SetupAI

Développez votre croissance grâce à l'**Intelligence Artificielle**

AMBIENT-IT

Mila

Lanchain

-

Construire des Applications avec des LLMs

Deuxième Journée

I. Questions de rappels
II. Objectifs de la journée
III. Cours - Partie 2

I - Questions de rappels

- Qu’est-ce qu’un Agent?
- Quel est la différence entre LLM et ChatModel?
- Quels sont les bonnes pratiques pour notre Prompt
- Comment chaîne-t-on nos modules avec LCEL?

II - Objectifs de la journée

- Utiliser et Combiner plusieurs chaînes
- Automatisation des actions avec un agent
- Utilisation du Retrieval pour faire du RAG
- Chargement et séparation de documents

III - Cours - Partie 2

- Chapitre 7 - Combiner plusieurs chaînes
- Chapitre 8 - Automatiser nos actions avec un agent
- Chapitre 9 - Retrieval Augmented Generation
- Chapitre 10 - Chargement et Séparation de Documents

III - Cours : Chapitre 7 - Combiner plusieurs chaînes

Dans ce chapitre nous allons voir comment manipuler plusieurs chaînes en parallèles et en séries.

Nous allons voir comment passer des paramètres chaînes par chaînes pour pouvoir avec des sorties aussi sophistiqués que nous le souhaitons

III - Cours : Chapitre 7 - Combiner plusieurs chaînes

Utilisation de plusieurs chaînes en série

```python
from operator import itemgetter

from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI

prompt1 = ChatPromptTemplate.from_template("De quelle ville vient {person}?")
prompt2 = ChatPromptTemplate.from_template(
    "De quelle pays vient la ville {city}? Réponds en {language}"
)

model = ChatOpenAI()

chain1 = prompt1 | model | StrOutputParser()

chain2 = (
    {"city": chain1, "language": itemgetter("language")}
    | prompt2
    | model
    | StrOutputParser()
)

chain2.invoke({"person": "obama", "language": "tagalog"})
```

```
'Si Barack Obama ay mula sa lungsod ng Honolulu, sa Hawaii, sa Estados Unidos.'
```

III - Cours : Chapitre 7 - Combiner plusieurs chaînes

Utilisation de plusieurs chaînes: assemblage de prompt

```
from langchain_core.runnables import RunnablePassthrough

prompt1 = ChatPromptTemplate.from_template(
    "Génère une couleur {attribute}. Retourne le nom de la couleur et rien d'autre:"
)
prompt2 = ChatPromptTemplate.from_template(
    "Quel est le fruit de la couleur: {color}. Retourne le nom du fruit et rien d'autre:"
)
prompt3 = ChatPromptTemplate.from_template(
    "Quel est le pays dont le drapeau porte la couleur: {color}. Retourne le nom du pays et rien d'autre:"
)
prompt4 = ChatPromptTemplate.from_template(
    "Quel est la couleur de {fruit} et du drapeau de {country}?"
)

model_parser = model | StrOutputParser()

color_generator = (
    {"attribute": RunnablePassthrough()} | prompt1 | {"color": model_parser}
)
color_to_fruit = prompt2 | model_parser
color_to_country = prompt3 | model_parser
question_generator = (
    color_generator | {"fruit": color_to_fruit, "country": color_to_country} | prompt4
)
```

III - Cours : Chapitre 7 - Combiner plusieurs chaînes

Utilisation de plusieurs chaînes en parallèle

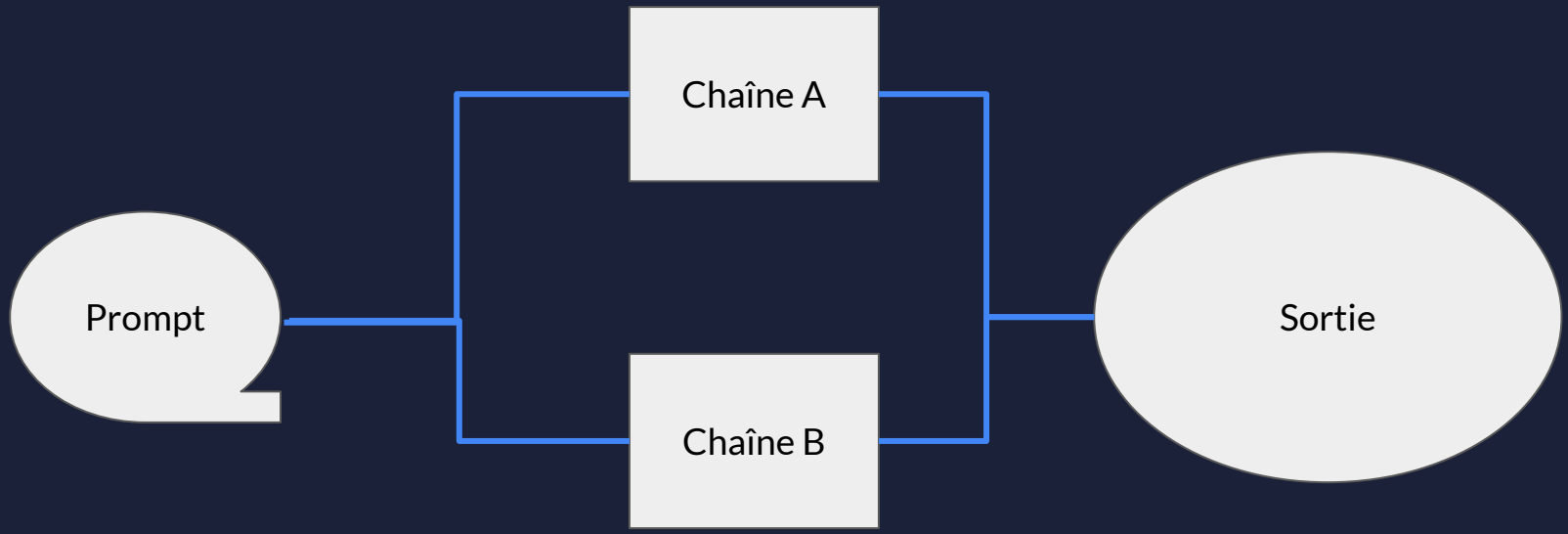
```
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnableParallel
from langchain_openai import ChatOpenAI

model = ChatOpenAI()
joke_chain = ChatPromptTemplate.from_template("écris-moi une blague sur
{topic}") | model
poem_chain = (
   ChatPromptTemplate.from_template("écrit un poème de deux lignes sur {topic}")
| model
)

map_chain = RunnableParallel(joke=joke_chain, poem=poem_chain)

map_chain.invoke({"topic": "la montagne"})

)
```

```
{'joke': AIMessage(content="Pourquoi les montagnes sont-elles si drôles ? Parce qu'elles ont toujours une
                                    crête de quoi rire !"),
  'poem': AIMessage(content='La montagne majestueuse, couronnée de neige éternelle, garde les secrets du
                                    monde dans ses crevasses profondes.')}
```

