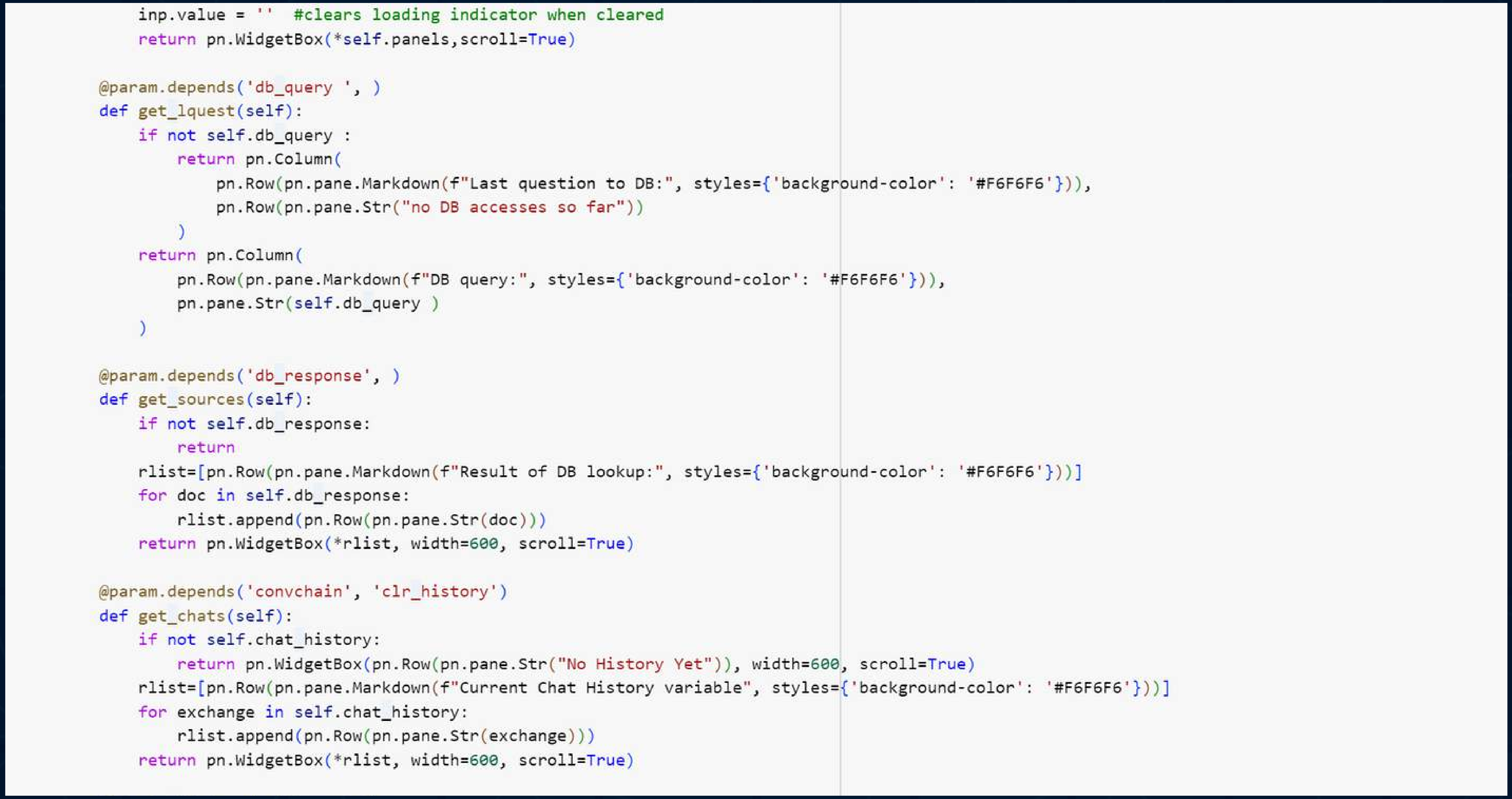
Create UI



```
        inp.value = ''  #clears loading indicator when cleared
        return pn.WidgetBox(*self.panels,scroll=True)

    @param.depends('db_query ', )
    def get_lquest(self):
        if not self.db_query :
            return pn.Column(
                pn.Row(pn.pane.Markdown(f"Last question to DB:", styles={'background-color': '#F6F6F6'})),
                pn.Row(pn.pane.Str("no DB accesses so far"))
            )
        return pn.Column(
            pn.Row(pn.pane.Markdown(f"DB query:", styles={'background-color': '#F6F6F6'})),
            pn.pane.Str(self.db_query )
        )

    @param.depends('db_response', )
    def get_sources(self):
        if not self.db_response:
            return
        rlist=[pn.Row(pn.pane.Markdown(f"Result of DB lookup:", styles={'background-color': '#F6F6F6'}))]
        for doc in self.db_response:
            rlist.append(pn.Row(pn.pane.Str(doc)))
        return pn.WidgetBox(*rlist, width=600, scroll=True)
```

