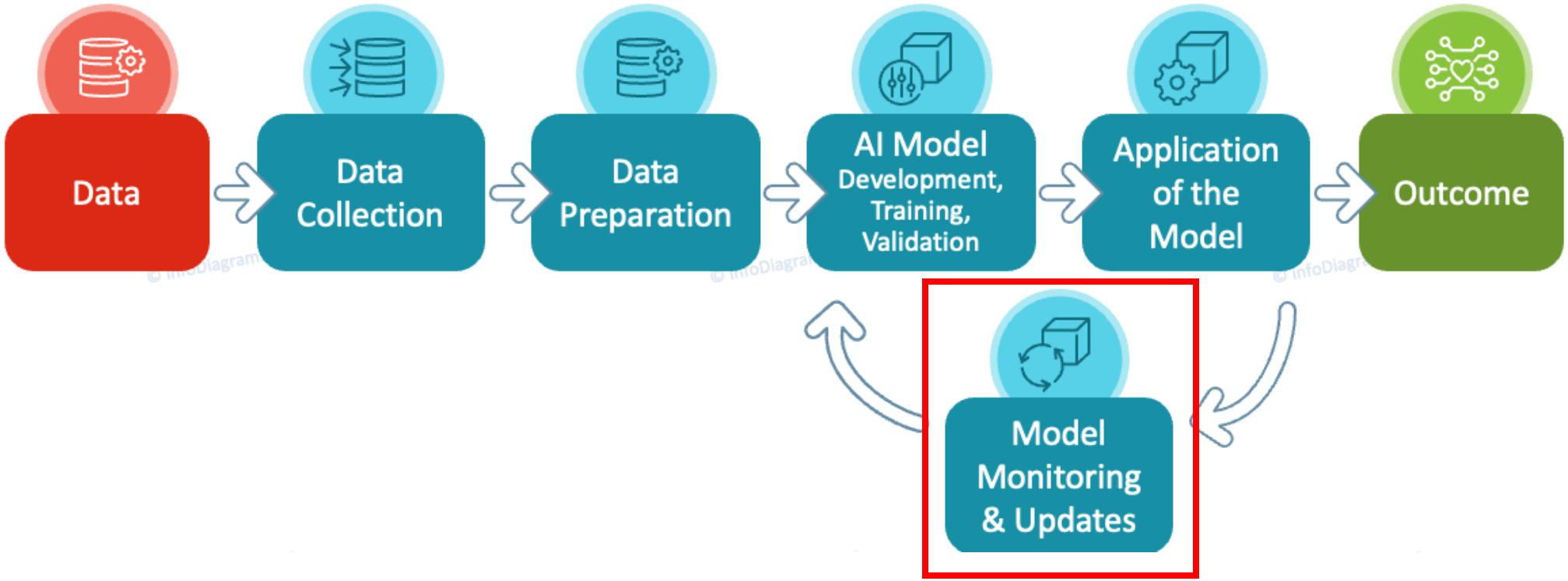
IMEN891K/IDSC501 Data Engineering

17. **LLM Systems I**: Basics

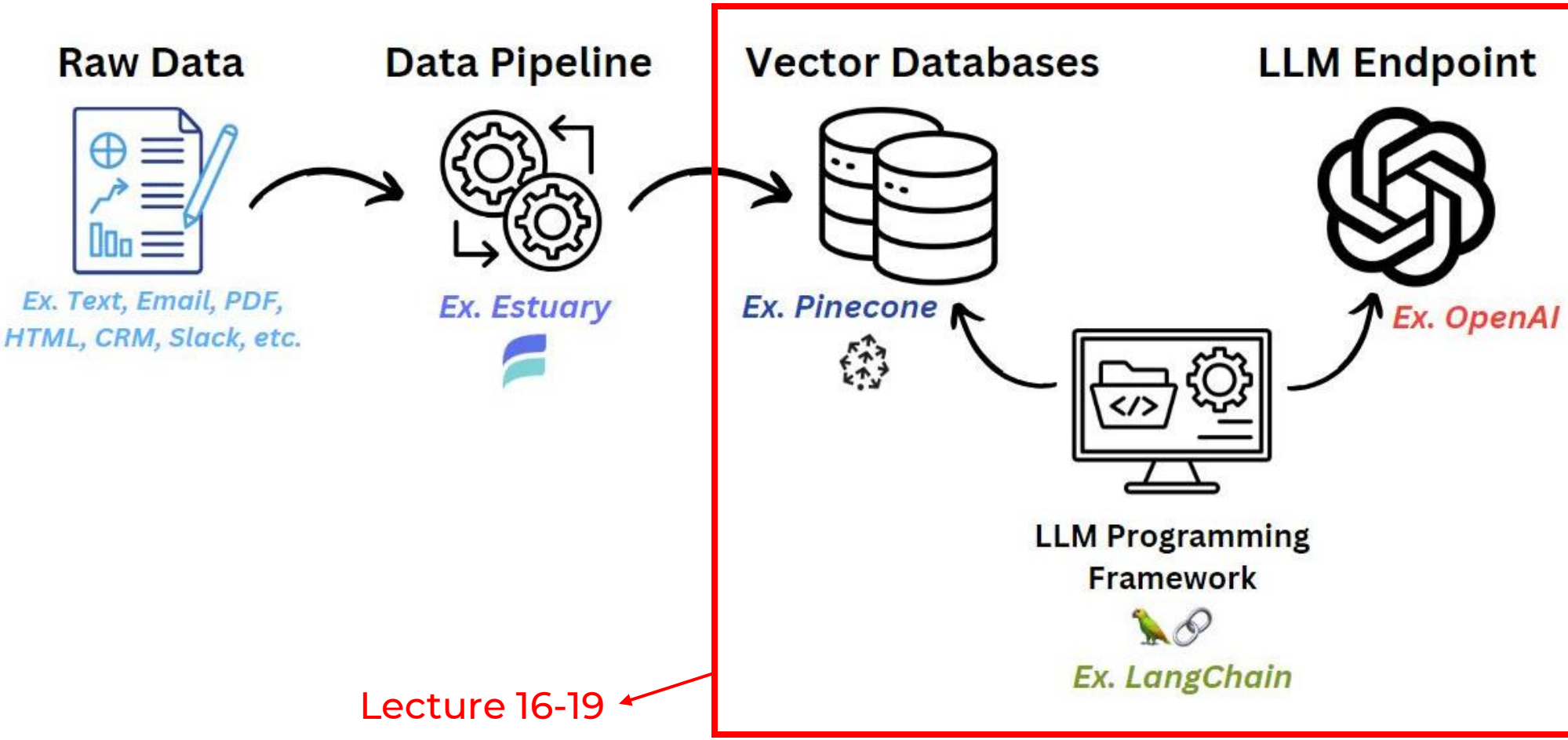
Fall 2025

Instructor: **Sangwoo Mo**

Recap: Course Overview

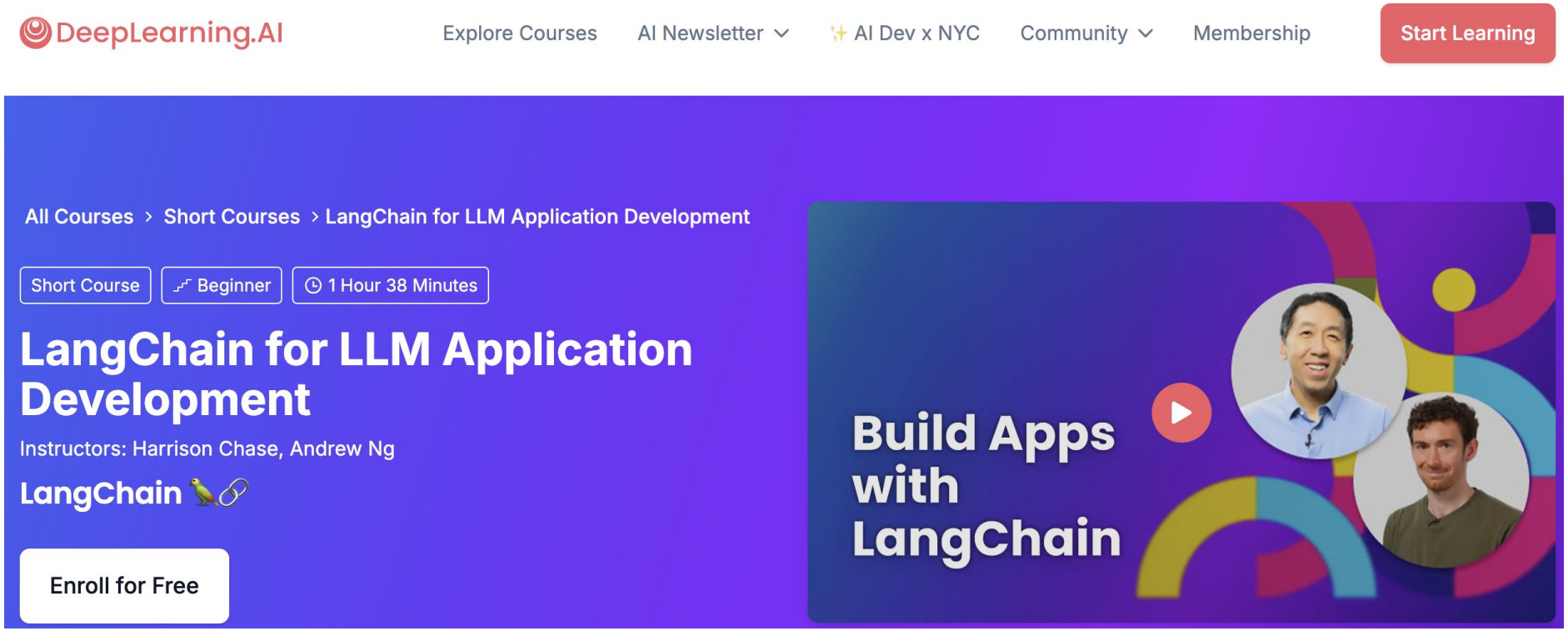