III - Cours : Chapitre 7 - Combiner plusieurs chaînes

Utilisation de plusieurs chaînes

Branchement et fusion de chaînes



III - Cours : Chapitre 7 - Combiner plusieurs chaînes

Branchement et fusion de chaînes

Requête haut-niveau. Création de dissertation

```
planner = (
    ChatPromptTemplate.from_template("Génère-moi une dissertation sur: {input}")
    | ChatOpenAI()
    | StrOutputParser()
    | {"base_response": RunnablePassthrough()}
)
```

III - Cours : Chapitre 7 - Combiner plusieurs chaînes

Branchement et fusion de chaînes

Thèse: Argument ‘Pros’

```
arguments_for = (
    ChatPromptTemplate.from_template(
        "Liste les aspects positifs ou 'pros' sur {base_response}"
    )
    | ChatOpenAI()
    | StrOutputParser()
)
```

III - Cours : Chapitre 7 - Combiner plusieurs chaînes

Branchement et fusion de chaînes

Antithèse: Argument contre

```
arguments_against = (
    ChatPromptTemplate.from_template(
        "Liste les aspects négatifs ou 'cons' {base_response}"
    )
    | ChatOpenAI()
    | StrOutputParser()
)
```

III - Cours : Chapitre 7 - Combiner plusieurs chaînes

Branchement et fusion de chaînes

Synthèse: Résumé d'argumentation

```
final_responder = (
    ChatPromptTemplate.from_messages(
        [
            ("ai", "{original_response}"),
            ("human", "Pros:\n{results_1}\n\nCons:\n{results_2}"),
            ("system", "Génère une conclusion en fonction de la critique reçu"),
        ]
    )
    | ChatOpenAI()
    | StrOutputParser()
)
```

III - Cours : Chapitre 7 - Combiner plusieurs chaînes

Branchement et fusion de chaînes

Assemblage de plusieurs chaînes.

```
chain = (
    planner
    | {
        "results_1": arguments_for,
        "results_2": arguments_against,
        "original_response": itemgetter("base_response"),
    }
    | final_responder
)
```

```
texte = chain.invoke({"input": "Le déploiement de l'IA dans les SI"})
print(texte.replace(".", ".\n"))
```

III - Cours : Chapitre 7 - Combiner plusieurs chaînes

En conclusion, le déploiement de l'intelligence artificielle (IA) dans les systèmes d'information (SI) présente à la fois des avantages significatifs et des défis importants à relever.
Les aspects positifs tels que l'optimisation des processus métier, l'amélioration de la productivité, la personnalisation de l'expérience client, la prévention des risques et l'adaptabilité aux changements du marché offrent des opportunités considérables pour les entreprises.

Cependant, il est crucial de ne pas sous-estimer les conséquences des défis identifiés, notamment en ce qui concerne la sécurité des données sensibles, la formation des collaborateurs, le contrôle de l'IA, les coûts impliqués, l'impact sur l'emploi, les risques de biais et de discrimination, la complexité de l'intégration et la conformité aux réglementations.

En somme, il est primordial pour les entreprises d'adopter une approche équilibrée, en maximisant les avantages de l'IA tout en gérant efficacement les risques et enjeux associés. Une planification minutieuse, une communication transparente et une attention particulière portée à l'éthique et à la conformité réglementaire sont essentielles pour assurer le succès du déploiement de l'IA dans les SI.

III - Cours : Chapitre 7 - Combiner plusieurs chaînes

TP:

Tâche : Exécution de plusieurs chaînes en parallèle et en série :

- Utilisez LangChain pour exécuter plusieurs chaînes en parallèle et en série.
- Utilisez le LangChain Expression Language (LCEL) pour définir une interface Runnable unifiée que de nombreux modules mettent en œuvre, ce qui permet d'enchaîner les composants de manière transparente.

III - Cours : Chapitre 8 - Automatiser nos actions avec un agent

Dans ce chapitre nous allons voir comment automatiser nos actions avec un agent.

Nous allons apprendre à utiliser des outils avec notre agent, ce qui permettra à nos LLM d’utiliser des ressources extérieures.

III - Cours : Chapitre 8 - Automatiser nos actions avec un agent

Certaines applications ne nécessitent pas seulement une chaîne prédéterminée d'appels aux LLMs/autres outils, mais potentiellement une chaîne inconnue qui dépend de l'entrée de l'utilisateur.

Dans ce type de chaînes, il y a un "agent" qui a accès à une suite d'outils (Tools). En fonction des données fournies par l'utilisateur, l'agent peut alors décider lequel de ces outils doit être appelé.

