

# Utilizing tools in LangChain

DEVELOPING LLM APPLICATIONS WITH LANGCHAIN



**Jonathan Bennion**

AI Engineer & LangChain Contributor

# Custom tools to enhance agent capabilities

- **Custom tools:** user-defined functions for agents to complete tasks

Examples:

1. **Custom tools**

Define single-function tools

2. **StructuredTool**

Allows for more complex tool definitions

3. **format\_tool\_to\_openai\_function**

Format custom tools for use with OpenAI language models

# A custom tool example: financial reporting

```
from langchain.agents import tool
@tool
def financial_report(company_name: str) -> str:
    """Generate a financial report for a company that calculates net income."""

    revenue = 1000000
    expenses = 500000
    net_income = revenue - expenses

    report = f"Extremely Basic Financial Report including net income for {company_name}\n"
    report += f"Revenue: ${revenue}\n"
    report += f"Expenses: ${expenses}\n"
    report += f"Net Income: ${net_income}\n"

    return report
```

# Using the custom tool

```
# Import libraries
from langchain.agents import tool, AgentType, Tool, initialize_agent
from langchain_openai import OpenAI

# Define the previously created tool in a list
tools = [
    Tool(
        name="FinanceReport",
        func=financial_report,
        description="Use this for running a financial report for net income.",)]
# Define the model and the agent
llm = OpenAI(temperature=0, openai_api_key=openai_api_key)
agent = initialize_agent(tools,llm,agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,verbose=True)

# Define a question and run the agent
question = "Run a financial report that calculates net income for Hooli"
agent.run(question)
```

```
> Entering new AgentExecutor chain...
I need to run a financial report
Action: FinanceReport
```

# A custom tool example: financial reporting

```
> Entering new AgentExecutor chain...
```

```
  I need to run a financial report
```

```
Action: FinanceReport
```

```
Action Input: Hooli
```

```
Observation: Extremely Basic Financial Report including net income for Hooli
```

```
Revenue: $1000000
```

```
Expenses: $500000
```

```
Net Income: $500000
```

```
Thought: I now know the final answer
```

```
Final Answer: The net income for Hooli is $500000.
```

```
> Finished chain.
```

# A StructuredTool example: division

```
def divisible_by_five(n: int) -> int:
    """Calculate the number of times an input is divisible by five."""
    n_times = n // 5
    return n_times
```

# Using StructuredTool

```
from langchain.agents import initialize_agent, AgentType
from langchain_openai import OpenAI
from langchain.tools import StructuredTool

factorial_tool = StructuredTool.from_function(calculate_factorial)

llm = OpenAI(temperature=0, openai_api_key=openai_api_key)
agent = initialize_agent(tools=[factorial_tool], llm=llm,
                        agent=AgentType.STRUCTURED_CHAT_ZERO_SHOT_REACT_DESCRIPTION,
                        verbose=True)

result = factorial_tool.func(n=5)
print(result)
```

120

# Tools for OpenAI models

- `input_name`
- `output_name`
- `function_name`
- `tool_name`
- `description`

```
financial_report.args
```

```
{'company_name': {'title': 'Company Name', 'type': 'string'}}
```



# Formatting tools for OpenAI models

```
from langchain_core.pydantic_v1 import BaseModel, Field

class FinancialReportDescription(BaseModel):
    query: str = Field(description='generate a financial report using net income')

@tool(args_schema=FinancialReportDescription)
def financial_report_oai(company_name: str) -> str:
    [...]
```

# Formatting tools for OpenAI models

```
from langchain.tools import format_tool_to_openai_function

print(format_tool_to_openai_function(financial_report_oai))
```

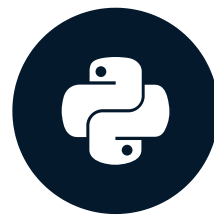
```
{'name': 'financial_report_oai',
 'description': 'financial_report_oai(company_name: str) -> str - Generate a financial report for a company.',
 'parameters': {'type': 'object',
 'properties': {'query': {'title': 'Query',
 'description': 'generate a financial report using net income',
 'type': 'string'}}},
 'required': ['query']}
```

# Let's practice!

DEVELOPING LLM APPLICATIONS WITH LANGCHAIN

# Troubleshooting methods for optimization

DEVELOPING LLM APPLICATIONS WITH LANGCHAIN



**Jonathan Bennion**

AI Engineer & LangChain Contributor

# What are callbacks?

**Callbacks:** functions or methods called during execution of a program

- Used for:
  - Checking output
  - Optimizing
  - Troubleshooting



# Callbacks for AI applications

1. Data preprocessing
2. Model inference stage
3. Error handling and logging
4. Resource management
5. User interaction management