Recap: Course Overview

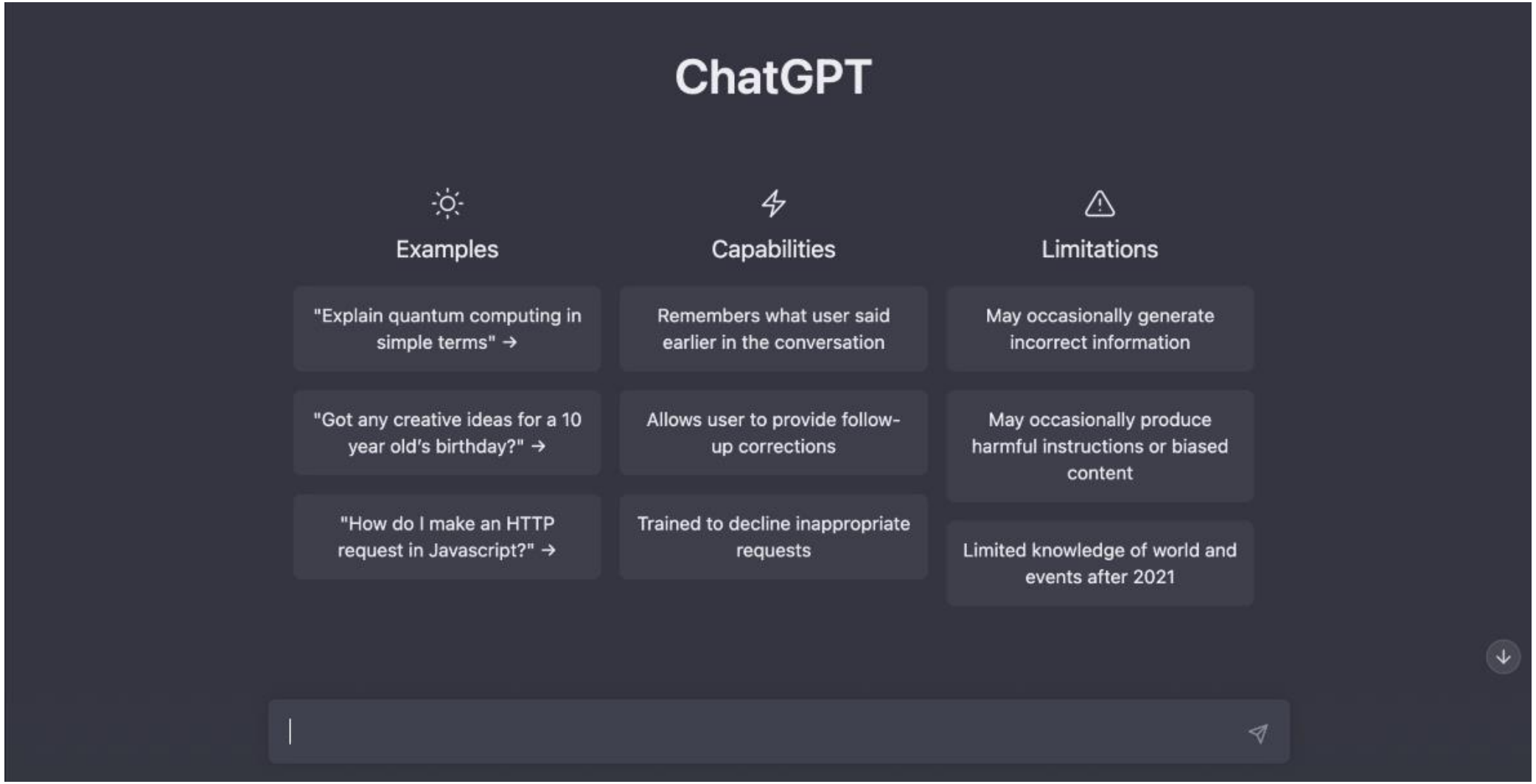


Disclaimer

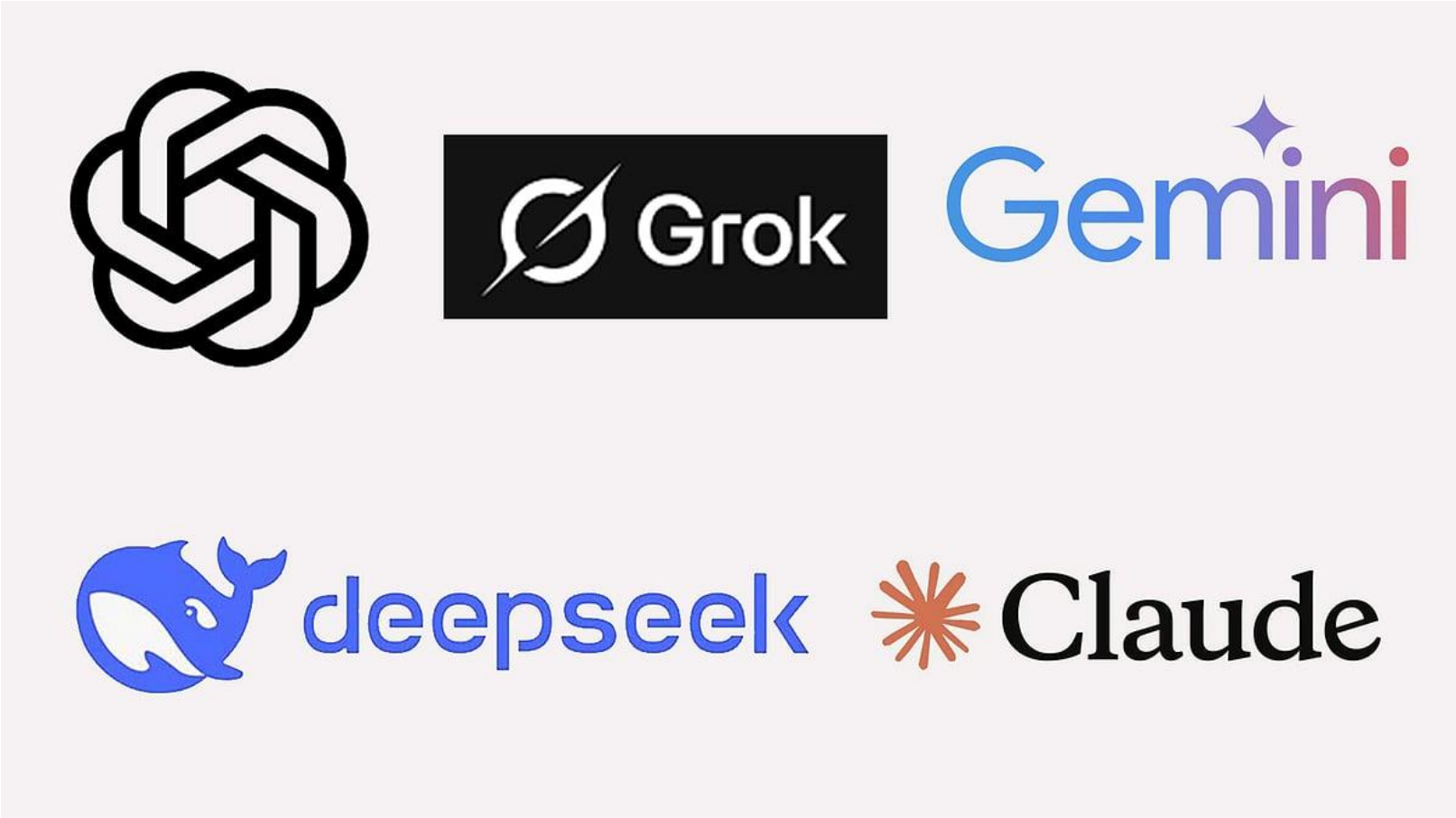
The LangChain part was heavily copy-pasted from DeepLearning.AI



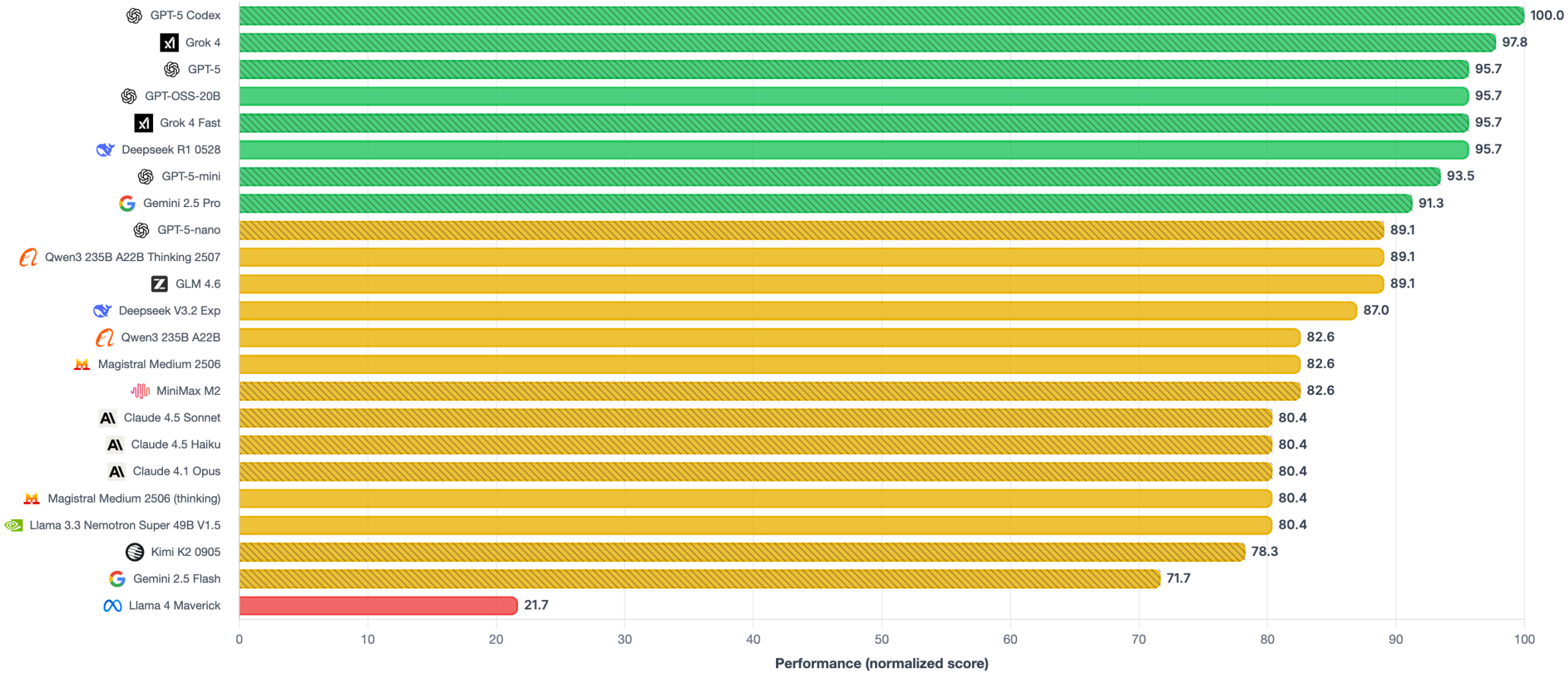
Large Language Models (LLMs)