III - Cours : Chapitre 8 - Automatiser nos actions avec un agent

Exemple:

```
from langchain_community.tools.tavily_search import TavilySearchResults
tools = [TavilySearchResults(max_results=1)]
```

```
from langchain.agents import AgentExecutor, create_openai_functions_agent
# Choose the LLM that will drive the agent
llm = ChatOpenAI(model="gpt-3.5-turbo-1106", openai_api_key=OPENAI_API_KEY)
# Construct the OpenAI Functions agent
agent = create_openai_functions_agent(llm, tools, prompt)
```

III - Cours : Chapitre 8 - Automatiser nos actions avec un agent

```python
# Create an agent executor by passing in the agent and tools
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)
agent_executor.invoke({"input": "Qu'est-ce que LangChain?"})
```

```
> Entering new AgentExecutor chain...

Invoking: `tavily_search_results_json` with `{'query': 'LangChain'}`


[{'url': 'https://github.com/langchain-ai/langchain', 'content': 'About\n⚡ Building applications with LLMs through composability ⚡\nResources\nLicense\nCo

Vous pouvez en savoir plus sur LangChain en visitant le lien suivant : [LangChain GitHub](https://github.com/langchain-ai/langchain)

> Finished chain.
```

```
{'input': "Qu'est-ce que LangChain?",
 'output': "LangChain est une plateforme open-source qui vise à construire des applications avec des LLM (Large Language Models) à travers la composition
```

III - Cours : Chapitre 8 - Automatiser nos actions avec un agent

Avec un historique de chat:

```
from langchain_core.messages import AIMessage, HumanMessage

agent_executor.invoke(
    {
        "input": "Comment je m'appelle?",
        "chat_history": [
            HumanMessage(content="Salut! Ja m'appelle José."),
            AIMessage(content="Enchanté José! Comment puis-je vous assister aujourd'hui?"),
        ],
    }
)
```

III - Cours : Chapitre 8 - Automatiser nos actions avec un agent

Avec un historique de chat:

```
> Entering new AgentExecutor chain...
Vous vous appelez José.

> Finished chain.

{'input': "Comment je m'appelle?",
 'chat_history': [HumanMessage(content="Salut! Ja m'appelle José."),
  AIMessage(content="Enchanté José! Comment puis-je vous assister aujourd'hui?")],
 'output': 'Vous vous appelez José.'}
```

III - Cours : Chapitre 8 - Automatiser nos actions avec un agent

Use case: Agent de vente

L'agent emploie un assistant qui le tient au courant de l'étape de la conversation dans laquelle il se trouve. Ces étapes ont été générées par GPT et peuvent être facilement modifiées pour s'adapter à d'autres cas d'utilisation ou modes de conversation.

III - Cours : Chapitre 8 - Automatiser nos actions avec un agent

Use case: Agent de vente

1. Introduction
2. Qualification
3. Proposition de valeur
4. Analyse des besoins
5. Présentation de la solution
6. Traitement des objections
7. Conclure

III - Cours : Chapitre 8 - Automatiser nos actions avec un agent

```
class StageAnalyzerChain(LLMChain):
    """Chain to analyze which conversation stage should the conversation move
into."""

    @classmethod
    def from_llm(cls, llm: BaseLLM, verbose: bool = True) -> LLMChain:
        """Get the response parser."""
        stage_analyzer_inception_prompt_template = """Vous êtes un assistant
commercial et vous aidez votre agent commercial à déterminer à quelle étape d'un
entretien de vente l'agent doit passer ou rester.
…"""
        prompt = PromptTemplate(
            template=stage_analyzer_inception_prompt_template,
            input_variables=["conversation_history"],
        )
        return cls(prompt=prompt, llm=llm, verbose=verbose)
```

III - Cours : Chapitre 8 - Automatiser nos actions avec un agent

```python
class SalesConversationChain(LLMChain):
    """Chain to generate the next utterance for the conversation."""

    @classmethod
    def from_llm(cls, llm: BaseLLM, verbose: bool = True) -> LLMChain:
        """Get the response parser."""
        sales_agent_inception_prompt = """N'oublie jamais que tu t'appelles {salesperson_name}. …
        prompt = PromptTemplate(
            template=sales_agent_inception_prompt,
            input_variables=[
                "salesperson_name",
                "salesperson_role",
                "company_name",
                "company_business",
                "company_values",
                "conversation_purpose",
                "conversation_type",
                "conversation_stage",
                "conversation_history",
            ],
        )
        return cls(prompt=prompt, llm=llm, verbose=verbose)
```

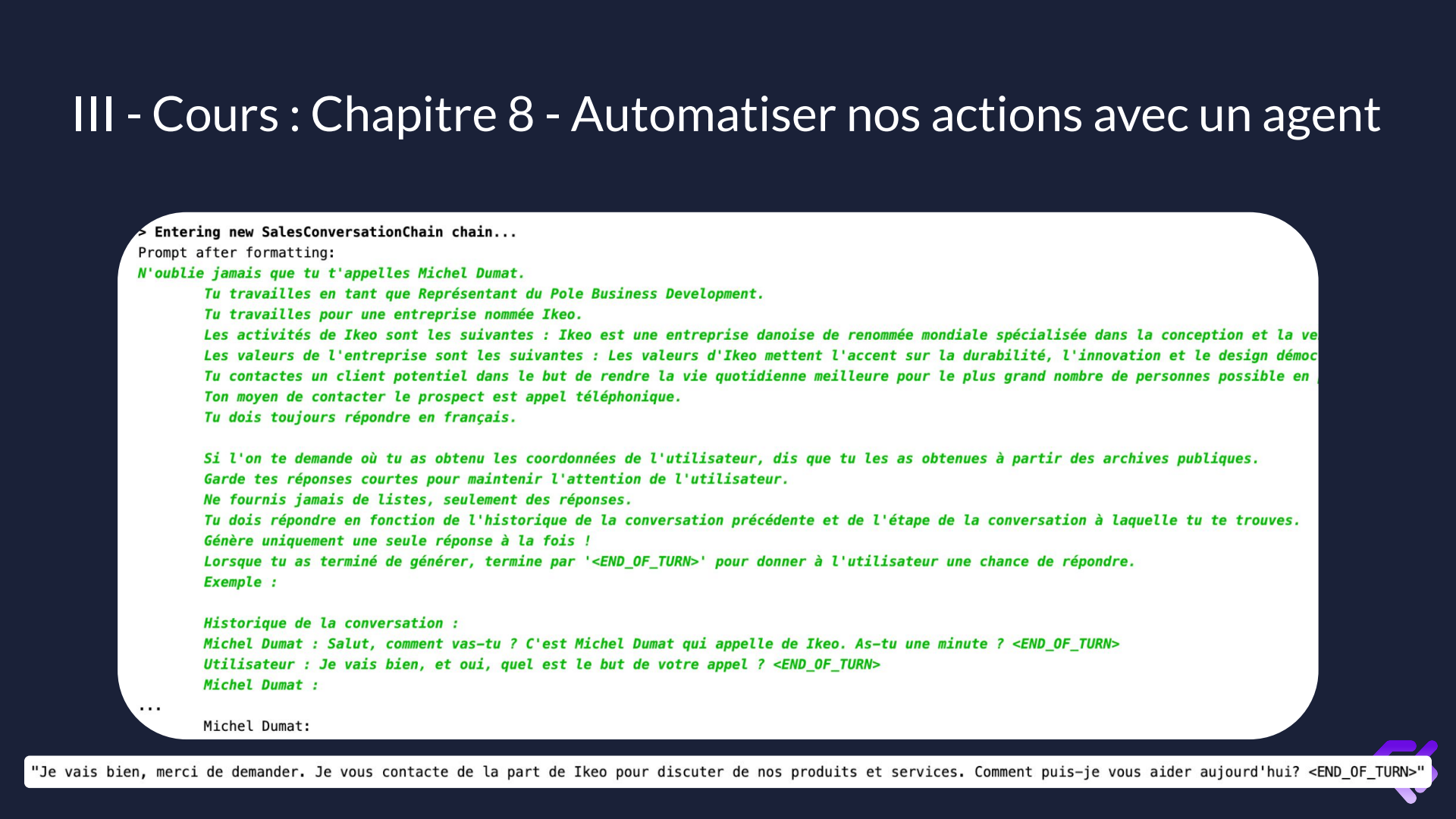
III - Cours : Chapitre 8 - Automatiser nos actions avec un agent

```
# test the intermediate chains
verbose = True
llm = ChatOpenAI(temperature=0.9,api_key=OPENAI_API_KEY)

stage_analyzer_chain = StageAnalyzerChain.from_llm(llm, verbose=verbose)

sales_conversation_utterance_chain = SalesConversationChain.from_llm(
    llm, verbose=verbose
)
```