11 PRACTICE

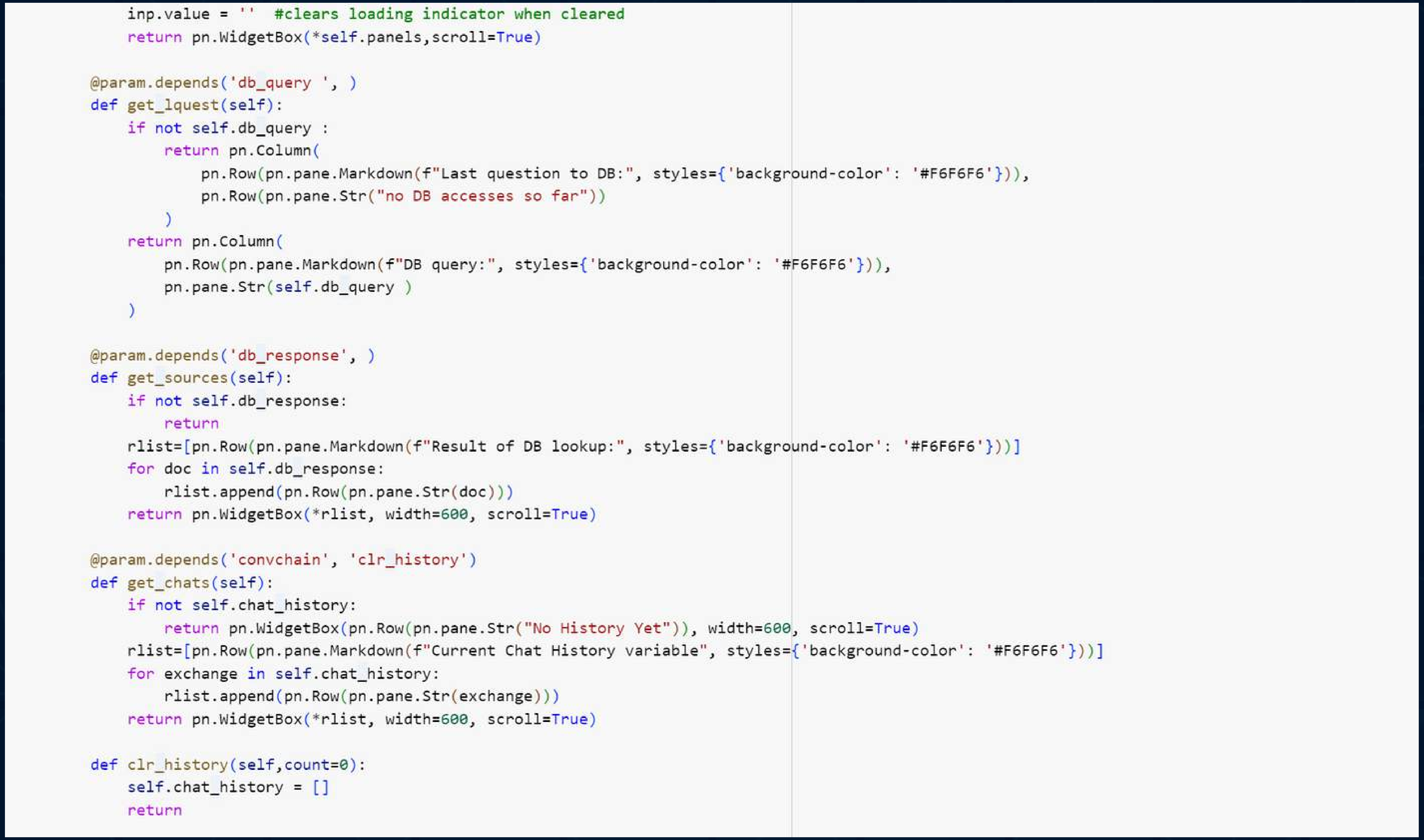
Create UI



```
        inp.value = ''  #clears loading indicator when cleared
        return pn.WidgetBox(*self.panels,scroll=True)

    @param.depends('db_query ', )
    def get_lquest(self):
        if not self.db_query :
            return pn.Column(
                pn.Row(pn.pane.Markdown(f"Last question to DB:", styles={'background-color': '#F6F6F6'})),
                pn.Row(pn.pane.Str("no DB accesses so far"))
            )
        return pn.Column(
            pn.Row(pn.pane.Markdown(f"DB query:", styles={'background-color': '#F6F6F6'})),
            pn.pane.Str(self.db_query )
        )

    @param.depends('db_response', )
    def get_sources(self):
        if not self.db_response:
            return
        rlist=[pn.Row(pn.pane.Markdown(f"Result of DB lookup:", styles={'background-color': '#F6F6F6'}))]
        for doc in self.db_response:
            rlist.append(pn.Row(pn.pane.Str(doc)))
        return pn.WidgetBox(*rlist, width=600, scroll=True)

    @param.depends('convchain', 'clr_history')
    def get_chats(self):
        if not self.chat_history:
            return pn.WidgetBox(pn.Row(pn.pane.Str("No History Yet")), width=600, scroll=True)
        rlist=[pn.Row(pn.pane.Markdown(f"Current Chat History variable", styles={'background-color': '#F6F6F6'}))]
        for exchange in self.chat_history:
            rlist.append(pn.Row(pn.pane.Str(exchange)))
        return pn.WidgetBox(*rlist, width=600, scroll=True)
```