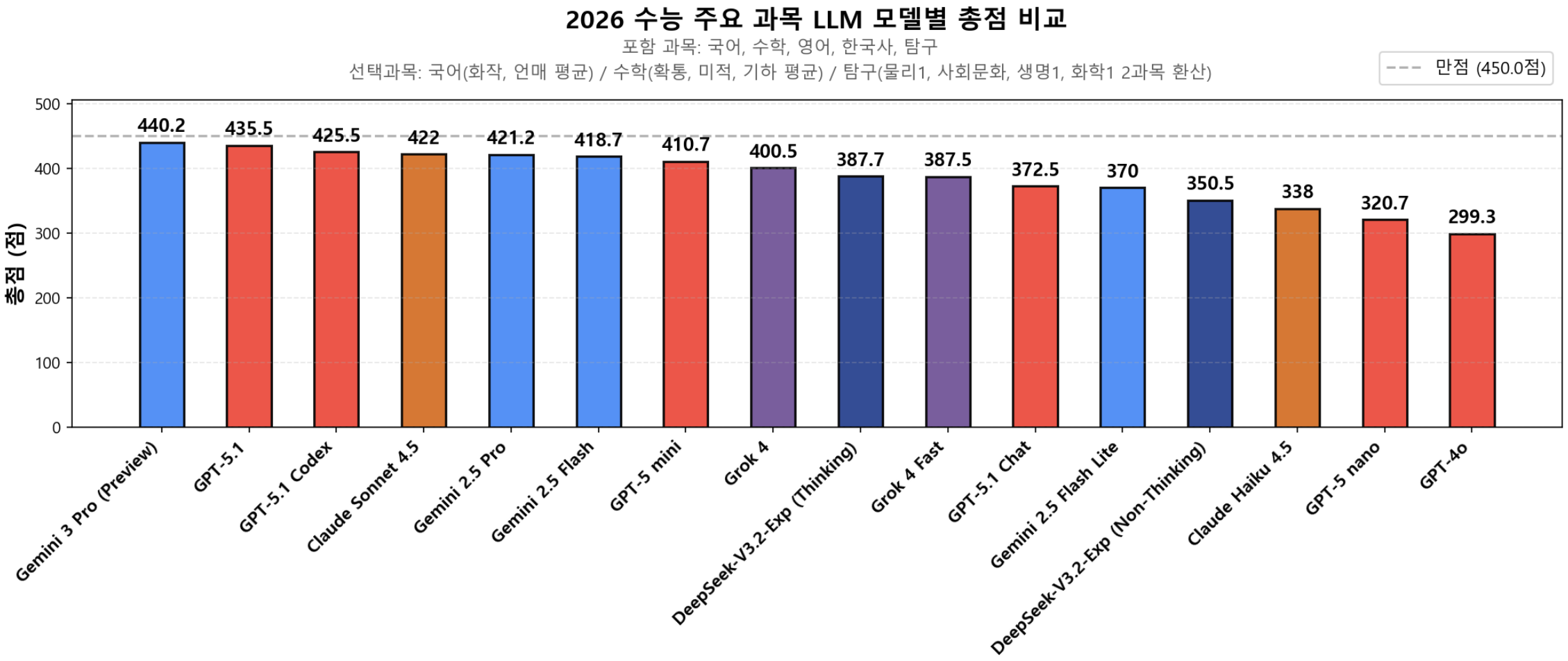
Large Language Models (LLMs)



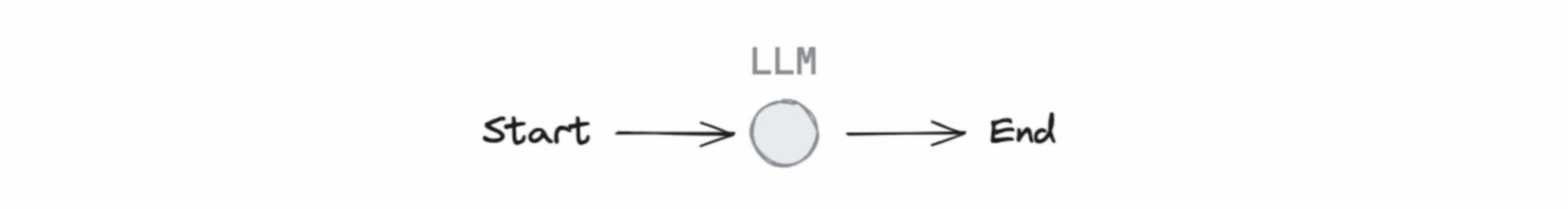
Large Language Models (LLMs)



Large Language Models (LLMs)



LangChain



LangChain

A solitary language model is fairly limited...



... e.g., access to tools, external context, multi-step workflows.

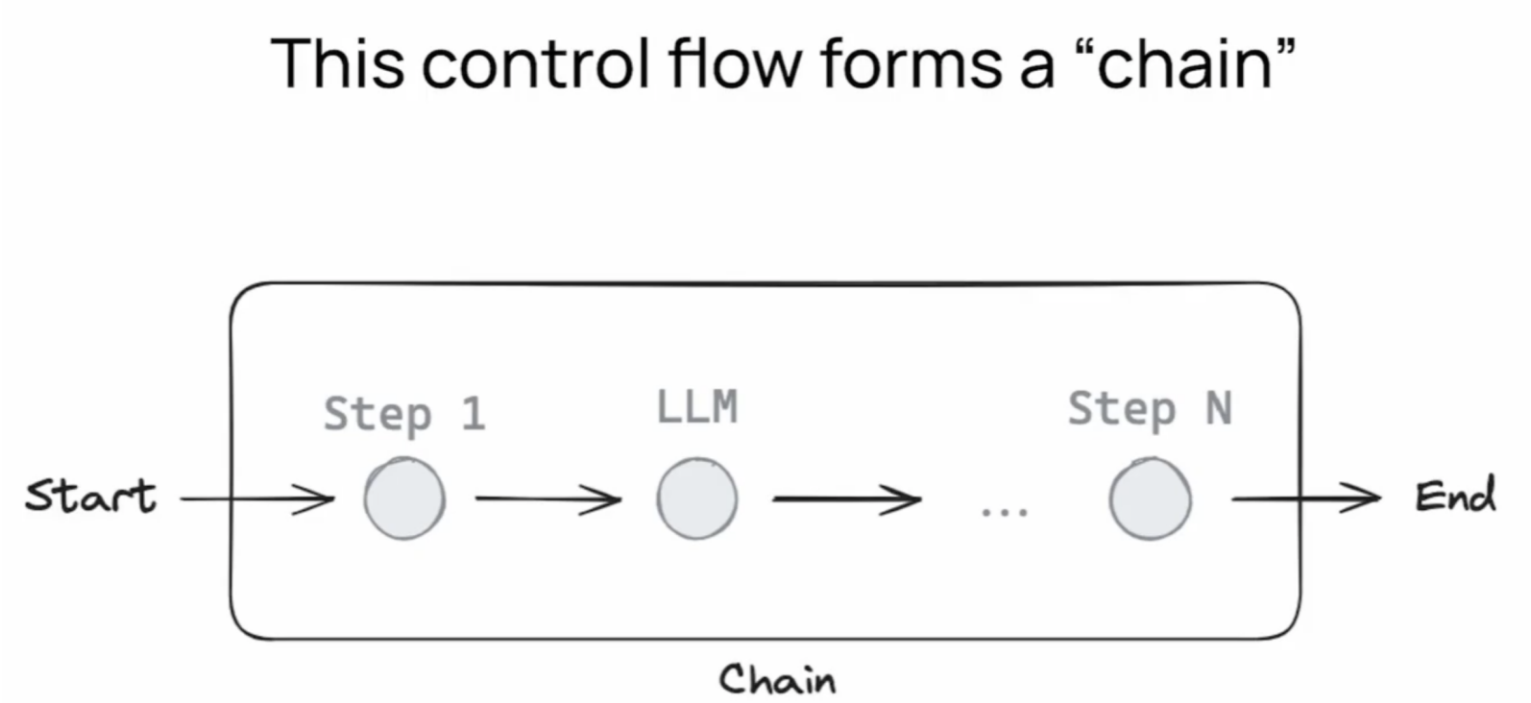
LangChain

So, many LLM applications use a control flow ...



... with steps pre / post-LLM call (tool calls, retrieval, etc).

LangChain


This control flow forms a “chain”
Step 1
LLM
Step N
Start
...
End
Chain

LangChain

Open-source developer framework for building LLM applications

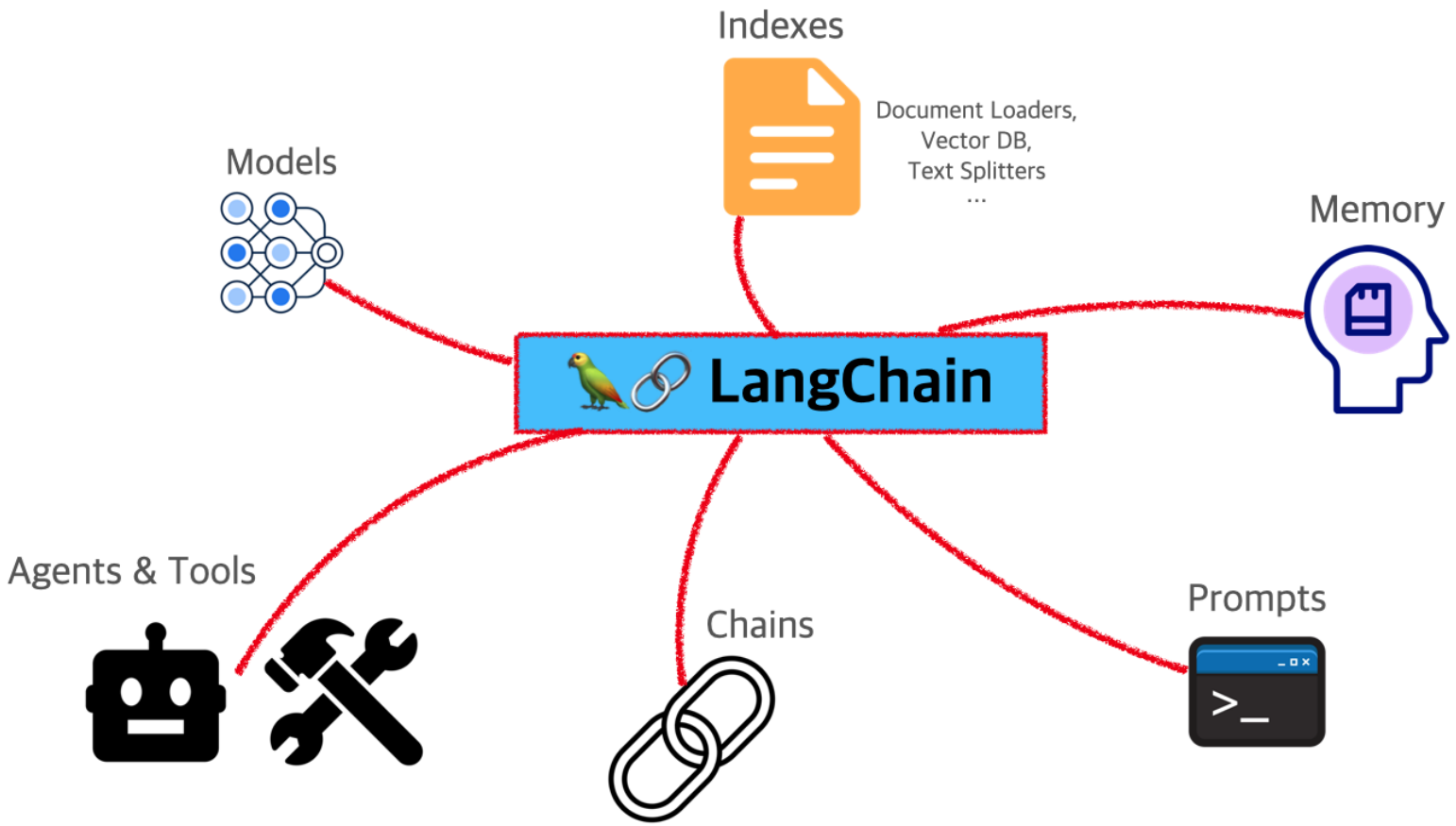
Python and TypeScript packages

Focused on composition and modularity

Key value adds:

1. Modular components (and implementations of those components)
2. Use cases - common ways to combine those components together

LangChain – Overview



LangChain – Overview

- Prompts
  - Prompt Templates
  - Output Parsers: 5+ implementations
    - Retry/fixing logic
  - Example Selectors: 5+ implementations
- Models
  - LLM's: 20+ integrations
  - Chat Models
  - Text Embedding Models: 10+ integrations
- Indexes
  - Document Loaders: 50+ implementations
  - Text Splitters: 10+ implementations
  - Vector stores: 10+ integrations
  - Retrievers: 5+ integrations/implementations
- Chains
  - Can be used as building blocks for other chains
  - More application specific chains: 20+ different types
- Agents
  - Agent Types: 5+ types
    - Algorithms for getting LLMs to use tools
  - Agent Toolkits: 10+ implementations
    - Agents armed with specific tools for a specific application

Models & Prompts



Models & Prompts

Get your OpenAI API Key

In [ ]:
```
#!pip install python-dotenv
#!pip install openai
```

In [ ]:
```
import os
import openai

from dotenv import load_dotenv, find_dotenv
_ = load_dotenv(find_dotenv()) # read local .env file
openai.api_key = os.environ['OPENAI_API_KEY']
```

Models & Prompts

Chat API : OpenAI

Let's start with a direct API calls to OpenAI.

In [ ]:
```python
def get_completion(prompt, model=llm_model):
    messages = [{"role": "user", "content": prompt}]
    response = openai.ChatCompletion.create(
        model=model,
        messages=messages,
        temperature=0,
    )
    return response.choices[0].message["content"]
```

Models & Prompts

```
get_completion("What is 1+1?")
```

Models & Prompts

```
get_completion("What is 1+1?")
```

```
'As an AI language model, I can tell you that the answer
to 1+1 is 2.'
```

Models & Prompts

```
customer_email = """
Arrr, I be fuming that me blender lid \
flew off and splattered me kitchen walls \
with smoothie! And to make matters worse,\
the warranty don't cover the cost of \
cleaning up me kitchen. I need yer help \
right now, matey!
"""
```

```
style = """American English \
in a calm and respectful tone
"""
```

```
prompt = f"""Translate the text \
that is delimited by triple backticks
into a style that is {style}.
text: ```{customer_email}```
"""

print(prompt)
```

Models & Prompts

```
response = get_completion(prompt)
```

```
response
```

```
"I'm really frustrated that my blender lid flew off and ma
de a mess of my kitchen walls with smoothie! To add insult
to injury, the warranty doesn't cover the cost of cleaning
up my kitchen. I could really use your help right now, my
friend."
```

Models & Prompts

Chat API : LangChain

Let's try how we can do the same using LangChain.

In [ ]:
```
#!pip install --upgrade langchain
```

Model

In [ ]:
```
from langchain.chat_models import ChatOpenAI
```

In [ ]:
```
# To control the randomness and creativity of the generated
# text by an LLM, use temperature = 0.0
chat = ChatOpenAI(temperature=0.0, model=llm_model)
chat
```

Models & Prompts

```
template_string = """Translate the text \
that is delimited by triple backticks \
into a style that is {style}. \
text: ```{text}```
"""
```

```
from langchain.prompts import ChatPromptTemplate

prompt_template = ChatPromptTemplate.from_template(template_
```

```
prompt_template.messages[0].prompt
```

```
PromptTemplate(input_variables=['style', 'text'], output_p
arser=None, partial_variables={}, template='Translate the
text that is delimited by triple backticks into a style th
at is {style}. text: ```{text}```\n', template_format='f-s
tring', validate_template=True)
```

Models & Prompts

```
customer_style = """American English \
in a calm and respectful tone
"""
```

```
customer_email = """
Arrr, I be fuming that me blender lid \
flew off and splattered me kitchen walls \
with smoothie! And to make matters worse, \
the warranty don't cover the cost of \
cleaning up me kitchen. I need yer help \
right now, matey!
"""
```

```
customer_messages = prompt_template.format_messages(
                    style=customer_style,
                    text=customer_email)
```

```
print(type(customer_messages))
print(type(customer_messages[0]))
```

```
<class 'list'>
<class 'langchain.schema.HumanMessage'>
```

Models & Prompts

```
customer_response = chat(customer_messages)
```

```
print(customer_response.content)
```

```
I'm really frustrated that my blender lid flew off and mad
e a mess of my kitchen walls with smoothie. To add to my f
rustration, the warranty doesn't cover the cost of cleanin
g up my kitchen. Can you please help me out, friend?
```

Models & Prompts – Why Templates?

```
prompt = """
Your task is to determine if
the student's solution is
correct or not.
```

```
To solve the problem do the following:
- First, work out your own solution to the problem.
- Then compare your solution to the student's solution
and evaluate if the student's solution is correct or not.
...
Use the following format:
Question:
```
question here
```
Student's solution:
```
student's solution here
```
Actual solution:
```
...
steps to work out the solution and your solution here
```
Is the student's solution the same as actual solution \
just calculated:
```
yes or no
```
Student grade:
```
correct or incorrect
```

Question:
```
{question}
```
Student's solution:
```
{student_solution}
```
Actual solution:
"""
```

Prompts can be long and detailed.

Reuse good prompts when you can!

LangChain also provides prompts for common operations.

Models & Prompts – Output Parsers

Parse the LLM output string into a Python dictionary

In [ ]:
```
from langchain.output_parsers import ResponseSchema
from langchain.output_parsers import StructuredOutputParser
```

In [ ]:
```
gift_schema = ResponseSchema(name="gift",
                             description="Was the item purchased\
                             as a gift for someone else? \
                             Answer True if yes,\
                             False if not or unknown.")
delivery_days_schema = ResponseSchema(name="delivery_days",
                                      description="How many days\
                                      did it take for the product\
                                      to arrive? If this \
                                      information is not found,\
                                      output -1.")
price_value_schema = ResponseSchema(name="price_value",
                                    description="Extract any\
                                    sentences about the value or \
                                    price, and output them as a \
                                    comma separated Python list.")

response_schemas = [gift_schema,
                    delivery_days_schema,
                    price_value_schema]
```

Models & Prompts – Output Parsers

```
output_parser = StructuredOutputParser.from_response_schemas(respo
```

```
format_instructions = output_parser.get_format_instructions()
```

```
print(format_instructions)
```

```
The output should be a markdown code snippet formatted in
the following schema, including the leading and trailing
"\`\`\`json" and "\`\`\`":

```json
{
        "gift": string  // Was the item purchased as a gif
t for someone else? Answer True if yes, False if not or un
known.
        "delivery_days": string  // How many days did it t
ake for the product to arrive? If this information is not
found, output -1.
        "price_value": string  // Extract any sentences ab
out the value or price, and output them as a comma separat
ed Python list.
}
```
```

Models & Prompts – Output Parsers

```
review_template_2 = """\
For the following text, extract the following information:

gift: Was the item purchased as a gift for someone else? \
Answer True if yes, False if not or unknown.

delivery_days: How many days did it take for the product\
to arrive? If this information is not found, output -1.

price_value: Extract any sentences about the value or price,\
and output them as a comma separated Python list.

text: {text}

{format_instructions}
"""

prompt = ChatPromptTemplate.from_template(template=review_template

messages = prompt.format_messages(text=customer_review,
                                  format_instructions=format_instruc
```

Models & Prompts – Output Parsers

```
response = chat(messages)
```

```
print(response.content)
```

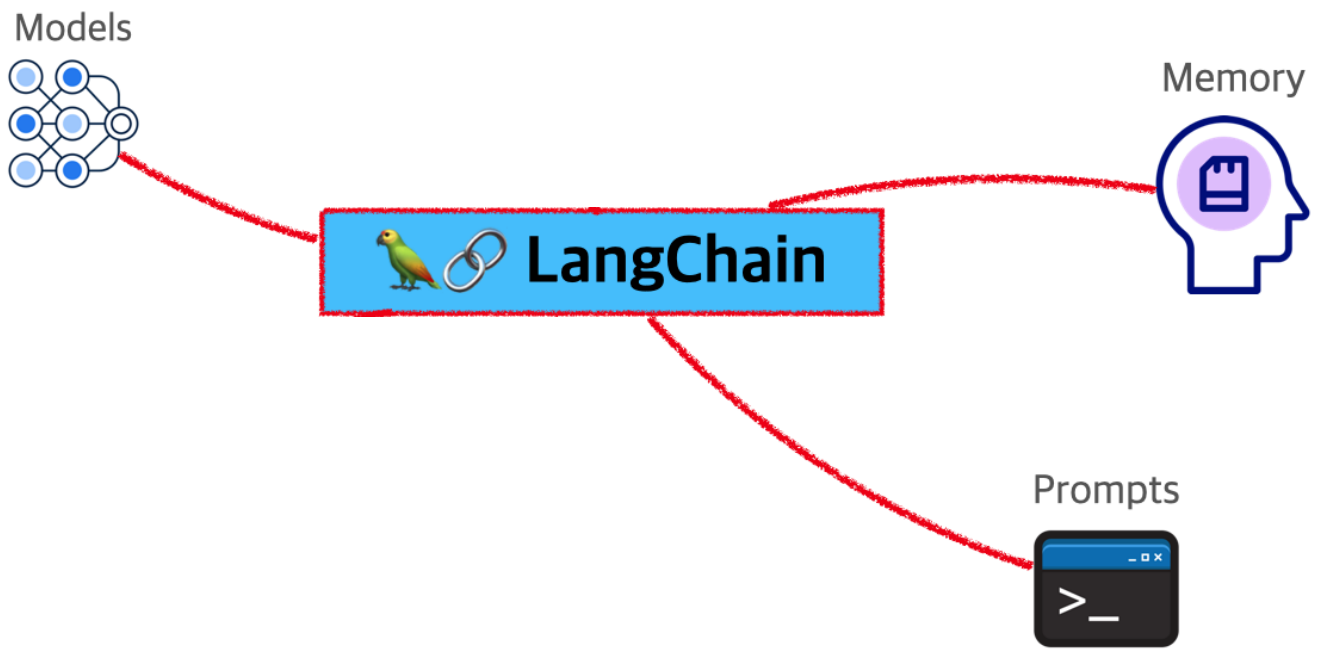
```
```json
{
        "gift": true,
        "delivery_days": "2",
        "price_value": ["It's slightly more expensive than
the other leaf blowers out there, but I think it's worth i
t for the extra features."]
}
```
```

```
output_dict = output_parser.parse(response.content)
```

```
output_dict
```

```
{'gift': True,
 'delivery_days': '2',
 'price_value': ["It's slightly more expensive than the ot
her leaf blowers out there, but I think it's worth it for
the extra features."]}
```