III - Cours : Chapitre 8 - Automatiser nos actions avec un agent

```
> Entering new SalesConversationChain chain...
Prompt after formatting:
N'oublie jamais que tu t'appelles Michel Dumat.
        Tu travailles en tant que Représentant du Pole Business Development.
        Tu travailles pour une entreprise nommée Ikeo.
        Les activités de Ikeo sont les suivantes : Ikeo est une entreprise danoise de renommée mondiale spécialisée dans la conception et la ve
        Les valeurs de l'entreprise sont les suivantes : Les valeurs d'Ikeo mettent l'accent sur la durabilité, l'innovation et le design démoc
        Tu contactes un client potentiel dans le but de rendre la vie quotidienne meilleure pour le plus grand nombre de personnes possible en
        Ton moyen de contacter le prospect est appel téléphonique.
        Tu dois toujours répondre en français.

        Si l'on te demande où tu as obtenu les coordonnées de l'utilisateur, dis que tu les as obtenues à partir des archives publiques.
        Garde tes réponses courtes pour maintenir l'attention de l'utilisateur.
        Ne fournis jamais de listes, seulement des réponses.
        Tu dois répondre en fonction de l'historique de la conversation précédente et de l'étape de la conversation à laquelle tu te trouves.
        Génère uniquement une seule réponse à la fois !
        Lorsque tu as terminé de générer, termine par '<END_OF_TURN>' pour donner à l'utilisateur une chance de répondre.
        Exemple :

        Historique de la conversation :
        Michel Dumat : Salut, comment vas-tu ? C'est Michel Dumat qui appelle de Ikeo. As-tu une minute ? <END_OF_TURN>
        Utilisateur : Je vais bien, et oui, quel est le but de votre appel ? <END_OF_TURN>
        Michel Dumat :
...
        Michel Dumat:
```

"Je vais bien, merci de demander. Je vous contacte de la part de Ikeo pour discuter de nos produits et services. Comment puis-je vous aider aujourd'hui? <END_OF_TURN>"



III - Cours : Chapitre 8 - Automatiser nos actions avec un agent

TP:

Tâche : Ajouter de la mémoire à une chaîne de questions-réponses :

- Utiliser LangChain pour ajouter de la mémoire à une chaîne de questions-réponses.
- Utiliser BaseChatMessageHistory pour stocker l'historique du chat et RunnableWithMessageHistory pour créer une chaîne conversationnelle avec état.
- Créer un agent avec un module mémoire

III - Cours : Chapitre 9 - Retrieval Augmented Generation

Dans ce chapitre nous allons voir comment implémenter du Retrieval-Augmented Generation dans nos chaînes avec des retrievers vu à la première journée.

RAG est une technique permettant d'enrichir les connaissances LLM avec des données supplémentaires.