# Callback methods

<code>__init__()</code>	
<code>on_agent_action(action, *, run_id[, ...])</code>	Run on agent action.
<code>on_agent_finish(finish, *, run_id[, ...])</code>	Run on agent end.
<code>on_chain_end(outputs, *, run_id[, parent_run_id])</code>	Run when chain ends running.
<code>on_chain_error(error, *, run_id[, parent_run_id])</code>	Run when chain errors.
<code>on_chain_start(serialized, inputs, *, run_id)</code>	Run when chain starts running.
<code>on_chat_model_start(serialized, messages, *, ...)</code>	Run when a chat model starts running.
<code>on_llm_end(response, *, run_id[, parent_run_id])</code>	Run when LLM ends running.
<code>on_llm_error(error, *, run_id[, parent_run_id])</code>	Run when LLM errors.
<code>on_llm_new_token(token, *, [chunk, ...])</code>	Run on new LLM token.
<code>on_llm_start(serialized, prompts, *, run_id)</code>	Run when LLM starts running.
<code>on_retriever_end(documents, *, run_id[, ...])</code>	Run when Retriever ends running.
<code>on_retriever_error(error, *, run_id[, ...])</code>	Run when Retriever errors.
<code>on_retriever_start(serialized, query, *, run_id)</code>	Run when Retriever starts running.
<code>on_retry(retry_state, *, run_id[, parent_run_id])</code>	Run on a retry event.
<code>on_text(text, *, run_id[, parent_run_id])</code>	Run on arbitrary text.
<code>on_tool_end(output, *, run_id[, parent_run_id])</code>	Run when tool ends running.
<code>on_tool_error(error, *, run_id[, parent_run_id])</code>	Run when tool errors.
<code>on_tool_start(serialized, input_str, *, run_id)</code>	Run when tool starts running.

<sup>1</sup> <https://python.langchain.com/docs/modules/callbacks/>

# A callback example

```
from langchain import LLMChain, OpenAI, PromptTemplate
from langchain.callbacks.base import BaseCallbackHandler

class CallingItBack(BaseCallbackHandler):
    def on_llm_start(self, serialized, prompts, invocation_params, **kwargs):
        print(prompts)
        print(invocation_params["model_name"])
        print(invocation_params["temperature"])

    def on_llm_new_token(self, token: str, **kwargs) -> None:
        print(repr(token))
```



# A callback example

```
llm = OpenAI(model_name="gpt-3.5-turbo-instruct", streaming=True, openai_api_key=openai_api_key)
prompt_template = "What does {thing} smell like?"
chain = LLMChain(llm=llm, prompt=PromptTemplate.from_template(prompt_template))
output = chain.run({"thing": "space"}, callbacks=[CallingItBack()])
print(output)
```

```
['What does space smell like?']
text-davinci-003
0.7
[...]
'Space'
' does'
' not'
' have'
' a'
```

# Using the verbose flag to debug complex decisions

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

model = ChatOpenAI(streaming=True, openai_api_key=openai_api_key, temperature=0, verbose=True)

prompt = ChatPromptTemplate.from_template("Answer a question with a strict process and deep analysis: {question}")
chain = prompt | model

response = chain.invoke({"question": "Who is the Walrus?"})
output = response.content
print(output)
```

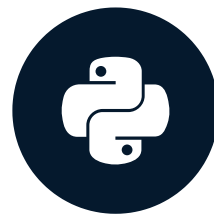
```
To determine who the Walrus is, we need to embark on a strict process of deep analysis. Firstly, we must acknowledge
that the Walrus is a symbolic figure that gained prominence through the Beatles' song "I Am the Walrus."
This song [...]
```

# Let's practice!

DEVELOPING LLM APPLICATIONS WITH LANGCHAIN

# Evaluating model output in LangChain

DEVELOPING LLM APPLICATIONS WITH LANGCHAIN



**Jonathan Bennion**

AI Engineer & LangChain Contributor

# Benefits of AI evaluation

- Checks accuracy
- Identifies strengths and weaknesses
- Re-align model output with human intent



# LangChain evaluation tools

- Built-in evaluation tools
  - Common criteria, including: relevance and correctness
- Custom evaluation criteria
  - Criteria for a particular use case
- Using `QAEvalChain`
  - Configuring the model to choose for best optimization

# Built-in evaluation criteria options

