LangChain Expression Language (LCEL)

DEVELOPING LLM APPLICATIONS WITH LANGCHAIN

Jonathan Bennion
AI Engineer & LangChain Contributor

datacamp

Why use LCEL over the previous syntax?

- Part of the LangChain toolkit
- Easier chaining of prompt, model, and retrieval components
- Effective for production environments
- Integrate nicely with **LangSmith** and **LangServe**

Basic LCEL components in a chain

```
from langchain.chat_models import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate

model = ChatOpenAI(openai_api_key=openai_api_key)
prompt = ChatPromptTemplate.from_template("You are a helpful personal assistant.
    Answer the following question: {question}")

chain = prompt | model

print(chain.invoke({"question": "Can you still have fun, Wilson?"}))
```

```
AIMessage(content="As an AI, I don't have personal experiences or emotions like humans do [...])
```

Calling a chain created with LCEL

- **Streaming**

```
for chunk in chain.stream({"question": "What's shaking on Shakedown Street?"}):
    print(chunk.content)
```

- **Batching**

```
inputs = [{"question": "What's shaking on Shakedown Street?"},
          {"question": "Where is the heart of town?"}]

results = chain.batch(inputs)
for result in results:
    print(result.content)
```

Runnables in LCEL

Runnables: functions or actions executed during the expression

Examples:

- `RunnablePassThrough` : pass inputs to the model
- `RunnableLambda` : transform inputs
- `RunnableMap` : processing inputs in parallel

RAG Operations with LCEL

```
from langchain_core.runnables import RunnablePassthrough
from langchain.schema.output_parser import StrOutputParser

model = ChatOpenAI(openai_api_key=openai_api_key, temperature=0)
vectorstore = Chroma.from_texts(["Nothing is shaking on Shakedown Street."],
    embedding=OpenAIEmbeddings(openai_api_key=openai_api_key))
retriever = vectorstore.as_retriever()

template = """Answer the question based on the context:{context}. Question: {question}"""
prompt = ChatPromptTemplate.from_template(template)

chain = ({"context": retriever, "question": RunnablePassthrough()} | prompt | model | StrOutputParser())
chain.invoke("What is shaking on Shakedown Street?")
```

```
Nothing is shaking on Shakedown Street.
```

Let's practice!

DEVELOPING LLM APPLICATIONS WITH LANGCHAIN

Implementing functional LangChain chains

DEVELOPING LLM APPLICATIONS WITH LANGCHAIN

Jonathan Bennion
AI Engineer & LangChain Contributor

datacamp

Chain categories

- **Generation** chains
  - Example: `ChatOpenAI`, `ChatAnthropic`
- **Retrieval** chains
  - Example: `WikipediaRetriever`, `Chroma`
- **Preprocessing** chains
  - Example: `StrOutputParser`

Sequential chains

- **Sequential chains:** output from one call becomes the input for another call

```
bartender_prompt = PromptTemplate.from_template(
    """You are an expert bartender. Mention the most popular drink in your region. question: {question}. Your answer: """)
translation_prompt = PromptTemplate.from_template(
    """You are an expert translator. Translate an order of the drink in the language native to drink. Item: {answer}. Your translation:""")

llm = ChatOpenAI(openai_api_key=openai_api_key)

chain = ({"answer": bartender_prompt | llm | StrOutputParser()}
         | translation_prompt
         | llm
         | StrOutputParser())

chain.invoke({"question": "I am in Southern Portugal, what's good to drink?"})
```

```
'O item mais popular no sul de Portugal é definitivamente a "Caipirinha". É um coquetel refrescante feito com cachaça, limão, açúcar [...]
```

Manipulating values with sequential chains

```
prompt1 = ChatPromptTemplate.from_template("Generate a random number")
prompt2 = ChatPromptTemplate.from_template("Multiply {number} by 2")

llm = ChatOpenAI(openai_api_key=openai_api_key)
chain1 = prompt1 | llm
chain2 = prompt2 | llm

response1 = chain1.invoke({})
response2 = chain2.invoke({"number": response1.content})

print("Generated number:", response1.content)
print("Result of multiplication:", response2.content)
```

```
Generated number: The random number is 57.
Result of multiplication: The result of multiplying the random number 57 by 2 is 114.
```

RunnablePassthrough in chains

- `RunnablePassthrough()` : passing values between chains

```
from langchain_core.runnables import RunnablePassthrough

q_response = (
    ChatPromptTemplate.from_template("You are a helpful assistant. Answer the question: {input}")
    | ChatOpenAI(openai_api_key=openai_api_key)
    | {"response": RunnablePassthrough() | StrOutputParser()})

contrarian_response = (
    ChatPromptTemplate.from_template(
        "You are a contrarian. Describe the most powerful opposing perspective for {response}")
    | ChatOpenAI(openai_api_key=openai_api_key)
    | StrOutputParser())
```

RunnablePassthrough in chains

```
final_chain = (
  {"response": q_response, "opposing_response": contrarian_response}
    | ChatPromptTemplate.from_messages(
        [("ai", "{response}"),
         ("human", "Response:\n{response}\n\nOpposing response:\n{opposing_response}"),
         ("system", "Summarize the original response and an opposing response.")])
    | ChatOpenAI(openai_api_key=openai_api_key)
    | StrOutputParser()
)

print(final_chain.invoke({"input": "What is the best ice cream?", "response": "", "opposing_response": ""}))
```

```
Original response:
The best ice cream is subjective and can vary based on personal preferences. Some popular flavors [...]

Opposing response:
[...]
```

Let's practice!