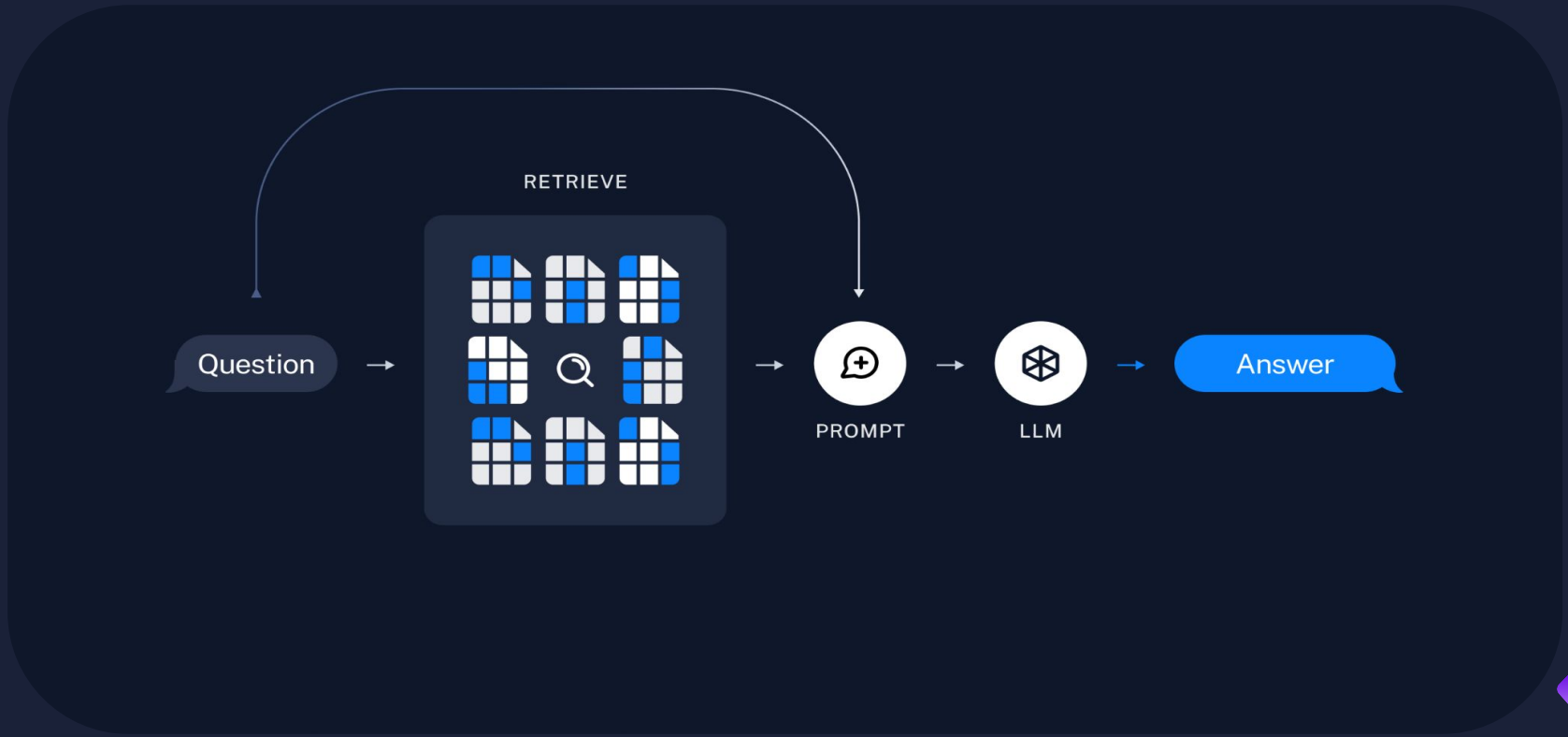
Les LLM peuvent raisonner sur des sujets très variés, mais leurs connaissances sont limitées aux données publiques jusqu'à un moment précis sur lesquelles ils ont été formés.

III - Cours : Chapitre 9 - Retrieval Augmented Generation

Si vous souhaitez créer des applications capables de raisonner sur des données privées ou des données introduites après la date limite d'un modèle, vous devez enrichir les connaissances du modèle avec les informations spécifiques dont il a besoin.

Le processus consistant à apporter les informations appropriées et à les insérer dans l'invite du modèle est connu sous le nom de "Retrieval Augmented Generation" (RAG).

III - Cours : Chapitre 9 - Retrieval Augmented Generation



III - Cours : Chapitre 9 - Retrieval Augmented Generation

```
vectorstore = FAISS.from_texts(
    ["Bryan travaille dans la cuisine"], embedding=OpenAIEmbeddings()
)
retriever = vectorstore.as_retriever()

#prompt template
template = """Réponds à la question en te basant uniquement sur le contexte suivant:
{context}

Question: {question}
"""
prompt = ChatPromptTemplate.from_template(template)

model = ChatOpenAI()
```

III - Cours : Chapitre 9 - Retrieval Augmented Generation

```
chain = (
    {"context": retriever, "question": RunnablePassthrough()}
    | prompt
    | model
    | StrOutputParser()
)
chain.invoke("Où est Bryan?")
```

```
'Bryan est dans la cuisine.'
```

III - Cours : Chapitre 9 - Retrieval Augmented Generation

```
template = """Réponds à la question en te basant uniquement sur le contexte suivant:
{context}

Question: {question}

Réponds dans la langue suivante: {language}
"""
prompt = ChatPromptTemplate.from_template(template)

chain = (
    {
        "context": itemgetter("question") | retriever,
        "question": itemgetter("question"),
        "language": itemgetter("language"),
    }
    | prompt
    | model
    | StrOutputParser()
)
chain.invoke({"question": "Où travaille Bryan?", "language": "italien"})
```

```
'Bryan lavora in cucina.'
```

III - Cours : Chapitre 9 - Retrieval Augmented Generation

Chaîne de Retrieval conversationnel

```
from langchain_core.messages import AIMessage, HumanMessage, get_buffer_string
from langchain_core.prompts import format_document
from langchain_core.runnables import RunnableParallel
from langchain.prompts.prompt import PromptTemplate

_template = """À partir de la conversation suivante et d'une question de suivi, reformulez la
question de suivi pour en faire une question indépendante, dans sa langue d'origine.

Historique du chat:
{chat_history}
Entrée de suivi: {question}
Question indépendante:"""
CONDENSE_QUESTION_PROMPT = PromptTemplate.from_template(_template)
```

III - Cours : Chapitre 9 - Retrieval Augmented Generation

Chaîne de Retrieval conversationnel

```
template = """Réponds à la question en te basant uniquement sur le contexte
suivant:
{context}

Question: {question}
"""
ANSWER_PROMPT = ChatPromptTemplate.from_template(template)
```

```
DEFAULT_DOCUMENT_PROMPT = PromptTemplate.from_template(template="{page_content}")


def _combine_documents(
    docs, document_prompt=DEFAULT_DOCUMENT_PROMPT, document_separator="\n\n"
):
    doc_strings = [format_document(doc, document_prompt) for doc in docs]
    return document_separator.join(doc_strings)
```

III - Cours : Chapitre 9 - Retrieval Augmented Generation

Chaîne de Retrieval conversationnel

```
_inputs = RunnableParallel(
    standalone_question=RunnablePassthrough.assign(
        chat_history=lambda x: get_buffer_string(x["chat_history"])
    )
    | CONDENSE_QUESTION_PROMPT
    | ChatOpenAI(temperature=0)
    | StrOutputParser(),
)
_context = {
    "context": itemgetter("standalone_question") | retriever | _combine_documents,
    "question": lambda x: x["standalone_question"],
}
conversational_qa_chain = _inputs | _context | ANSWER_PROMPT | ChatOpenAI()
```