11 PRACTICE

Create UI

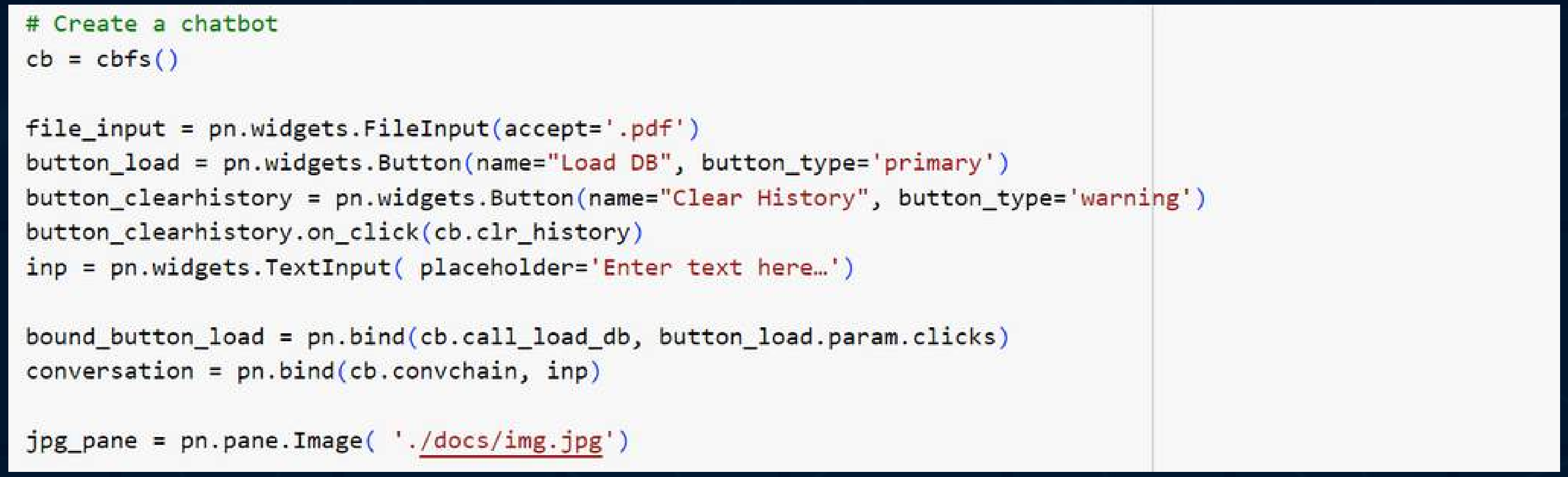


```
        inp.value = ''  #clears loading indicator when cleared
        return pn.WidgetBox(*self.panels,scroll=True)

    @param.depends('db_query ', )
    def get_lquest(self):
        if not self.db_query :
            return pn.Column(
                pn.Row(pn.pane.Markdown(f"Last question to DB:", styles={'background-color': '#F6F6F6'})),
                pn.Row(pn.pane.Str("no DB accesses so far"))
            )
        return pn.Column(
            pn.Row(pn.pane.Markdown(f"DB query:", styles={'background-color': '#F6F6F6'})),
            pn.pane.Str(self.db_query )
        )

    @param.depends('db_response', )
    def get_sources(self):
        if not self.db_response:
            return
        rlist=[pn.Row(pn.pane.Markdown(f"Result of DB lookup:", styles={'background-color': '#F6F6F6'}))]
        for doc in self.db_response:
            rlist.append(pn.Row(pn.pane.Str(doc)))
        return pn.WidgetBox(*rlist, width=600, scroll=True)

    @param.depends('convchain', 'clr_history')
    def get_chats(self):
        if not self.chat_history:
            return pn.WidgetBox(pn.Row(pn.pane.Str("No History Yet")), width=600, scroll=True)
        rlist=[pn.Row(pn.pane.Markdown(f"Current Chat History variable", styles={'background-color': '#F6F6F6'}))]
        for exchange in self.chat_history:
            rlist.append(pn.Row(pn.pane.Str(exchange)))
        return pn.WidgetBox(*rlist, width=600, scroll=True)

    def clr_history(self,count=0):
        self.chat_history = []
        return
```

