

# Build Smarter AI Apps: Empower LLMs with LangChain

## Module Cheat Sheet: Introduction to LangChain in GenAI

Package/Method	Description	Code Example
WatsonxLLM	A class from the <code>ibm_watson_machine_learning.foundation_models.extensions.langchain</code> module that creates a LangChain compatible wrapper around IBM's watsonx.ai models.	<pre>from ibm_watsonx_ai.foundation_models import Model from ibm_watson_machine_learning.foundation_models.extensions.langchain import WatsonxLLM  model_id = 'mistralai/mistral-7b-instruct-v0.1' parameters = {     GenParams.MAX_NEW_TOKENS: 256,     GenParams.TEMPERATURE: 0.2, } credentials = {"url": "https://us-south.ml.cloud.ibm.com",                "project_id": "skills-network"}  model = ModelInference(model_id=model_id,                        params=parameters,                        credentials=credentials,                        project_id=project_id)  mixtral_llm = WatsonxLLM(model=model) response = mixtral_llm.invoke("Who is my favorite author")</pre>
Message Types	Different types of messages that chat models can use to provide context and control the conversation. The most common message types are <code>SystemMessage</code> , <code>HumanMessage</code> , and <code>AIMessage</code> .	<pre>from langchain_core.messages import HumanMessage, SystemMessage  msg = mixtral_llm.invoke([     SystemMessage(content="You are a helpful assistant"),     HumanMessage(content="I enjoy mystery novels")])</pre>
PromptTemplate	A class from the <code>langchain_core.prompts</code> module that helps format prompts with variables. These templates allow you to define a consistent format while leaving placeholders for variables that change with each use case.	<pre>from langchain_core.prompts import PromptTemplate  prompt = PromptTemplate.from_template("Tell me a {adjective} joke about {topic}") input_ = {"adjective": "funny", "topic": "cats"}  formatted_prompt = prompt.invoke(input_)</pre>
ChatPromptTemplate	A class from the <code>langchain_core.prompts</code> module that formats a list of chat messages with variables. These templates consist of a list of message templates themselves.	<pre>from langchain_core.prompts import ChatPromptTemplate  prompt = ChatPromptTemplate.from_messages([     ("system", "You are a helpful assistant"),     ("user", "Tell me a joke about {topic}")])  input_ = {"topic": "cats"} formatted_messages = prompt.invoke(input_)</pre>

<b>MessagesPlaceholder</b>	A placeholder that allows you to add a list of messages to a specific spot in a ChatPromptTemplate. This capability is useful when you want the user to pass in a list of messages you would slot into a particular spot.	<pre> from langchain_core.prompts import MessagesPlaceholder from langchain_core.messages import HumanMessage  prompt = ChatPromptTemplate.from_messages(     ("system", "You are a helpful assistant"),     MessagesPlaceholder("msgs") )  input_ = {"msgs": [HumanMessage(content="formatted_messages = prompt.invoke(input_ </pre>
<b>JsonOutputParser</b>	A parser that allows users to specify an arbitrary JSON schema and query LLMs for outputs that conform to that schema. A parser is useful for obtaining structured data from LLMs.	<pre> from langchain_core.output_parsers import JsonOutputParser from langchain_core.pydantic_v1 import BaseModel  class Joke(BaseModel):     setup: str = Field(description="question")     punchline: str = Field(description="punchline")  output_parser = JsonOutputParser(pydantic_model=Joke)  format_instructions = output_parser.get_format_instructions() prompt = PromptTemplate(     template="Answer the user query.\n{format_instructions}\n{query}",     input_variables=["query"],     partial_variables={"format_instructions": format_instructions} )  chain = prompt   mixtral_llm   output_parser </pre>
<b>CommaSeparatedListOutputParser</b>	A parser used to return a list of comma-separated items. This parser converts the LLM's response into a Python list.	<pre> from langchain.output_parsers import CommaSeparatedListOutputParser  output_parser = CommaSeparatedListOutputParser()  format_instructions = output_parser.get_format_instructions() prompt = PromptTemplate(     template="Answer the user query. {format_instructions}\n{subject}",     input_variables=["subject"],     partial_variables={"format_instructions": format_instructions} )  chain = prompt   mixtral_llm   output_parser result = chain.invoke({"subject": "ice cream"}) </pre>
<b>Document</b>	A class from the langchain_core.documents module that contains information about some data. This class has the following two attributes: page_content (the content of the document) and metadata (arbitrary metadata associated with the document).	<pre> from langchain_core.documents import Document  doc = Document(     page_content="Python is an interpreted, high-level, general-purpose programming language. Python's design philosophy emphasizes code readability with the use of significant whitespace.",     metadata={         "my_document_id": 234234,         "my_document_source": "About Python",         "my_document_create_time": 1680000000     } ) </pre>