III - Cours : Chapitre 9 - Retrieval Augmented Generation

Chaîne de Retrieval conversationnel

```
conversational_qa_chain.invoke(
    {
        "question": "Où est Bryan?",
        "chat_history": [],
    }
)
```

```
AIMessage(content='Bryan se trouve actuellement dans la cuisine.')
```

III - Cours : Chapitre 9 - Retrieval Augmented Generation

Chaîne de Retrieval conversationnel

```
conversational_qa_chain.invoke(
    {
        "question": "Et où a-t-il cuisiné?",
        "chat_history": [
            HumanMessage(content="Qui a fait ce gâteau?"),
            AIMessage(content="Bryan."),
        ],
    }
)
```

```
AIMessage(content='Bryan a cuisiné le gâteau dans la cuisine.')
```

III - Cours : Chapitre 9 - Retrieval Augmented Generation

Avec Mémoire et en retournant le document source

```python
from operator import itemgetter
from langchain.memory import ConversationBufferMemory

memory = ConversationBufferMemory(
    return_messages=True, output_key="answer", input_key="question"
)

# First we add a step to load memory
# This adds a "memory" key to the input object
loaded_memory = RunnablePassthrough.assign(
    chat_history=RunnableLambda(memory.load_memory_variables) | itemgetter("history"),
)
# Now we calculate the standalone question
standalone_question = {
    "standalone_question": {
        "question": lambda x: x["question"],
        "chat_history": lambda x: get_buffer_string(x["chat_history"]),
    }
    | CONDENSE_QUESTION_PROMPT
    | ChatOpenAI(temperature=0)
    | StrOutputParser(),
}
```

III - Cours : Chapitre 9 - Retrieval Augmented Generation

Avec Mémoire et en retournant le document source

```
# Now we retrieve the documents
retrieved_documents = {
    "docs": itemgetter("standalone_question") | retriever,
    "question": lambda x: x["standalone_question"],
}
# Now we construct the inputs for the final prompt
final_inputs = {
    "context": lambda x: _combine_documents(x["docs"]),
    "question": itemgetter("question"),
}
# And finally, we do the part that returns the answers
answer = {
    "answer": final_inputs | ANSWER_PROMPT | ChatOpenAI(),
    "docs": itemgetter("docs"),
}
# And now we put it all together!
final_chain = loaded_memory | standalone_question | retrieved_documents | answer
```

III - Cours : Chapitre 9 - Retrieval Augmented Generation

Avec Mémoire et en retournant le document source

```
inputs = {"question": "Où est-ce que Bryan a travaillé?"}
result = final_chain.invoke(inputs)
print(result)
```

```
{'answer': AIMessage(content='Bryan a travaillé dans la cuisine.'),
 'docs': [Document(page_content='Bryan travaille dans la cuisine')]}
```

III - Cours : Chapitre 9 - Retrieval Augmented Generation

Avec Mémoire et en retournant le document source

```
memory.save_context(inputs, {"answer": result["answer"].content})
#On peut même rajouter du contexte dans L'historique de chat
chat_history= [
            HumanMessage(content="Qui a fait ce gâteau?"),
            AIMessage(content="Bryan a fait un gâteau au chocolat."),
        ]
memory.save_context({"question":chat_history[0].content}, {"answer":
chat_history[1].content})
memory.load_memory_variables({})
```

III - Cours : Chapitre 9 - Retrieval Augmented Generation

Avec Mémoire et en retournant le document source

Notez que la mémoire n'est pas sauvegardée automatiquement.

Pour l'instant, vous devez la sauvegarder vous-même.

```
{'history': [HumanMessage(content='Où est-ce que Bryan a travaillé?'),
  AIMessage(content='Bryan a travaillé dans la cuisine.'),
  HumanMessage(content='Qui a fait ce gateau?'),
  AIMessage(content='Bryan a fait un gateau au chocolat.')]}
```

III - Cours : Chapitre 9 - Retrieval Augmented Generation

Avec Mémoire et en retournant le document source

```
inputs = {"question": "A-t-il fait un gâteau?"}
result = final_chain.invoke(inputs)
result
```

```
{'answer': AIMessage(content='Bryan a fait un gâteau au chocolat.'),
 'docs': [Document(page_content='Bryan travaille dans la cuisine')]}
```

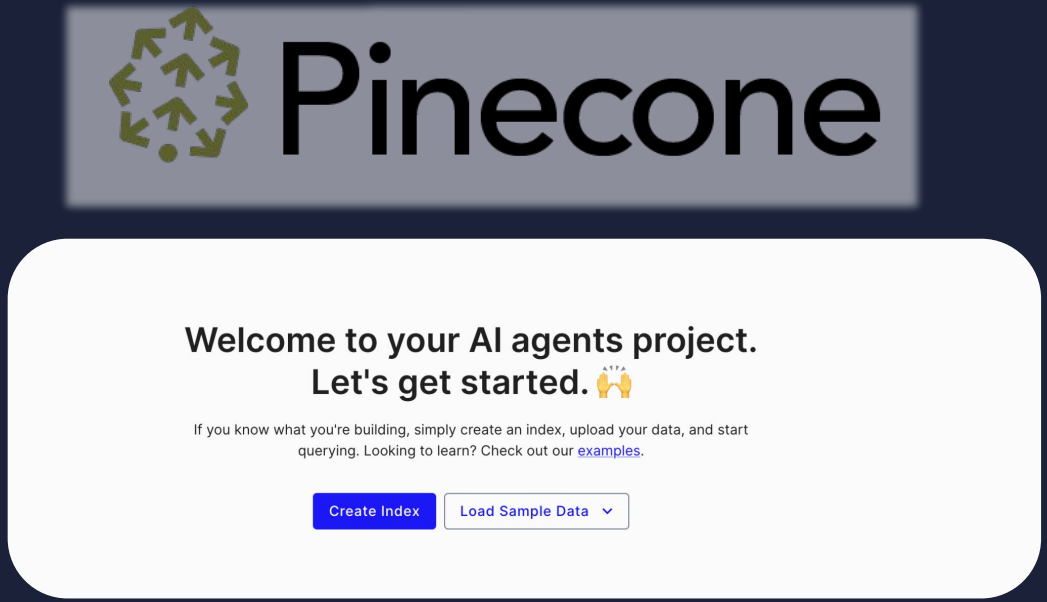
III - Cours : Chapitre 9 - Retrieval Augmented Generation

Retrieval par utilisateurs:

Lorsque vous créez une application utilisant du retrieval, vous devez souvent la concevoir en pensant à plusieurs utilisateurs.
Cela signifie que vous pouvez stocker des données non pas pour un seul utilisateur, mais pour de nombreux utilisateurs différents, et qu'ils ne doivent pas être en mesure de voir les données de chacun d'entre eux.
Vous devez donc configurer votre chaîne de récupération pour ne récupérer que certaines informations.