11

PRACTICE

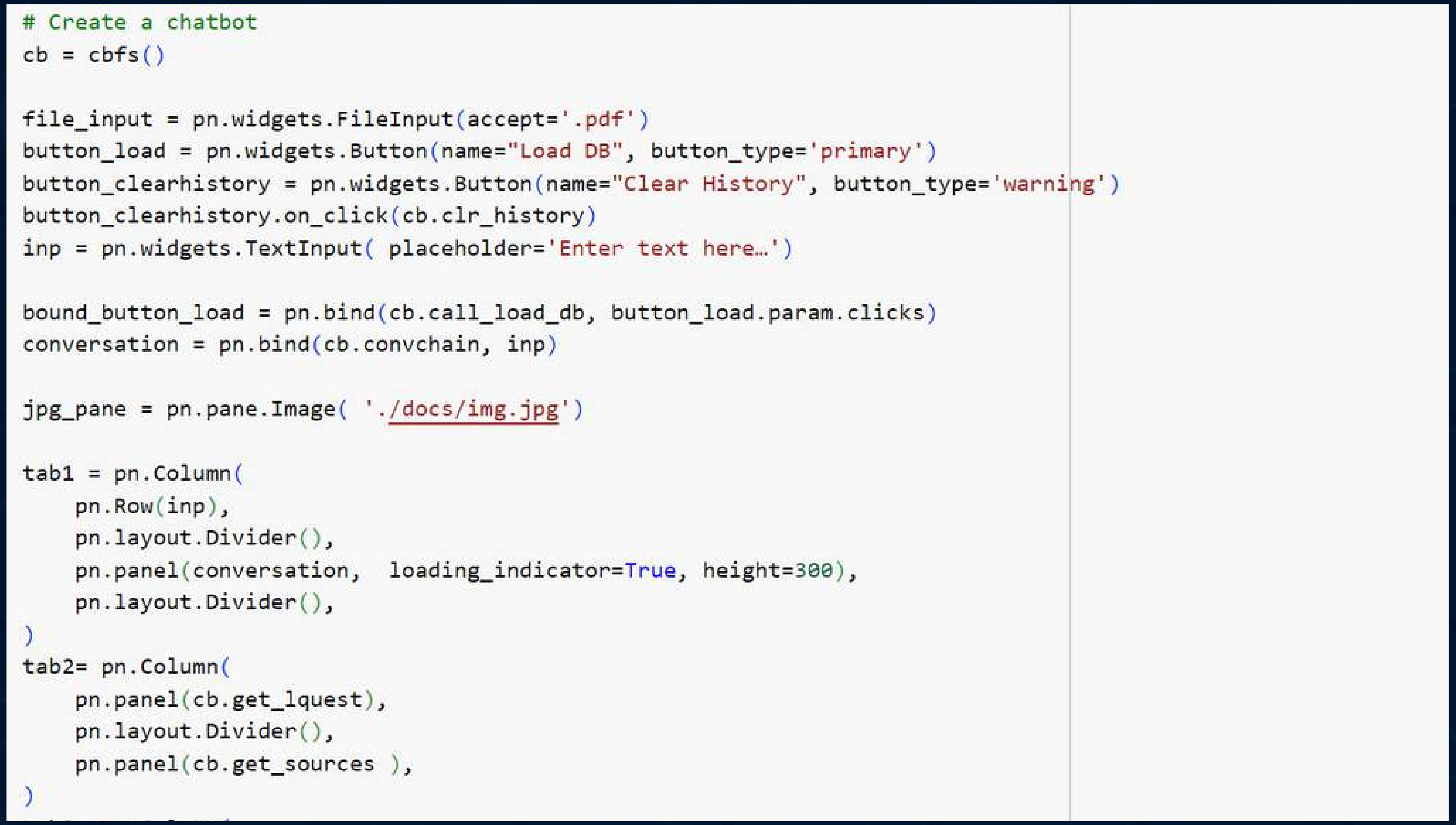
Create the ChatBot



```
# Create a chatbot
cb = cbfs()

file_input = pn.widgets.FileInput(accept='.pdf')
button_load = pn.widgets.Button(name="Load DB", button_type='primary')
button_clearhistory = pn.widgets.Button(name="Clear History", button_type='warning')
button_clearhistory.on_click(cb.clr_history)
inp = pn.widgets.TextInput( placeholder='Enter text here…')

bound_button_load = pn.bind(cb.call_load_db, button_load.param.clicks)
conversation = pn.bind(cb.convchain, inp)

jpg_pane = pn.pane.Image( './docs/img.jpg')
```