Memory

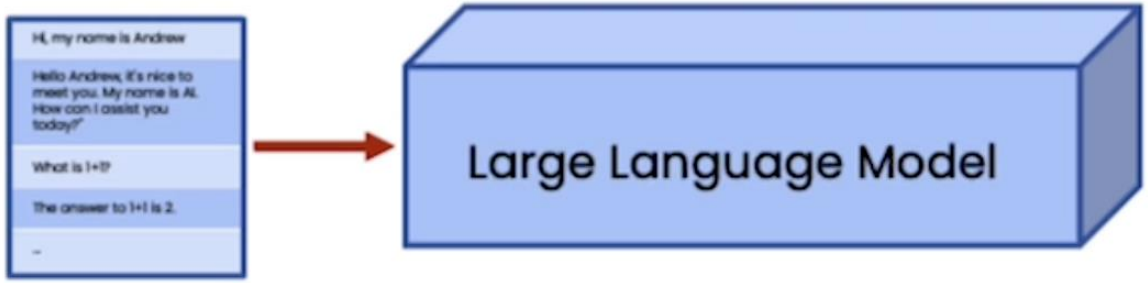


Memory

Large Language Models are ‘stateless’

- Each transaction is independent

Chatbots appear to have memory by providing the full conversation as ‘context’



LangChain provides several kinds of ‘memory’ to store and accumulate the conversation

Memory

ConversationBufferMemory

In [ ]:
```
from langchain.chat_models import ChatOpenAI
from langchain.chains import ConversationChain
from langchain.memory import ConversationBufferMemory
```

In [ ]:
```
llm = ChatOpenAI(temperature=0.0, model=llm_model)
memory = ConversationBufferMemory()
conversation = ConversationChain(
    llm=llm,
    memory = memory,
    verbose=True
)
```

In [ ]:
```
conversation.predict(input="Hi, my name is Andrew")
```

In [ ]:
```
conversation.predict(input="What is 1+1?")
```

In [ ]:
```
conversation.predict(input="What is my name?")
```

Memory

```
conversation.predict(input="Hi, my name is Andrew")
```

```
"Hello Andrew, it's nice to meet you. My name is AI. How c
an I assist you today?"
```

```
conversation.predict(input="What is 1+1?")
```

```
'The answer to 1+1 is 2.'
```

```
conversation.predict(input="What is my name?")
```

```
'Your name is Andrew, as you mentioned earlier.'
```

Memory

```
print(memory.buffer)
```

```
Human: Hi, my name is Andrew
AI: Hello Andrew, it's nice to meet you. My name is AI. Ho
w can I assist you today?
Human: What is 1+1?
AI: The answer to 1+1 is 2.
Human: What is my name?
AI: Your name is Andrew, as you mentioned earlier.
```

```
memory.load_memory_variables({})
```

```
{'history': "Human: Hi, my name is Andrew\nAI: Hello Andre
w, it's nice to meet you. My name is AI. How can I assist
you today?\nHuman: What is 1+1?\nAI: The answer to 1+1 is
2.\nHuman: What is my name?\nAI: Your name is Andrew, as y
ou mentioned earlier."}
```

Memory with Limited Buffers

ConversationBufferWindowMemory

In [ ]:
```
from langchain.memory import ConversationBufferWindowMemory
```

In [ ]:
```
memory = ConversationBufferWindowMemory(k=1)
```

Memory with Limited Buffers

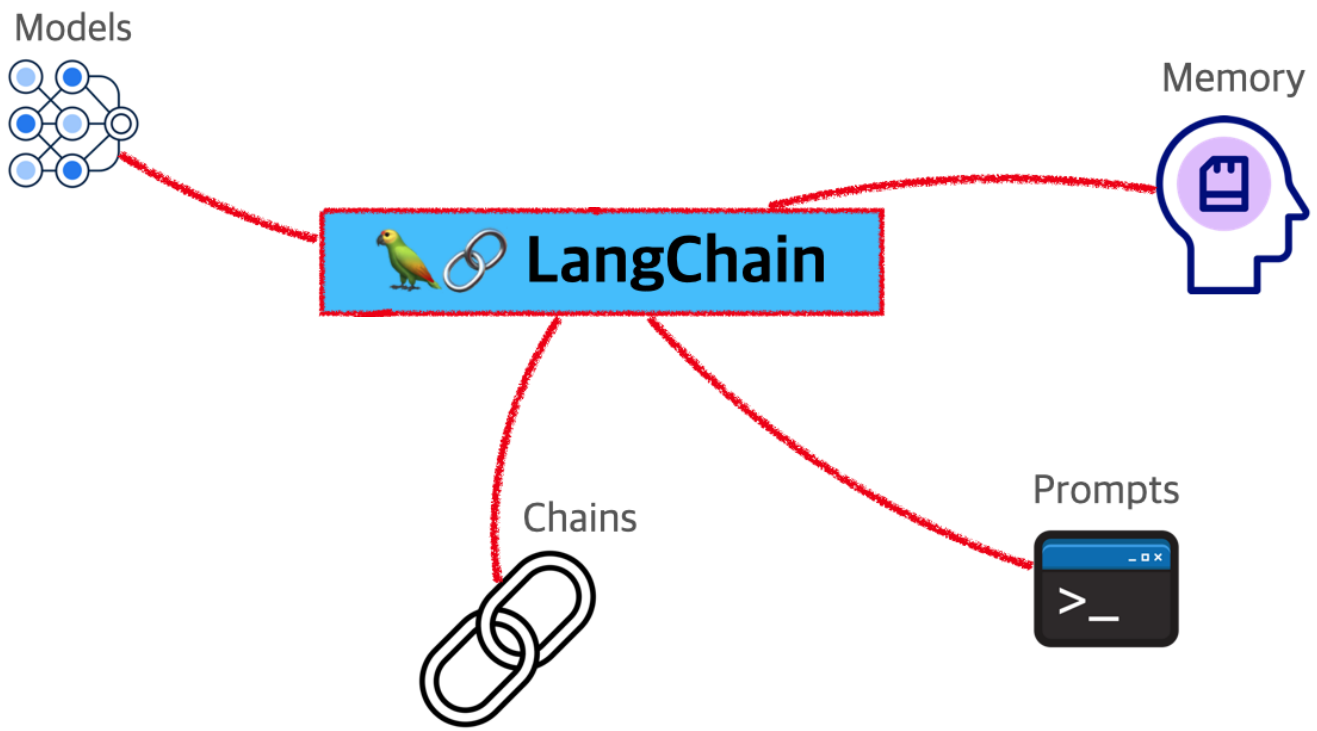
ConversationSummaryMemory

In [ ]:
```
from langchain.memory import ConversationSummaryBufferMemory
```

In [ ]:
```
# create a long string
schedule = "There is a meeting at 8am with your product team. \
You will need your powerpoint presentation prepared. \
9am-12pm have time to work on your LangChain \
project which will go quickly because Langchain is such a powerful
At Noon, lunch at the italian resturant with a customer who is dri
from over an hour away to meet you to understand the latest in AI.
Be sure to bring your laptop to show the latest LLM demo."

memory = ConversationSummaryBufferMemory(llm=llm, max_token_limit=
memory.save_context({"input": "Hello"}, {"output": "What's up"})
memory.save_context({"input": "Not much, just hanging"},
                    {"output": "Cool"})
memory.save_context({"input": "What is on the schedule today?"},
                    {"output": f"{schedule}"})
```

Chains



Chains

LLMChain

In [ ]:
```
from langchain.chat_models import ChatOpenAI
from langchain.prompts import ChatPromptTemplate
from langchain.chains import LLMChain
```

In [ ]:
```
llm = ChatOpenAI(temperature=0.9, model=llm_model)
```

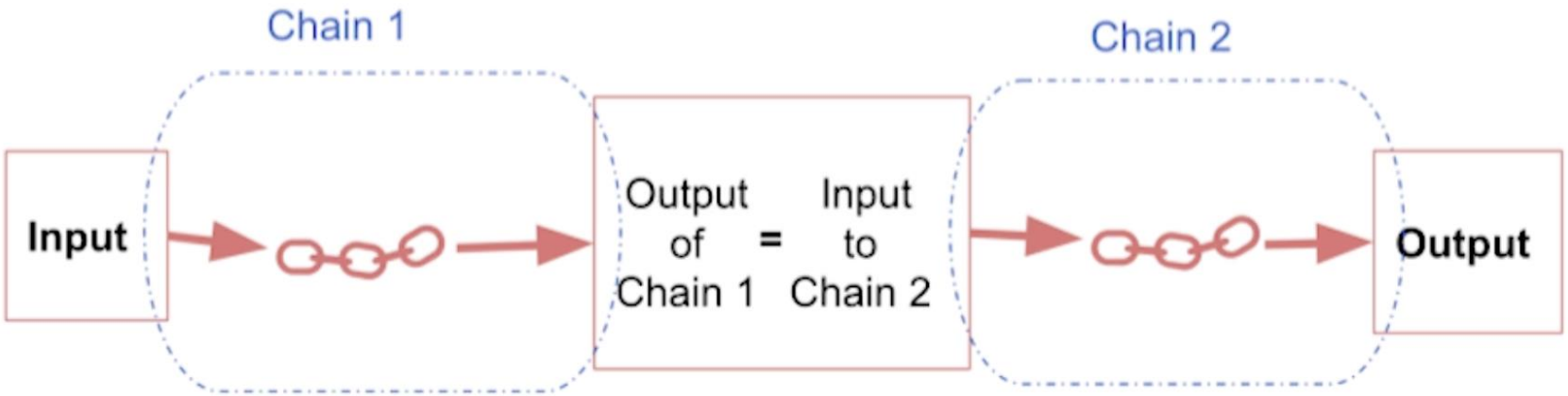
In [ ]:
```
prompt = ChatPromptTemplate.from_template(
    "What is the best name to describe \
    a company that makes {product}?"
)
```

In [ ]:
```
chain = LLMChain(llm=llm, prompt=prompt)
```

```
product = "Queen Size Sheet Set"
chain.run(product)
```

```
'Royal Beddings.'
```