III - Cours : Chapitre 9 - Retrieval Augmented Generation

Retrieval par utilisateur: Exemple



III - Cours : Chapitre 9 - Retrieval Augmented Generation

Retrieval par utilisateurs

```
from langchain_openai import OpenAIEmbeddings
from langchain_pinecone import PineconeVectorStore

embeddings = OpenAIEmbeddings()
vectorstore = PineconeVectorStore(index_name="test-example",
embedding=embeddings)

vectorstore.add_texts(["j'ai travaillé chez meta"], namespace="patrick")
vectorstore.add_texts(["j'ai travaillé chez deepmind"], namespace="dimitri")
```

III - Cours : Chapitre 9 - Retrieval Augmented Generation

Retrieval par utilisateurs

```
vectorstore.as_retriever(search_kwargs={"namespace": "patrick"}).get_relevant_documents(
    "Où travaillait-t-il?"
)
```

```
[Document(page_content="j'ai travaillé chez meta")]
```

III - Cours : Chapitre 9 - Retrieval Augmented Generation

Retrieval par utilisateurs

```
vectorstore.as_retriever(search_kwargs={"namespace": "dimitri"}).get_relevant_documents(
    "Où travaillait-t-il?"
)
```

```
[Document(page_content="j'ai travaillé chez deepmind")]
```

III - Cours : Chapitre 9 - Retrieval Augmented Generation

Retrieval par utilisateurs

```python
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import (
   ConfigurableField,
   RunnableBinding,
   RunnableLambda,
   RunnablePassthrough,
)
from langchain_openai import ChatOpenAI, OpenAIEmbeddings

# Il s'agit d'une chaîne de questions-réponses de base.
template = """Réponds à la question en te basant uniquement sur le contexte suivant:
{context}
Question: {question}
"""
prompt = ChatPromptTemplate.from_template(template)

model = ChatOpenAI()

retriever = vectorstore.as_retriever()
```

III - Cours : Chapitre 9 - Retrieval Augmented Generation

Retrieval par utilisateurs

Création de la chaîne avec le retriever configurable (par utilisateur)

```
configurable_retriever = retriever.configurable_fields(
    search_kwargs=ConfigurableField(
        id="search_kwargs",
        name="Search Kwargs",
        description="The search kwargs to use",
    )
)
```

```
chain = (
    {"context": configurable_retriever, "question": RunnablePassthrough()}
    | prompt
    | model
    | StrOutputParser()
)
```

III - Cours : Chapitre 9 - Retrieval Augmented Generation

Retrieval par utilisateurs

```
chain.invoke(
    "Où travaillait cet utilisateur?",
    config={"configurable": {"search_kwargs": {"namespace": "dimitri"}}},
)
```

```
'Cet utilisateur travaillait chez DeepMind.'
```

III - Cours : Chapitre 9 - Retrieval Augmented Generation

Retrieval par utilisateurs

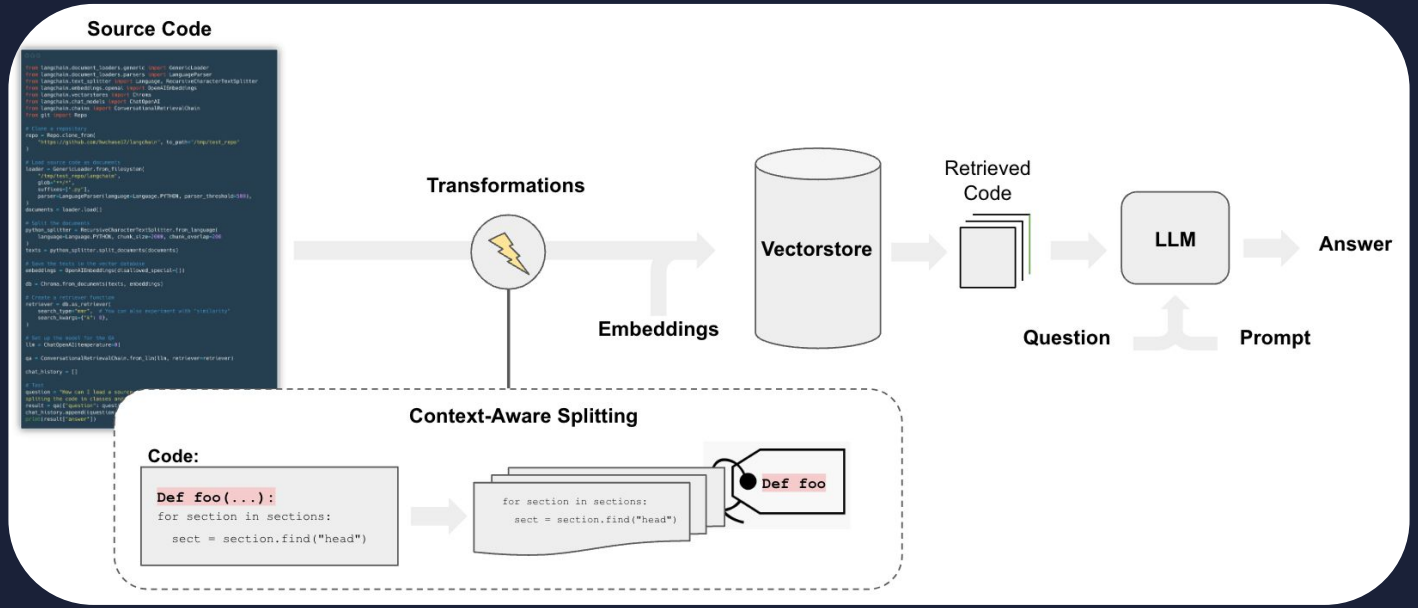
```
chain.invoke(
    "Où travaillait cet utilisateur?",
    config={"configurable": {"search_kwargs": {"namespace": "patrick"}}},
)
```

"L'utilisateur travaillait chez Meta."