DEVELOPING LLM APPLICATIONS WITH LANGCHAIN

An introduction to LangChain agents

DEVELOPING LLM APPLICATIONS WITH LANGCHAIN

Jonathan Bennion
AI Engineer & LangChain Contributor

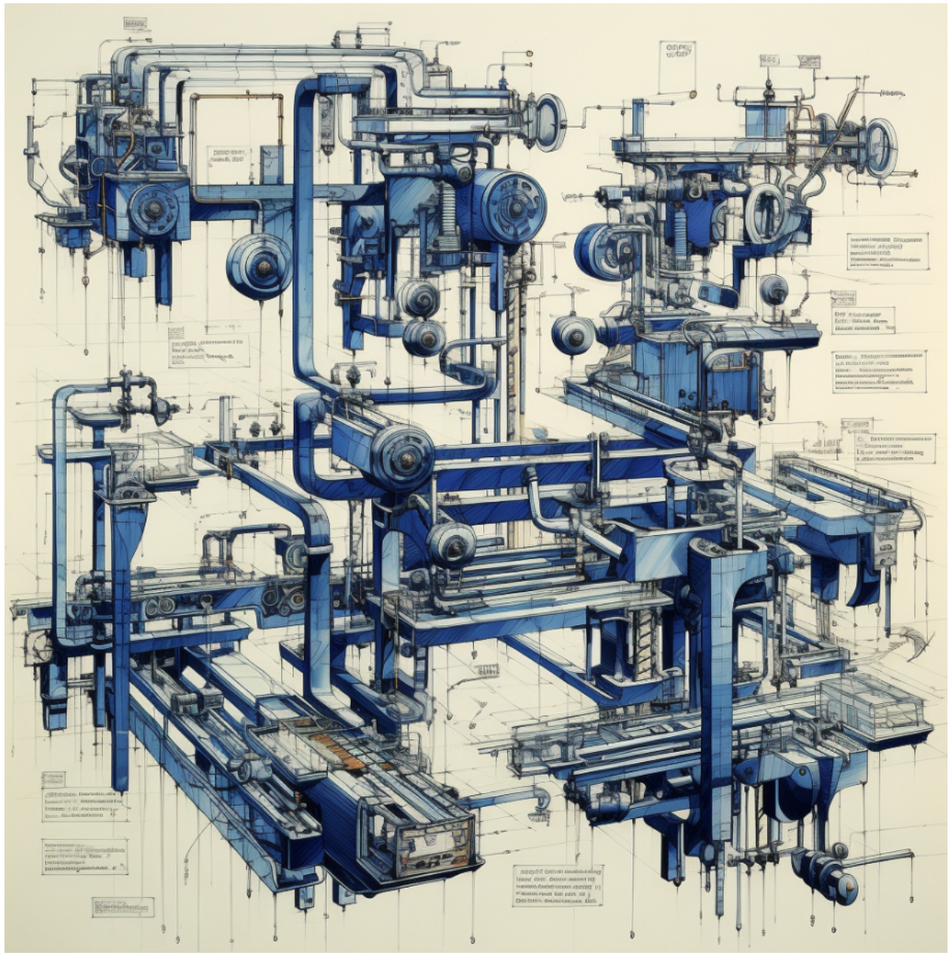
datacamp

What are agents?

Agents: use language models to decide which actions to take

Tools: functions used by the agent to interact with the system (utilities, chains, more agents)

- **Now** → built-in tools
- **Chapter 4** → custom tools!



Agent Types

Agent Type	Intended Model Type	Supports Chat History	Supports Multi-Input Tools	Supports Parallel Function Calling	Required Model Params	When to Use
OpenAI Tools	Chat	✅	✅	✅	`tools`	If you are using a recent OpenAI model (`1106` onwards)
OpenAI Functions	Chat	✅	✅		`functions`	If you are using an OpenAI model, or an open-source model that has been finetuned for function calling and exposes the same `functions` parameters as OpenAI
XML	LLM	✅				If you are using Anthropic models, or other models good at XML
Structured Chat	Chat	✅	✅			If you need to support tools with multiple inputs
JSON Chat	Chat	✅				If you are using a model good at JSON
ReAct	LLM	✅				If you are using a simple model
Self Ask With Search	LLM					If you are using a simple model and only have one search tool

[1] https://python.langchain.com/docs/modules/agents/agent_types/

Primary agent components

- User input in the form of a prompt
- Definition for handling the intermediate steps
- Tools and model behavior definition
- Output parser

Zero-Shot ReAct agent

```
from langchain.agents import initialize_agent, AgentType, load_tools

llm = OpenAI(model_name="gpt-3.5-turbo-instruct", temperature=0, openai_api_key=openai_api_key)
tools = load_tools(["llm-math"], llm=llm)

agent = initialize_agent(tools, llm, agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=True)
agent.run("What is 10 multiplied by 50?")
```

```
> Entering new AgentExecutor chain...
 I need to multiply two numbers
Action: Calculator
Action Input: 10 * 50
Observation: Answer: 500
Thought: I now know the final answer
Final Answer: 10 multiplied by 50 is 500.
```

Let's practice!

DEVELOPING LLM APPLICATIONS WITH LANGCHAIN