<b>PyPDFLoader</b>	A document loader from the langchain_community.document_loaders that loads PDFs into Document objects. You can use this document loader to extract text content from PDF files.	<pre> from langchain_community.document_loader  loader = PyPDFLoader("path/to/document.p documents = loader.load() </pre>
<b>WebBaseLoader</b>	A document loader from the langchain_community.document_loaders that loads content from websites into Document objects. You can use this document loader to extract text content from web pages.	<pre> from langchain_community.document_loader  loader = WebBaseLoader("https://python.l web_data = loader.load() </pre>
<b>CharacterTextSplitter</b>	A text splitter from langchain.text_splitter that splits text into chunks based on characters. This splitter is useful for breaking long documents into smaller, more manageable chunks for processing with LLMs.	<pre> from langchain.text_splitter import Char  text_splitter = CharacterTextSplitter(     chunk_size=200, # Maximum size of c     chunk_overlap=20, # Number of chara     separator="\n" # Character to split ) chunks = text_splitter.split_documents(c </pre>
<b>RecursiveCharacterTextSplitter</b>	A text splitter from langchain.text_splitter that splits text recursively based on a list of separators. This splitter tries to split on the first separator, then the second separator, and any subsequent separators, until the chunks of text attain the specified size.	<pre> from langchain.text_splitter import Recl  text_splitter = RecursiveCharacterTextSp     chunk_size=500,     chunk_overlap=50,     separators=["\n\n", "\n", ". ", " ", ) chunks = text_splitter.split_documents(c </pre>
<b>WatsonxEmbeddings</b>	A class from langchain_ibm that creates embeddings (vector representations) of text using IBM's watsonx.ai embedding models. You can use these embeddings for semantic search and other vector-based operations.	<pre> from langchain_ibm import WatsonxEmbeddi from ibm_watsonx_ai.metanames import Emk  embed_params = {     EmbedTextParamsMetaNames.TRUNCATE_IN     EmbedTextParamsMetaNames.RETURN_OPTI }  watsonx_embedding = WatsonxEmbeddings(     model_id="ibm/slate-125m-english-rtr     url="https://us-south.ml.cloud.ibm.c     project_id="skills-network",     params=embed_params, ) </pre>

<b>Chroma</b>	A vector store from langchain.vectorstores that stores embeddings and provides methods for similarity search. You can use Chroma for storing and retrieving documents based on semantic similarity.	<pre> from langchain.vectorstores import Chroma  // Create a vector store from documents docsearch = Chroma.from_documents(chunks, embedding_function)  // Perform a similarity search query = "Langchain" docs = docsearch.similarity_search(query) </pre>
<b>Retrievers</b>	Interfaces that return documents given an unstructured query. Retrievers accept a string query as input and return a list of Document objects as output. You can use vector stores as the backbone of a retriever.	<pre> # Convert a vector store to a retriever retriever = docsearch.as_retriever()  // Retrieve documents docs = retriever.invoke("Langchain") </pre>
<b>ParentDocumentRetriever</b>	A retriever from langchain.retrievers that splits documents into small chunks for embedding but returns the parent documents during retrieval. This retriever balances accurate embeddings with context preservation.	<pre> from langchain.retrievers import ParentDocumentRetriever from langchain.storage import InMemoryStore  parent_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0) child_splitter = CharacterTextSplitter(chunk_size=500, chunk_overlap=0)  vectorstore = Chroma(     collection_name="split_parents",     embedding_function=embedding_function, )  store = InMemoryStore()  retriever = ParentDocumentRetriever(     vectorstore=vectorstore,     docstore=store,     child_splitter=child_splitter,     parent_splitter=parent_splitter, )  retriever.add_documents(documents) retrieved_docs = retriever.invoke("Langchain") </pre>
<b>RetrievalQA</b>	A chain from langchain.chains that answers questions based on retrieved documents. The RetrievalQA chain combines a retriever with an LLM to generate answers based on the retrieved context.	<pre> from langchain.chains import RetrievalQA  qa = RetrievalQA.from_chain_type(     llm=mixtral_llm,     chain_type="stuff",     retriever=docsearch.as_retriever(),     return_source_documents=False )  query = "what is this paper discussing?" answer = qa.invoke(query) </pre>