11 PRACTICE

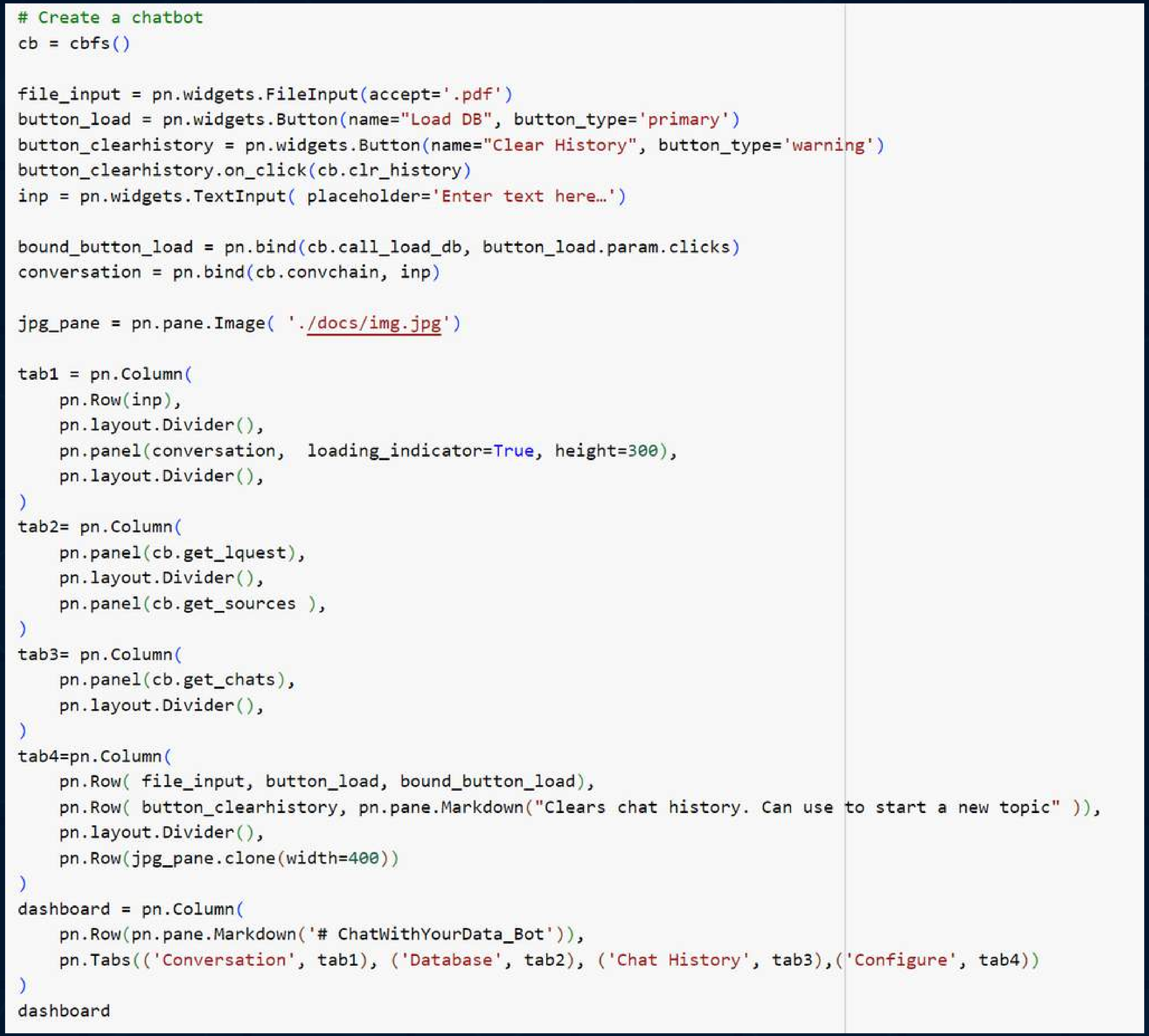
Create the ChatBot



```
# Create a chatbot
cb = cbfs()

file_input = pn.widgets.FileInput(accept='.pdf')
button_load = pn.widgets.Button(name="Load DB", button_type='primary')
button_clearhistory = pn.widgets.Button(name="Clear History", button_type='warning')
button_clearhistory.on_click(cb.clr_history)
inp = pn.widgets.TextInput( placeholder='Enter text here…')

bound_button_load = pn.bind(cb.call_load_db, button_load.param.clicks)
conversation = pn.bind(cb.convchain, inp)

jpg_pane = pn.pane.Image( './docs/img.jpg')

tab1 = pn.Column(
    pn.Row(inp),
    pn.layout.Divider(),
    pn.panel(conversation,  loading_indicator=True, height=300),
    pn.layout.Divider(),
)
tab2= pn.Column(
    pn.panel(cb.get_lquest),
    pn.layout.Divider(),
    pn.panel(cb.get_sources ),
)
```