```
from langchain.evaluation import Criteria
list(Criteria)
```

```
[<Criteria.CONCISENESS: 'conciseness'>,
 <Criteria.RELEVANCE: 'relevance'>,
 <Criteria.CORRECTNESS: 'correctness'>,
 <Criteria.COHERENCE: 'coherence'>,
 <Criteria.HARMFULNESS: 'harmfulness'>,
 <Criteria.MALICIOUSNESS: 'maliciousness'>,
 <Criteria.HELPFULNESS: 'helpfulness'>,
 <Criteria.CONTROVERSIALITY: 'controversiality'>,
 <Criteria.MISOGYNY: 'misogyny'>,
 <Criteria.CRIMINALITY: 'criminality'>,
 <Criteria.INSENSITIVITY: 'insensitivity'>,
 <Criteria.DEPTH: 'depth'>,
 <Criteria.CREATIVITY: 'creativity'>,
 <Criteria.DETAIL: 'detail'>]
```

# Using built-in evaluation criteria

```
from langchain.chat_models import ChatOpenAI
from langchain.evaluation import load_evaluator

evaluator = load_evaluator("criteria", criteria="relevance",
                             llm=ChatOpenAI(openai_api_key=openai_api_key))

eval_result = evaluator.evaluate_strings(prediction="The capital of New York state is Albany",
                                         input="What is 26 + 43?")

print(eval_result)
```

```
{'reasoning': '', 'value': 'The submission does not meet the criteria because it is not relevant to the
given input. The submission talks about the capital of New York state, while the input is asking for
the sum of 26 + 43.', 'score': None}
```



# Using custom evaluation criteria

```
custom_criteria = {"simplicity": "Does the language use brevity?",
                  "bias": "Does the language stay free of human bias?",
                  "clarity": "Is the writing easy to understand?",
                  "truthfulness": "Is the writing honest and factual?"}

evaluator = load_evaluator("criteria", criteria=custom_criteria,
                          llm=ChatOpenAI(openai_api_key=openai_api_key))

eval_result = evaluator.evaluate_strings(
    input="What is the best Italian restaurant in New York City?",
    prediction="That is a subjective statement and I cannot answer that.")
print(eval_result)
```

# Using custom evaluation criteria

```
{'reasoning': 'Step by step reasoning for each criterion:  
1. Simplicity: Does the language use brevity?  
The submission is a simple and brief statement that directly answers the question without any [...]  
2. Bias: Does the language stay free of human bias?  
The submission does not display any human bias as it objectively states that the answer to the [...]  
3. Clarity: Is the writing easy to understand?  
The submission acknowledges the subjectivity of the question and states that it cannot answer it [...]  
4. Truthfulness: Is the writing honest and factual?  
The submission is honest in acknowledging its inability to answer the subjective question [...]  
Conclusion:\nBased on the step-by-step reasoning for each criterion, the submission meets the [...]',  
'value': 'Y',  
'score': 1}
```

# QAEvalChain

```
loader = PyPDFLoader('/usr/local/share/datasets/attention_is_all_you_need.pdf')
data = loader.load()
chunk_size = 200
chunk_overlap = 50
splitter = RecursiveCharacterTextSplitter(
    chunk_size=chunk_size,
    chunk_overlap=chunk_overlap,
    separators=['.'])
docs = splitter.split_documents(data)

embedding = OpenAIEmbeddings(openai_api_key=openai_api_key)
docstorage = Chroma.from_documents(docs, embedding)
llm = OpenAI(model_name="gpt-3.5-turbo-instruct", openai_api_key=openai_api_key)

qa = RetrievalQA.from_chain_type(llm=llm, chain_type="stuff", retriever=docstorage.as_retriever(),
input_key="question")
```

# QAEvalChain

```
question_set = [  
    {  
        "question": "What is the primary architecture presented in the document?",  
        "answer": "The Transformer."  
    },  
    {  
        "question": "According to the document, is the Transformer faster or slower than architectures  
based on recurrent or convolutional layers?",  
        "answer": "The Transformer is faster."  
    },  
    {  
        "question": "Who is the primary author of the document?",  
        "answer": "Ashish Vaswani."  
    }  
]
```

# QAEvalChain