<b>ChatMessageHistory</b>	A lightweight wrapper from langchain.memory that provides convenient methods for saving HumanMessages, AIMessages, and then fetching them all. You can use the ChatMessageHistory wrapper to maintain conversation history.	<pre> from langchain.memory import ChatMessageHistory history = ChatMessageHistory()  history.add_ai_message("hi!") history.add_user_message("what is the capital of France?")  // Access the messages history.messages  // Generate a response using the history ai_response = mixtral_llm.invoke(history.messages) </pre>
<b>ConversationBufferMemory</b>	A memory module from langchain.memory that allows for the storage of messages and conversation history. You can use this memory module with conversation chains to maintain context across multiple interactions.	<pre> from langchain.memory import ConversationBufferMemory from langchain.chains import ConversationChain  conversation = ConversationChain(     llm=mixtral_llm,     verbose=True,     memory=ConversationBufferMemory() )  response = conversation.invoke(input="Hello, my name is John.") </pre>
<b>LLMChain</b>	A basic chain from langchain.chains that combines a prompt template with an LLM. It's the simplest form of chain in LangChain.	<pre> from langchain.chains import LLMChain from langchain.prompts import PromptTemplate  template = """Your job is to come up with a recipe for {location}"""  YOUR RESPONSE:  """ prompt_template = PromptTemplate(template=template)  location_chain = LLMChain(     llm=mixtral_llm,     prompt=prompt_template,     output_key='meal' )  result = location_chain.invoke(input="Paris") </pre>
<b>SequentialChain</b>	A chain from langchain.chains that combines multiple chains in sequence, where the output of one chain becomes the input for the next chain. SequentialChain is useful for multi-step processing.	<pre> from langchain.chains import SequentialChain from langchain.prompts import PromptTemplate from langchain.llms import Mixtral  // First chain - gets a meal based on location location_chain = LLMChain(     llm=mixtral_llm,     prompt=location_prompt_template,     output_key='meal' )  // Second chain - gets a recipe based on the meal dish_chain = LLMChain(     llm=mixtral_llm,     prompt=dish_prompt_template,     output_key='recipe' )  // Third chain - estimates cooking time recipe_chain = LLMChain(     llm=mixtral_llm,     prompt=recipe_prompt_template,     output_key='time' )  // Sequential chain chain = SequentialChain(     chains=[location_chain, dish_chain, recipe_chain],     input_variables=['location'],     output_variables=['meal', 'recipe', 'time'] ) </pre>

		<pre>// Combine into sequential chain overall_chain = SequentialChain(     chains=[location_chain, dish_chain,             input_variables=['location'],             output_variables=['meal', 'recipe',                              verbose=True                              )     ] )</pre>
<b>RunnablePassthrough</b>	A component from langchain_core.runnables that allows function chaining to use the 'assign' method, enabling structured multi-step processing.	<pre>from langchain_core.runnables import RunnablePassthrough  // Create each individual chain with the location_chain_lcel = (     PromptTemplate.from_template(location_prompt),     mixtral_llm,     StrOutputParser() )  dish_chain_lcel = (     PromptTemplate.from_template(dish_prompt),     mixtral_llm,     StrOutputParser() )  time_chain_lcel = (     PromptTemplate.from_template(time_prompt),     mixtral_llm,     StrOutputParser() )  overall_chain_lcel = (     RunnablePassthrough.assign(meal=lambda x: dish_chain_lcel.invoke(x['location']),                               recipe=lambda x: time_chain_lcel.invoke(x['meal']),                               time=lambda x: time_chain_lcel.invoke(x['recipe'])),     RunnablePassthrough.assign(recipe=lambda x: time_chain_lcel.invoke(x['recipe']),                               time=lambda x: time_chain_lcel.invoke(x['recipe'])), )  // Run the chain result = overall_chain_lcel.invoke({"location": "New York City"}) pprint(result)</pre>
<b>Tool</b>	A class from langchain_core.tools that represents an interface that an agent, chain, or LLM can use to interact with the world. Tools perform specific tasks like calculations and data retrieval.	<pre>from langchain_core.tools import Tool from langchain_experimental.utilities import PythonREPL  python_repl = PythonREPL()  python_calculator = Tool(     name="Python Calculator",     func=python_repl.run,     description="Useful for when you need to perform calculations." )  result = python_calculator.invoke("a = 3 + 4")</pre>
<b>@tool decorator</b>	A decorator from langchain.tools that simplifies the creation of custom tools. This tool automatically converts a function into a Tool object.	<pre>from langchain.tools import tool  @tool def search_weather(location: str):     """Search for the current weather in {location}.     # In a real application, this function would use an API to fetch weather data.     return f"The weather in {location} is {weather}." </pre>

<b>create_react_agent</b>	A function from langchain.agents that creates an agent following the ReAct (Reasoning + Acting) framework. This function takes an LLM, a list of tools, and a prompt template as input and returns an agent that can reason and select tools to accomplish tasks.	<pre>from langchain.agents import create_react_agent agent = create_react_agent(     llm=mixtral_llm,     tools=tools,     prompt=prompt )</pre>
<b>AgentExecutor</b>	A class from langchain.agents that manages the execution flow of an agent. This class handles the orchestration between the agent's reasoning and the actual tool execution.	<pre>from langchain.agents import AgentExecutor agent_executor = AgentExecutor(     agent=agent,     tools=tools,     verbose=True,     handle_parsing_errors=True )  result = agent_executor.invoke({"input":</pre>

Author

Hailey Quach



Skills Network