III - Cours : Chapitre 9 - Retrieval Augmented Generation

TP - Use Case:

Utiliser ce qu'on a appris aujourd'hui pour recréer un outil type Github Copilot



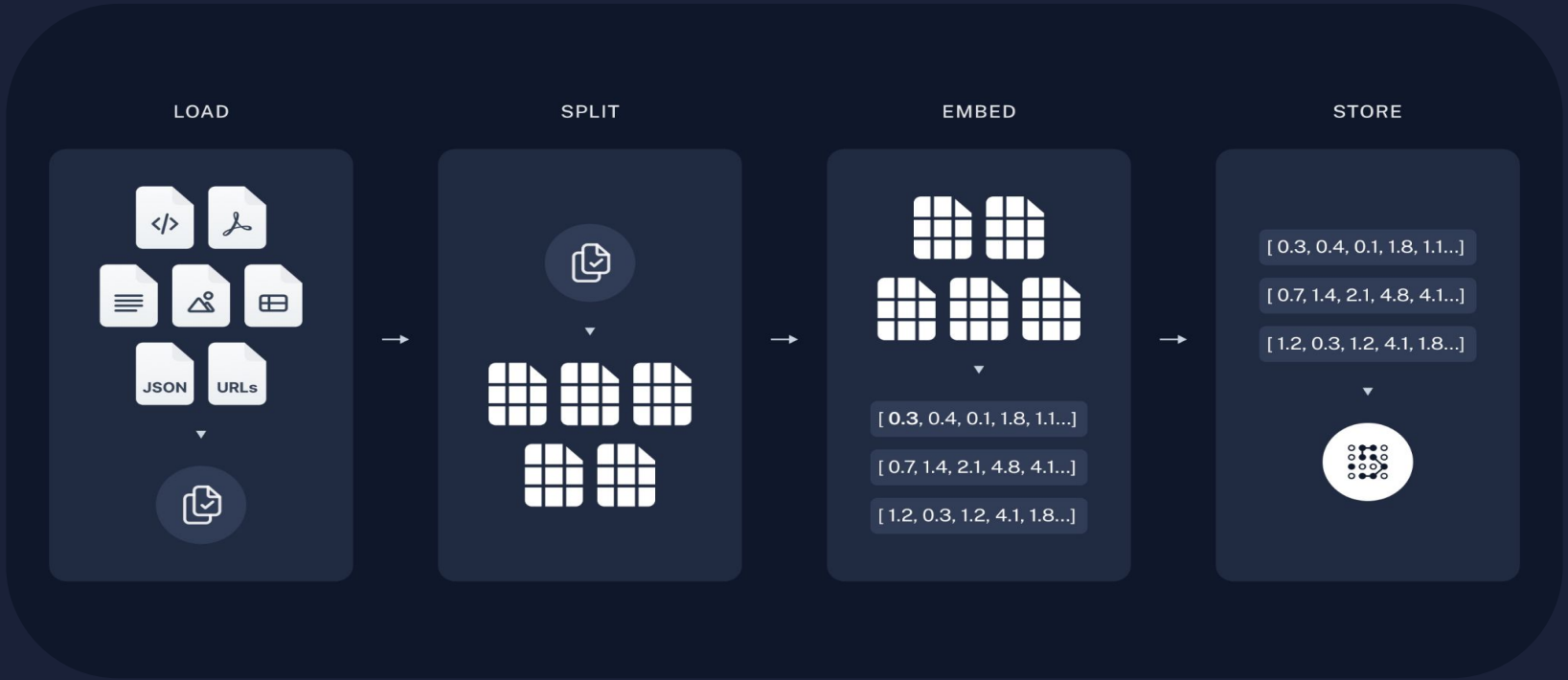
III - Cours : Chapitre 10 - Chargement et Séparation de Documents

Dans ce chapitre nous allons voir comment traiter des documents extérieurs pour utiliser du RAG avec des Loader et Splitters.

Les Loaders permettent de charger un ou plusieurs documents avec un format spécifique (Site Web, Base de Données, etc…).

Les splitters permettent de découper ce document pour pouvoir le “feed” au LLM et utiliser le contexte du document en aval.

III - Cours : Chapitre 10 - Chargement et Séparation de Documents



III - Cours : Chapitre 10 - Chargement et Séparation de Documents

Exemple: Chargement de document

```
from langchain_community.document_loaders import WebBaseLoader
loader = WebBaseLoader("https://docs.smith.langchain.com/overview")

docs = loader.load()
docs
```

```
[Document(page_content="\n\n\n\n\nLangSmith | 🦜🛠️ LangSmith\n\n\n\n\n\nSkip to main content🦜🛠️ LangSmith DocsLangChain Python DocsLangChain JS/TS DocsLangSmith API DocsSearchGo to AppLangSmithUser GuideSetupPricing (Coming Soon)Self-HostingTracingEvaluationMonitoringPrompt HubLangSmithOn this pageLangSmithIntroduction\u200bLangSmith is a platform for building production-grade LLM applications.It lets you debug, test, evaluate, and monitor chains and intelligent agents built on any LLM framework and seamlessly integrates with LangChain, the go-to open source framework for building with LLMs.LangSmith is developed by LangChain, the company behind the open source LangChain framework.Quick Start\u200bTracing:...]
```

III - Cours : Chapitre 10 - Chargement et Séparation de Documents

Exemple: Découpage de document

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter()
documents = text_splitter.split_documents(docs)
```

III - Cours : Chapitre 10 - Chargement et Séparation de Documents

Exemple: Indexage en VectorDB

```
from langchain_openai import OpenAIEmbeddings
from langchain_pinecone import PineconeVectorStore

embeddings = OpenAIEmbeddings()
vectorstore = PineconeVectorStore(index_name="test-example", embedding=embeddings)

docsearch = PineconeVectorStore.from_documents(documents, embeddings,
index_name="test-example")
```