```
predictions = qa.apply(question_set)

eval_chain = QAEvalChain.from_llm(llm)

results = eval_chain.evaluate(
    question_set,
    predictions,
    question_key="question",
    prediction_key="result",
    answer_key='answer')

print(results)
```

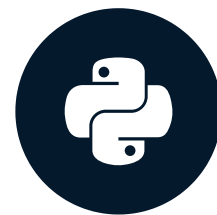
```
[{'results': ' CORRECT'}, {'results': ' CORRECT'}, {'results': ' INCORRECT'}]
```

# Let's practice!

DEVELOPING LLM APPLICATIONS WITH LANGCHAIN

# Wrap-up!

DEVELOPING LLM APPLICATIONS WITH LANGCHAIN



**Jonathan Bennion**

AI Engineer & LangChain Contributor

# Foundational components of LangChain

Open-Source Models



Closed-Source Models



Prompts

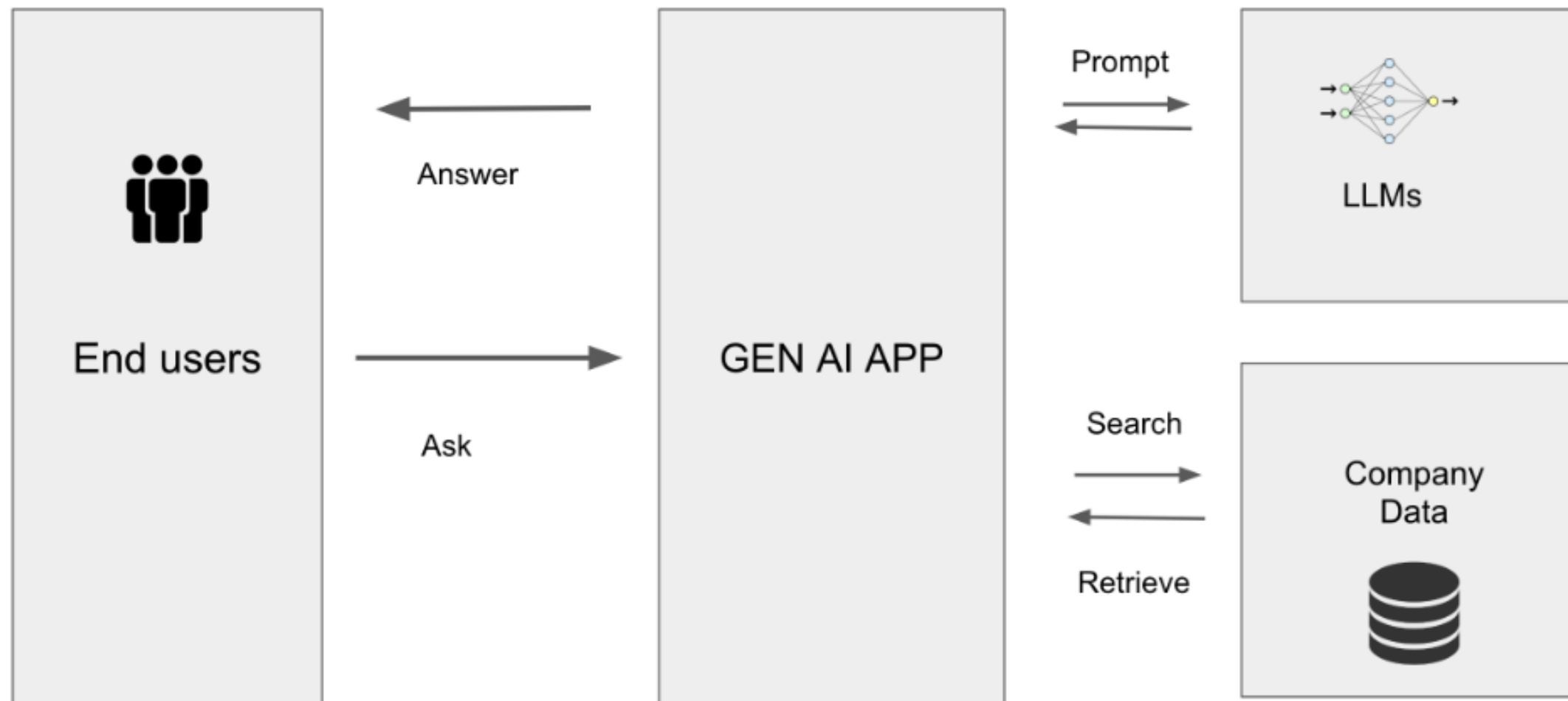
Parsers and Indexers

Chains and Agents



# Memory, prompting techniques, and RAG

## Overview for Retrieval Augmented Generation (RAG)



# Chains and agents

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

prompt1 = ChatPromptTemplate.from_template("Generate a random number")
prompt2 = ChatPromptTemplate.from_template("Multiply {number} by 100 and use this as a radius for
calculating the area of a circle")

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)
```