Simple Sequential Chain



Simple Sequential Chain

SimpleSequentialChain

In [ ]:
```
from langchain.chains import SimpleSequentialChain
```

In [ ]:
```
llm = ChatOpenAI(temperature=0.9, model=llm_model)

# prompt template 1
first_prompt = ChatPromptTemplate.from_template(
    "What is the best name to describe \
    a company that makes {product}?"
)

# Chain 1
chain_one = LLMChain(llm=llm, prompt=first_prompt)
```

In [ ]:
```
# prompt template 2
second_prompt = ChatPromptTemplate.from_template(
    "Write a 20 words description for the following \
    company:{company_name}"
)
# chain 2
chain_two = LLMChain(llm=llm, prompt=second_prompt)
```

In [ ]:
```
overall_simple_chain = SimpleSequentialChain(chains=[chain_one, ch
                                             verbose=True
                                            )
```

Simple Sequential Chain

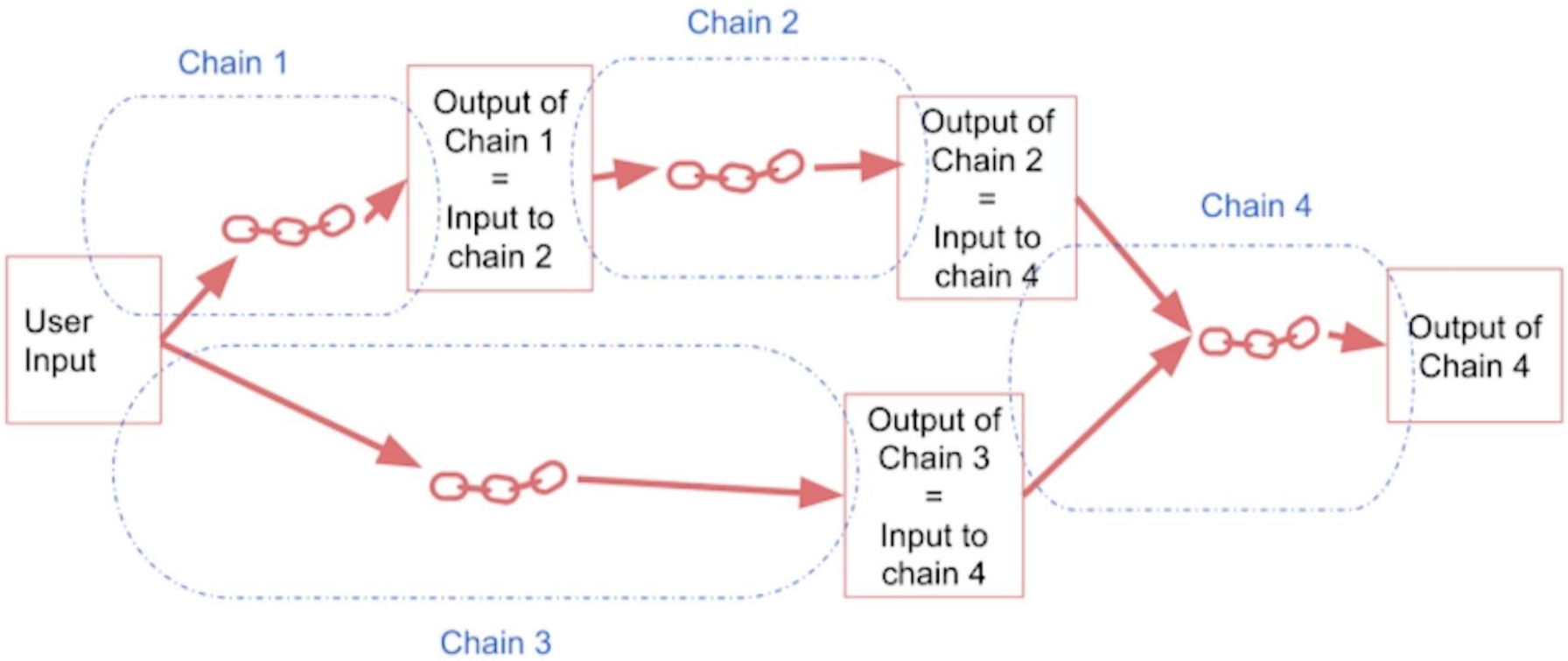
```
overall_simple_chain.run(product)
```

```
> Entering new SimpleSequentialChain chain...
"Regal Bedding"
Regal Bedding offers luxurious bedding sets, including co
mforters, sheets, and pillows, designed to ensure a comfo
rtable and restful sleep experience.

> Finished chain.

'Regal Bedding offers luxurious bedding sets, including c
omforters, sheets, and pillows, designed to ensure a comf
ortable and restful sleep experience.'
```

Sequential Chain



Sequential Chain

SequentialChain

In [ ]:
```
from langchain.chains import SequentialChain
```

In [ ]:
```
llm = ChatOpenAI(temperature=0.9, model=llm_model)

# prompt template 1: translate to english
first_prompt = ChatPromptTemplate.from_template(
    "Translate the following review to english:"
    "\n\n{Review}"
)
# chain 1: input= Review and output= English_Review
chain_one = LLMChain(llm=llm, prompt=first_prompt,
                     output_key="English_Review"
                    )
```

In [ ]:
```
second_prompt = ChatPromptTemplate.from_template(
    "Can you summarize the following review in 1 sentence:"
    "\n\n{English_Review}"
)
# chain 2: input= English_Review and output= summary
chain_two = LLMChain(llm=llm, prompt=second_prompt,
                     output_key="summary"
                    )
```

```
# prompt template 3: translate to english
third_prompt = ChatPromptTemplate.from_template(
    "What language is the following review:\n\n{Review}"
)
# chain 3: input= Review and output= language
chain_three = LLMChain(llm=llm, prompt=third_prompt,
                       output_key="language"
                      )
```

```
# prompt template 4: follow up message
fourth_prompt = ChatPromptTemplate.from_template(
    "Write a follow up response to the following "
    "summary in the specified language:"
    "\n\nSummary: {summary}\n\nLanguage: {language}"
)
# chain 4: input= summary, language and output= followup_message
chain_four = LLMChain(llm=llm, prompt=fourth_prompt,
                      output_key="followup_message"
                     )
```

Sequential Chain

```
# overall_chain: input= Review
# and output= English_Review,summary, followup_message
overall_chain = SequentialChain(
    chains=[chain_one, chain_two, chain_three, chain_four],
    input_variables=["Review"],
    output_variables=["English_Review", "summary","followup_messag
    verbose=True
)
```

Sequential Chain

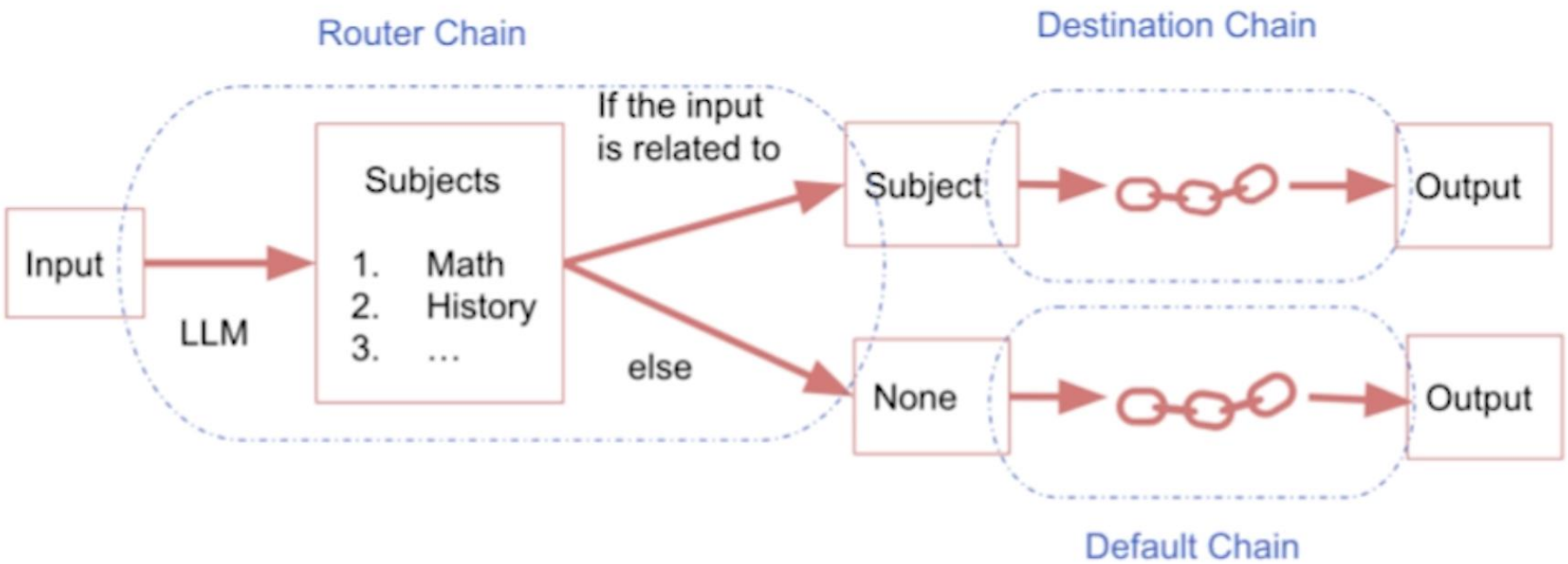
```
review = df.Review[5]
overall_chain(review)
```

```
> Entering new SequentialChain chain...

> Finished chain.

{'Review': "Je trouve le goût médiocre. La mousse ne tien
t pas, c'est bizarre. J'achète les mêmes dans le commerce
et le goût est bien meilleur...\nVieux lot ou contrefaçon
!?",
 'English_Review': "I find the taste mediocre. The foam d
oesn't hold, it's weird. I buy the same ones in stores an
d the taste is much better... Old batch or counterfei
t?!",
 'summary': "The reviewer thinks the taste is mediocre, t
he foam doesn't hold, and suspects it could be an old bat
ch or counterfeit product.",
 'followup_message': "Réponse : Le critique pense que le
goût est médiocre, que la mousse ne tient pas et soupçonn
e qu'il pourrait s'agit d'un ancien lot ou d'un produit c
ontrefait. Nous vous remercions pour votre retour sur ce
produit. Nous travaillons en étroite collaboration avec n
os fournisseurs pour nous assurer que nos clients reçoive
nt toujours des produits frais et authentiques. Nous allo
ns enquêter sur votre problème de mousse et faire tout no
tre possible pour améliorer la qualité de notre produit.
Nous espérons que vous nous donnerez une autre chance de
gagner votre confiance et votre satisfaction."}
```