11 PRACTICE

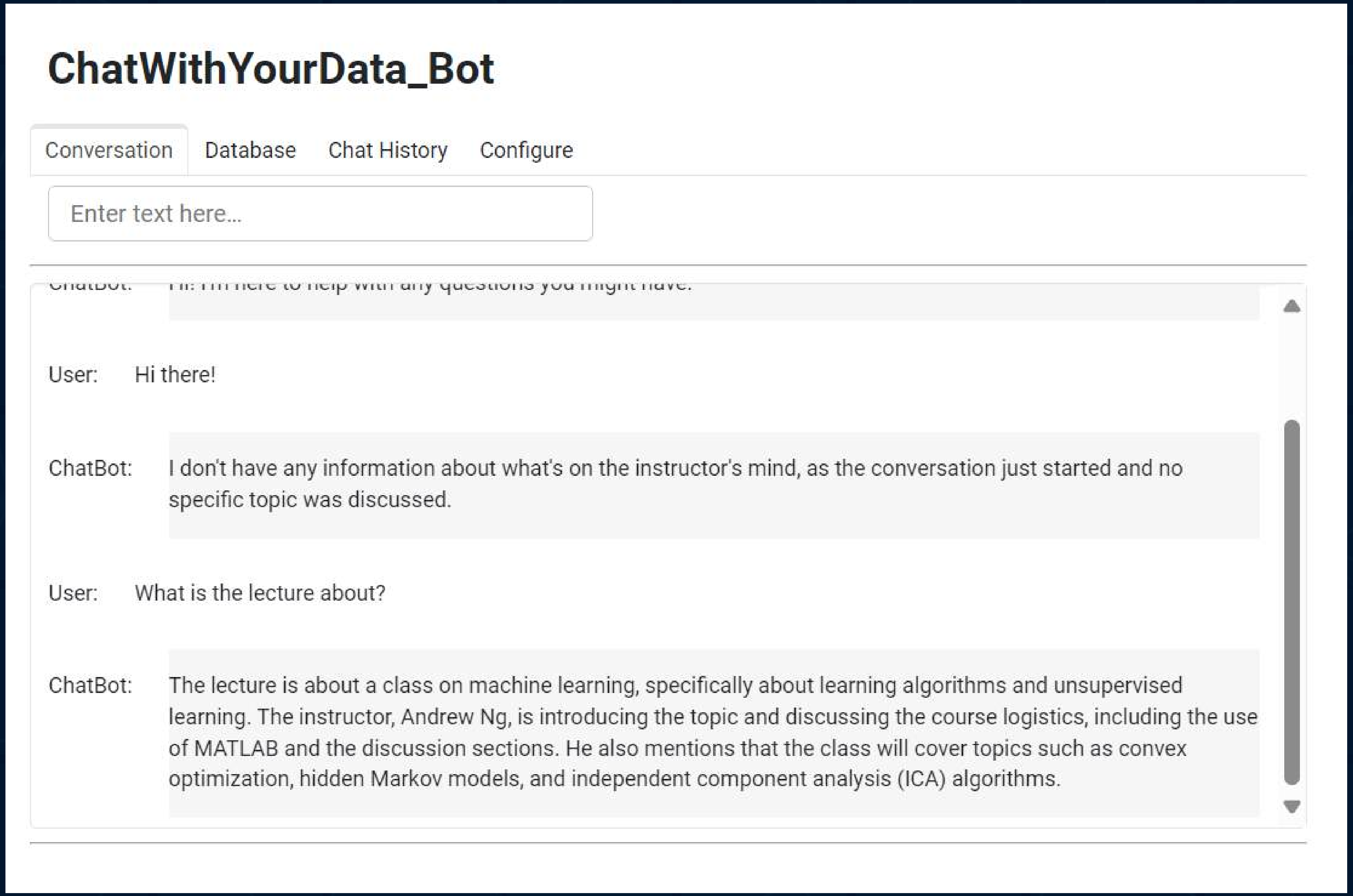
Create the ChatBot



```python
# Create a chatbot
cb = cbfs()

file_input = pn.widgets.FileInput(accept='.pdf')
button_load = pn.widgets.Button(name="Load DB", button_type='primary')
button_clearhistory = pn.widgets.Button(name="Clear History", button_type='warning')
button_clearhistory.on_click(cb.clr_history)
inp = pn.widgets.TextInput( placeholder='Enter text here…')

bound_button_load = pn.bind(cb.call_load_db, button_load.param.clicks)
conversation = pn.bind(cb.convchain, inp)

jpg_pane = pn.pane.Image( './docs/img.jpg')

tab1 = pn.Column(
    pn.Row(inp),
    pn.layout.Divider(),
    pn.panel(conversation,  loading_indicator=True, height=300),
    pn.layout.Divider(),
)
tab2= pn.Column(
    pn.panel(cb.get_lquest),
    pn.layout.Divider(),
    pn.panel(cb.get_sources ),
)
tab3= pn.Column(
    pn.panel(cb.get_chats),
    pn.layout.Divider(),
)
tab4=pn.Column(
    pn.Row( file_input, button_load, bound_button_load),
    pn.Row( button_clearhistory, pn.pane.Markdown("Clears chat history. Can use to start a new topic" )),
    pn.layout.Divider(),
    pn.Row(jpg_pane.clone(width=400))
)
dashboard = pn.Column(
    pn.Row(pn.pane.Markdown('# ChatWithYourData_Bot')),
    pn.Tabs(('Conversation', tab1), ('Database', tab2), ('Chat History', tab3),('Configure', tab4))
)
dashboard
```

11 PRACTICE

Enjoy converstions with your ChatBot!



ChatWithYourData_Bot

ChatBot: Hi! I'm here to help with any questions you might have.

User: Hi there!

ChatBot: I don't have any information about what's on the instructor's mind, as the conversation just started and no specific topic was discussed.

User: What is the lecture about?

ChatBot: The lecture is about a class on machine learning, specifically about learning algorithms and unsupervised learning. The instructor, Andrew Ng, is introducing the topic and discussing the course logistics, including the use of MATLAB and the discussion sections. He also mentions that the class will cover topics such as convex optimization, hidden Markov models, and independent component analysis (ICA) algorithms.

Thank you!

Have questions? Let's Connect!