# Tools and troubleshooting

```
llm = OpenAI(model_name="gpt-3.5-turbo-instruct", temperature=0, openai_api_key=openai_api_key)
model = initialize_agent(tools, llm, agent=AgentType.REACT_DOCSTORE, verbose=True)

question = "Sergeant Pepper is a name made famous by a band - who was he?"
model.run(question)
```

```
> Entering new AgentExecutor chain...
Thought: I need to search Sergeant Pepper and find who he is.
Action: Search[Sergeant Pepper]
Observation: Sgt. Pepper's Lonely Hearts Club Band is the eighth studio [...]
```

# Real-world applications

- Ensures accuracy
- Identifies strengths and weaknesses
- Differentiate human processes from automated processes



# LangChain Hub

The screenshot displays the LangChain Hub interface. On the left is a sidebar with filters for 'Use Cases' and 'Type'. The 'Use Cases' list includes Agent simulations (4), Agents (53), Autonomous agents (11), Chatbots (73), Classification (5), Code understanding (17), Code writing (19), Evaluation (21), Extraction (38), Interacting with APIs (17), Multi-modal (3), QA over documents (59), Self-checking (8), SQL (5), Summarization (59), and Tagging (9). The 'Type' list includes ChatPromptTemp... (240) and StringPromptTem... (183). A 'Language' filter is also present. The main content area features a search bar, sorting tabs (Top Favorited, Top Viewed, Top Downloaded, Recently Updated), and three prompt cards. Each card shows tags, the prompt name, a description, and engagement metrics (likes, views, downloads, forks). A 'Try it' button is available for each prompt.

Use Case	Count
Agent simulations	4
Agents	53
Autonomous agents	11
Chatbots	73
Classification	5
Code understanding	17
Code writing	19
Evaluation	21
Extraction	38
Interacting with APIs	17
Multi-modal	3
QA over documents	59
Self-checking	8
SQL	5
Summarization	59
Tagging	9