Router Chain



Router Chain

```
from langchain.chains.router import MultiPromptChain
from langchain.chains.router.llm_router import LLMRouterChain,Rout
from langchain.prompts import PromptTemplate
```

```
llm = ChatOpenAI(temperature=0, model=llm_model)
```

```


destination_chains = {}
for p_info in prompt_infos:
    name = p_info["name"]
    prompt_template = p_info["prompt_template"]
    prompt = ChatPromptTemplate.from_template(template=prompt_temp
    chain = LLMChain(llm=llm, prompt=prompt)
    destination_chains[name] = chain

destinations = [f"{p['name']}: {p['description']}" for p in prompt
destinations_str = "\n".join(destinations)
```

```
prompt_infos = [
    {
        "name": "physics",
        "description": "Good for answering questions about physics
        "prompt_template": physics_template
    },
    {
        "name": "math",
        "description": "Good for answering math questions",
        "prompt_template": math_template
    },
    {
        "name": "History",
        "description": "Good for answering history questions",
        "prompt_template": history_template
    },
    {
        "name": "computer science",
        "description": "Good for answering computer science questi
        "prompt_template": computerscience_template
    }
]
```

Router Chain

```
MULTI_PROMPT_ROUTER_TEMPLATE = """Given a raw text input to a \
language model select the model prompt best suited for the input.
You will be given the names of the available prompts and a \
description of what the prompt is best suited for. \
You may also revise the original input if you think that revising\
it will ultimately lead to a better response from the language mod

<< FORMATTING >>
Return a markdown code snippet with a JSON object formatted to loo
```json
{{{{
    "destination": string \ "DEFAULT" or name of the prompt to use
    "next_inputs": string \ a potentially modified version of the
}}}}
```

REMEMBER: The value of “destination” MUST match one of \
the candidate prompts listed below.\
If “destination” does not fit any of the specified prompts, set it
REMEMBER: "next_inputs" can just be the original input \
if you don't think any modifications are needed.

<< CANDIDATE PROMPTS >>
{destinations}

<< INPUT >>
{{input}}

<< OUTPUT (remember to include the ```json)>>"""
```

```
router_template = MULTI_PROMPT_ROUTER_TEMPLATE.format(
    destinations=destinations_str
)
router_prompt = PromptTemplate(
    template=router_template,
    input_variables=["input"],
    output_parser=RouterOutputParser(),
)

router_chain = LLMRouterChain.from_llm(llm, router_prompt)
```

Router Chain

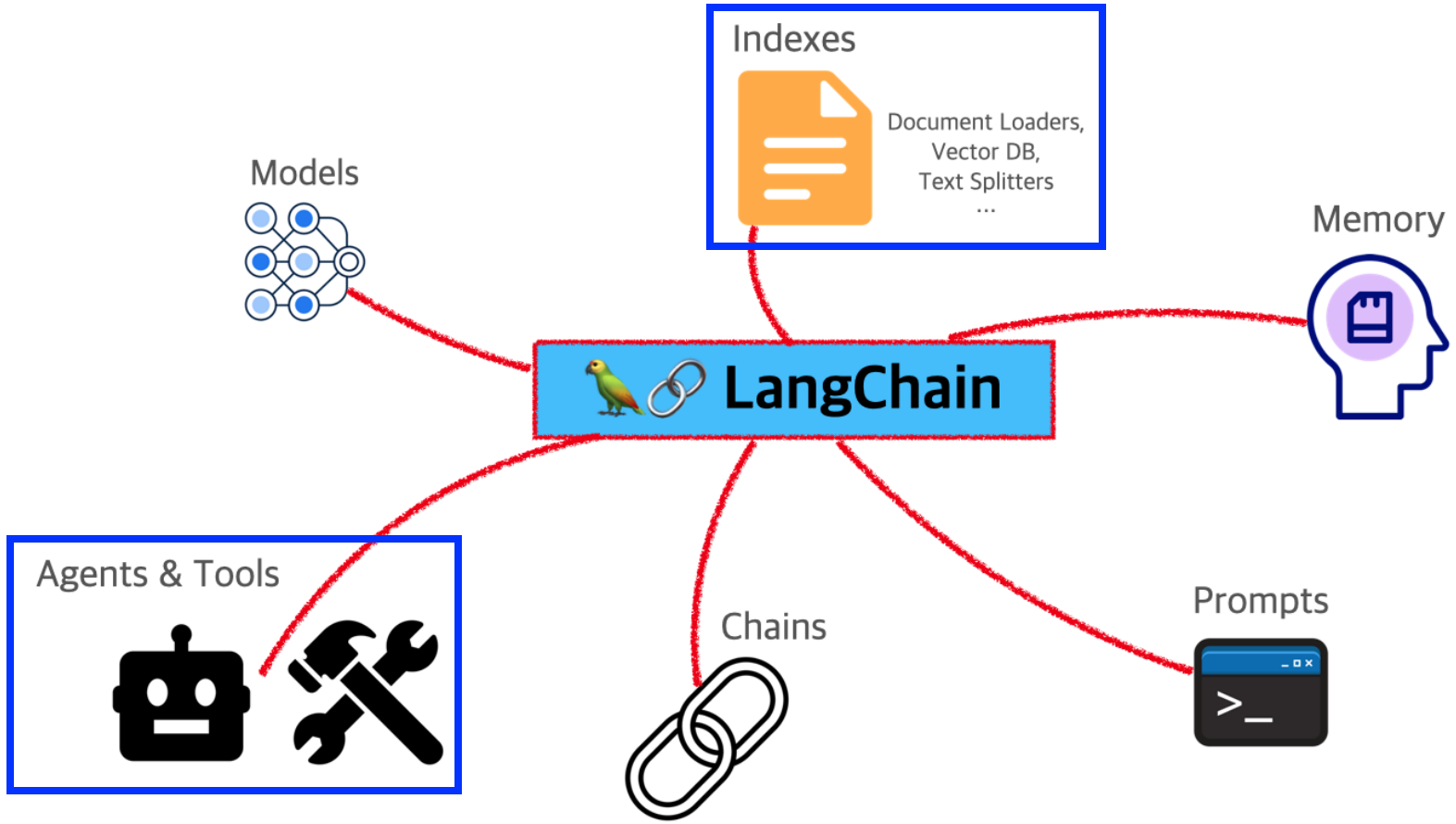
```
MultiPromptChain(router_chain=router_chain,
                 destination_chains=destination_chains,
                 default_chain=default_chain, verbose=True)
```

```
chain.run("What is black body radiation?")
```

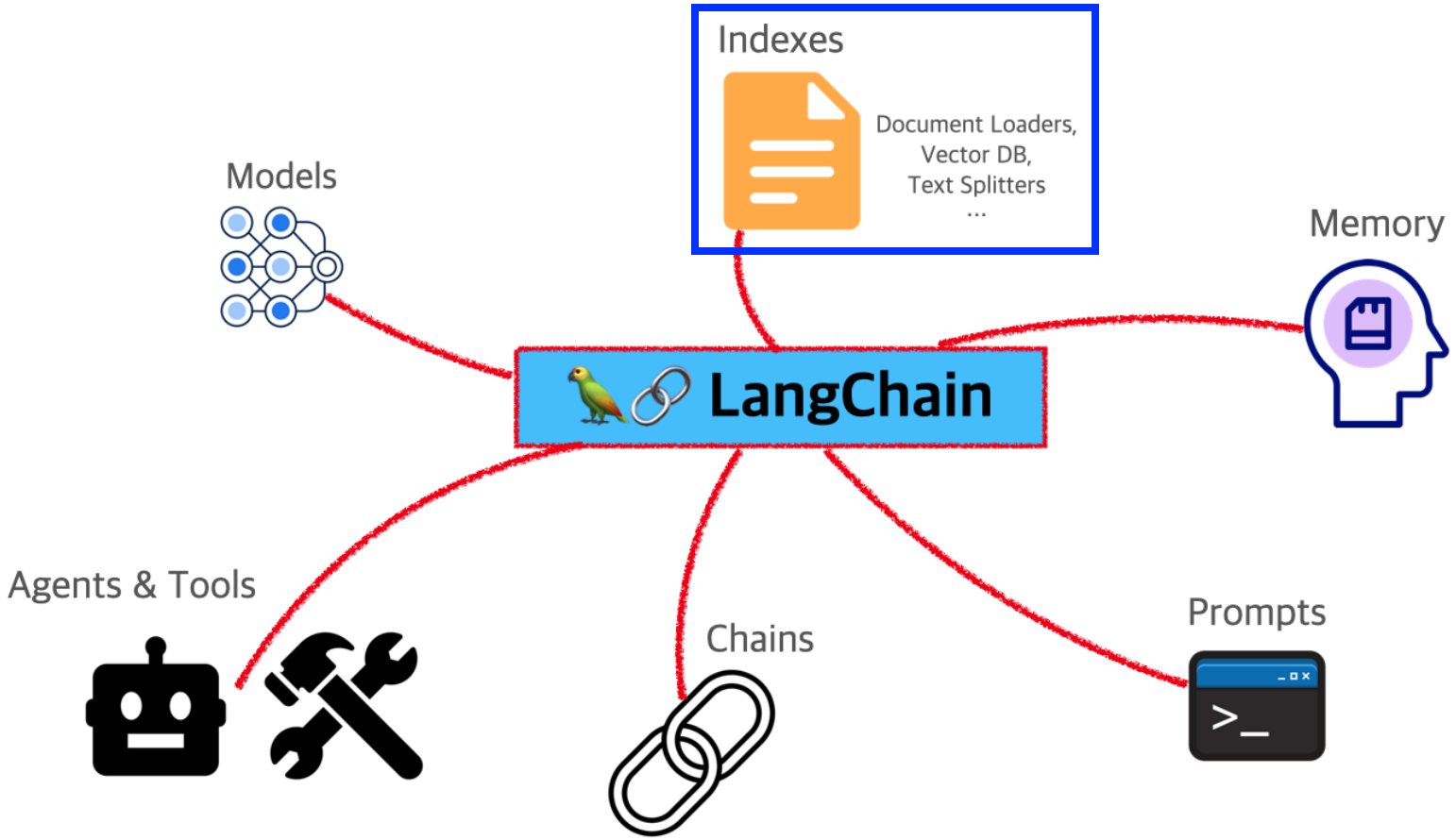
```
> Entering new MultiPromptChain chain...
physics: {'input': 'What is black body radiation?'}
> Finished chain.

"Black body radiation refers to the electromagnetic radia
tion emitted by a perfect black body, which is an object
that absorbs all radiation that falls on it and emits rad
iation at all wavelengths. The radiation emitted by a bla
ck body depends only on its temperature and follows a spe
cific distribution known as Planck's law. This type of ra
diation is important in understanding the behavior of sta
rs, as well as in the development of technologies such as
incandescent light bulbs and infrared cameras."
```