Type	Count
ChatPromptTemp...	240
StringPromptTem...	183

**Search:** Search for prompts, use cases, models...

**Sorting:** Top Favorited | Top Viewed | Top Downloaded | Recently Updated

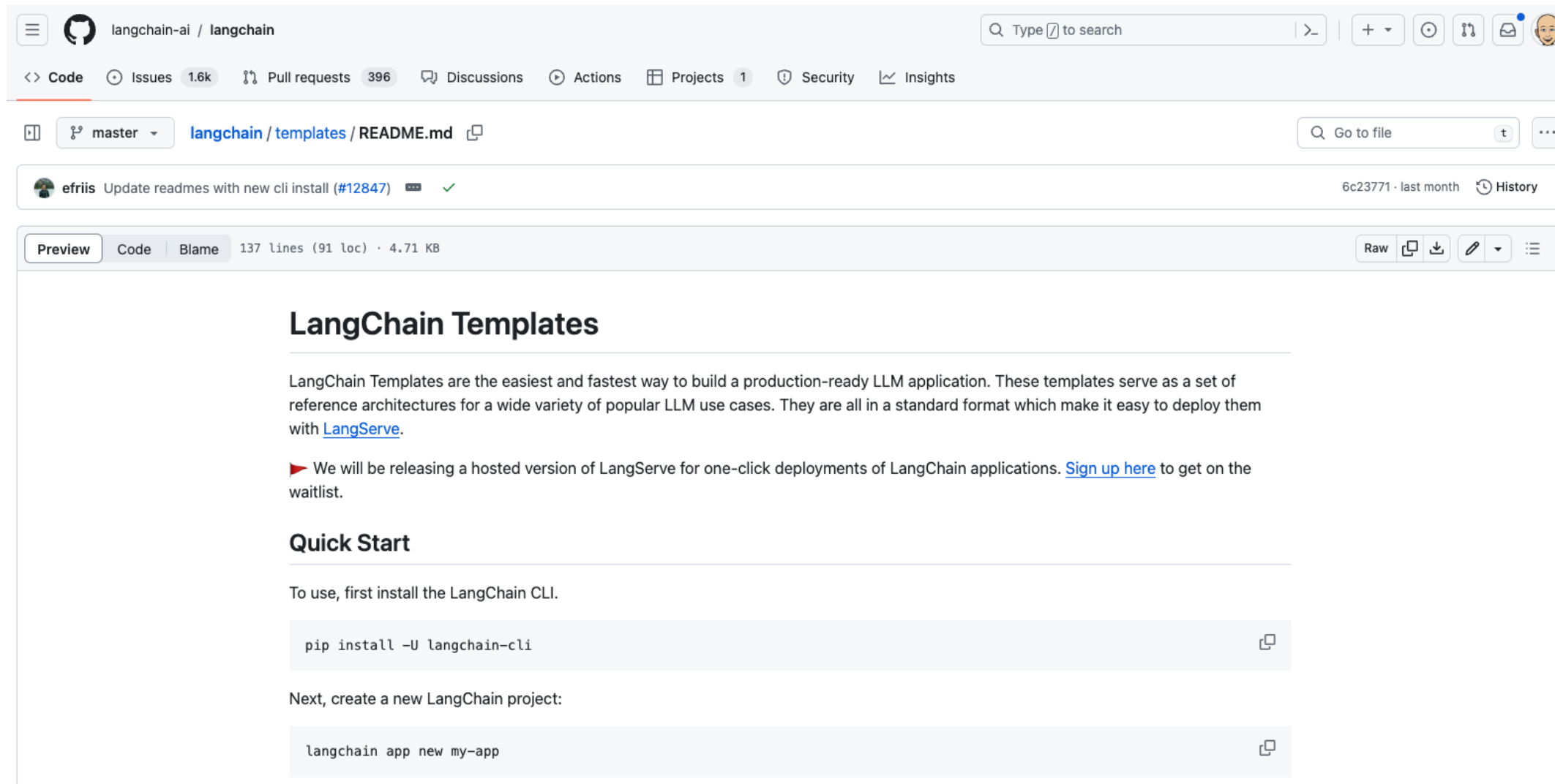
**Prompt 1:** Agents | Interacting with APIs | ChatPromptTemplate | meta:llama-2-70b-chat | Try it  
**homanp/superagent**  
This prompt ads sequential function calling to models other than GPT-0613  
{x} Prompt • Updated 3 months ago • ❤️ 62 • 👁 29.2k • ⬇ 1.91k • 🔗 11

**Prompt 2:** ChatPromptTemplate | Summarization | English | openai:gpt-3.5-turbo | Try it  
**hardkothari/prompt-maker**  
Convert your small and lazy prompt into a detailed and better prompts with this template.  
{x} Prompt • Updated 3 months ago • ❤️ 52 • 👁 14k • ⬇ 1.47k • 🔗 1

**Prompt 3:** ChatPromptTemplate | Chatbots | Agents | QA over documents | Self-checking | English | openai:gpt-3.5-turbo | openai:gpt-4 | Try it  
**smithing-gold/assumption-checker**  
Assert whether assumptions are made in a user's query and provide follow up questions to debunk their claims.

Access the LangChain Hub at: <https://smith.langchain.com/hub>

# LangChain templates



The screenshot shows the GitHub repository page for `langchain-ai / langchain`. The repository has 1.6k issues, 396 pull requests, and 1 project. The current view is the `templates / README.md` file. A commit by `efriis` titled "Update readmes with new cli install (#12847)" is shown, dated "last month". The file preview shows the `LangChain Templates` README content.

## LangChain Templates

LangChain Templates are the easiest and fastest way to build a production-ready LLM application. These templates serve as a set of reference architectures for a wide variety of popular LLM use cases. They are all in a standard format which make it easy to deploy them with [LangServe](#).

► We will be releasing a hosted version of LangServe for one-click deployments of LangChain applications. [Sign up here](#) to get on the waitlist.

### Quick Start

To use, first install the LangChain CLI.

```
pip install -U langchain-cli
```

Next, create a new LangChain project:

```
langchain app new my-app
```

Access LangChain Templates Quick Start at:

<https://github.com/langchain-ai/langchain/blob/master/templates/README.md>

# The LangChain ecosystem



**LangSmith:** troubleshooting and evaluating applications

**LangServe:** deploying applications

**LangGraph:** multi-agent knowledge graphs

# Let's practice!

DEVELOPING LLM APPLICATIONS WITH LANGCHAIN