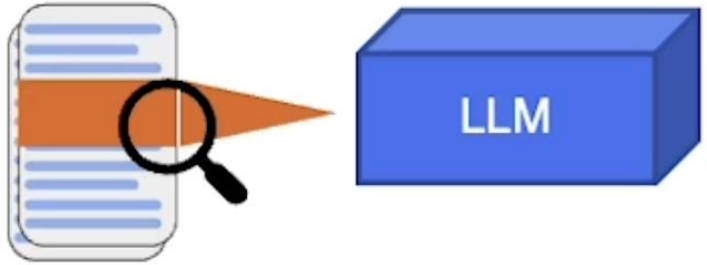
Remaining Topics



Retrieval Augmented Generation (RAG)

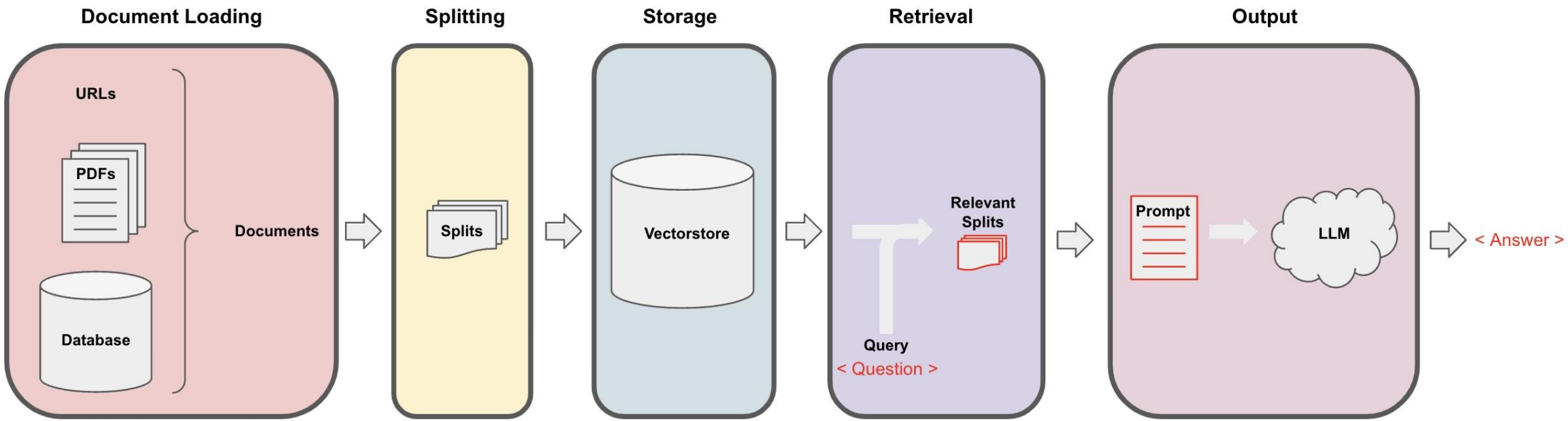


Retrieval Augmented Generation (RAG)

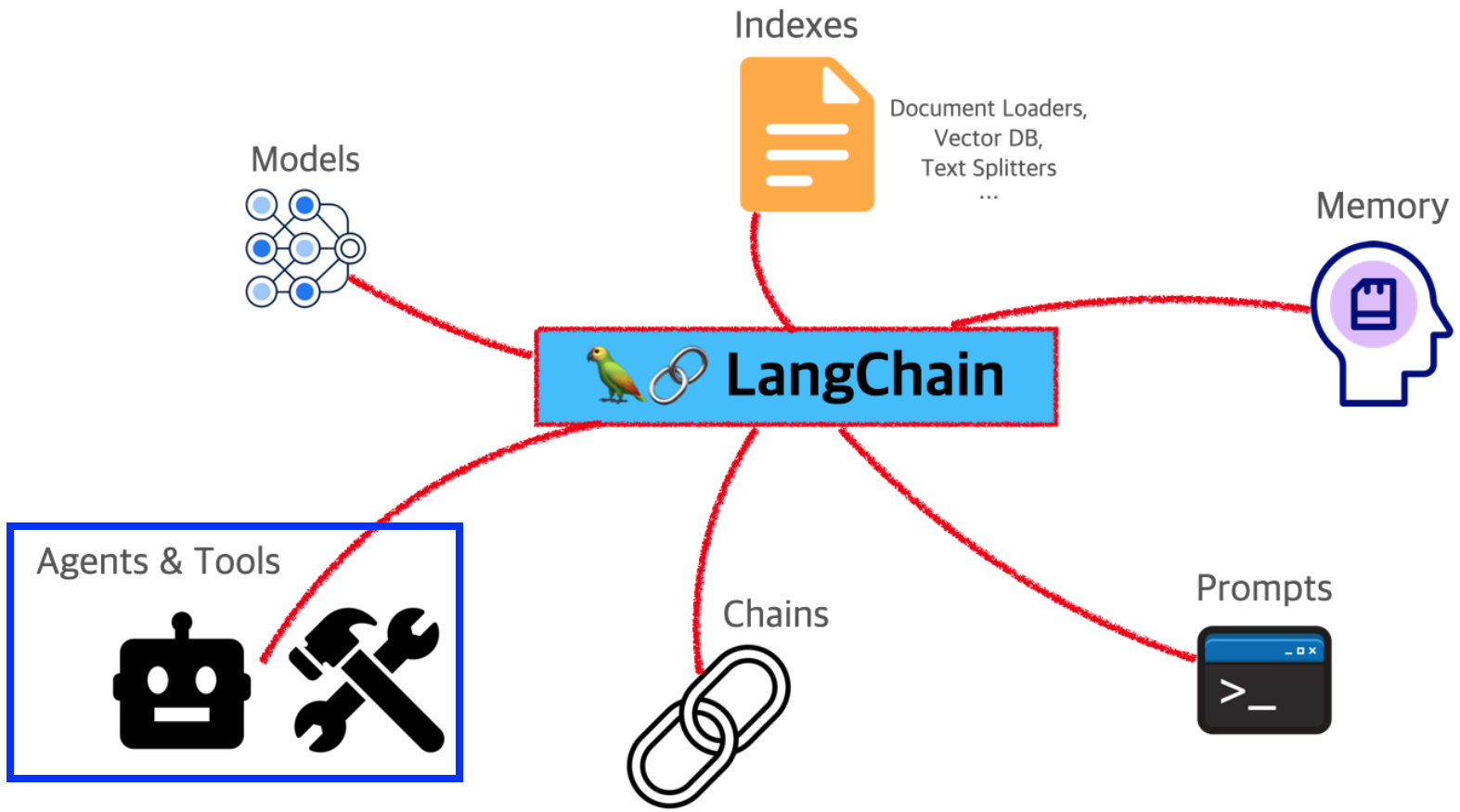


LLM's can only inspect a few thousand words at a time.

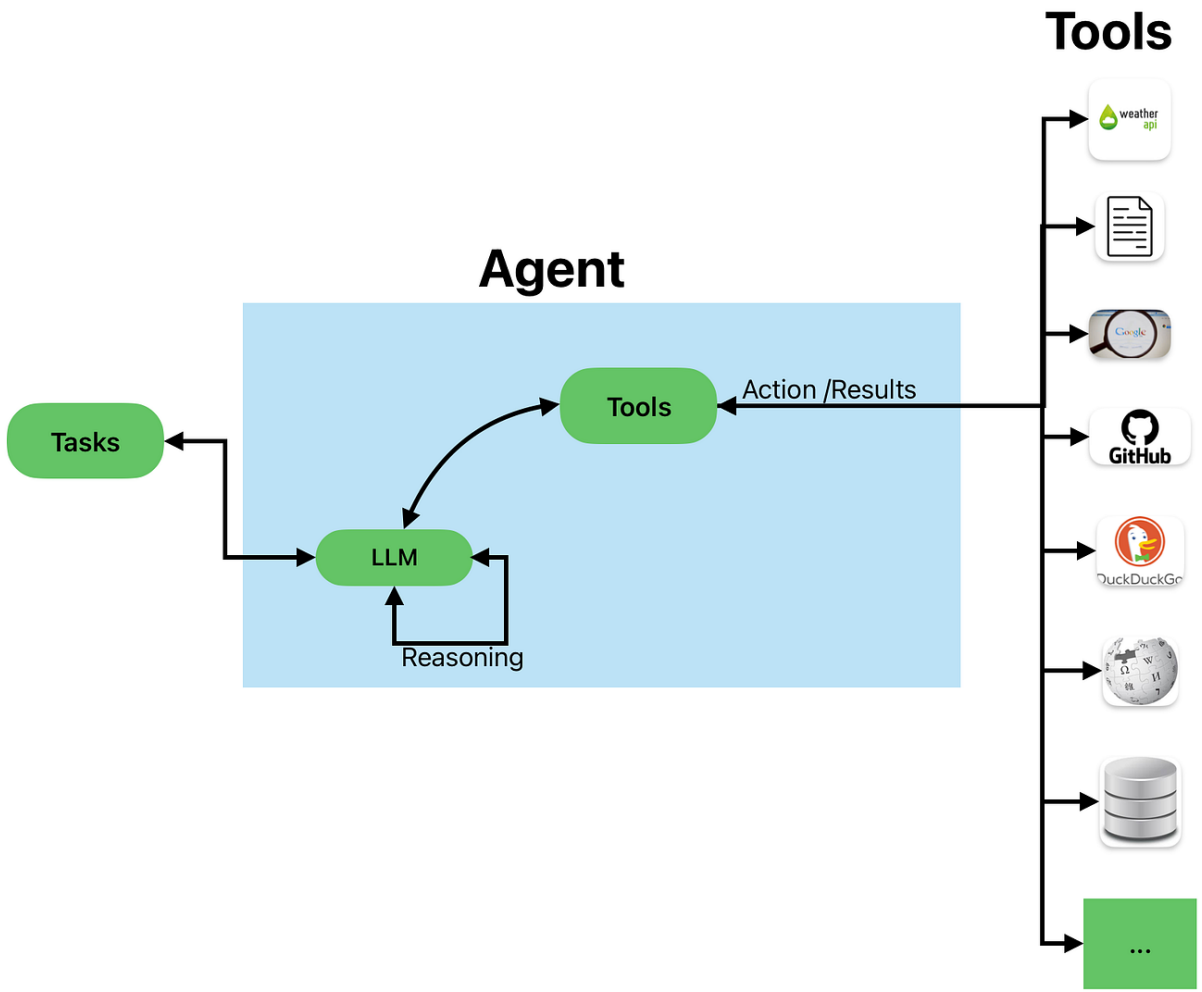
Retrieval Augmented Generation (RAG)



LLM Agents


Indexes
Document Loaders,
Vector DB,
Text Splitters
...
Models
Memory
LangChain
Agents & Tools
Chains
Prompts

LLM Agents


Tools
Agent
Tasks
Tools
Action /Results
LLM
Reasoning
GitHub
...

LangChain Ecosystem

