Build your Personal Assistant with Agents and Tools

LLMs alone suffer from not being able to access external or real-time data. Learn how to build your personal assistant using LangChain agents and Gemini by grounding it in external sources.

Benjamin Etienne · Follow

Published in Towards Data Science · 14 min read · 5 days ago

Summary:

1. The problem with LLMs
2. What are Agents, Tools and Chains ?
3. Creating a simple chat without Tools
4. Adding Tools to our chat: The Google way with Function Calling
5. Adding Tools to our chat : The Langchain way with Agents
6. Adding Memory to our Agent

1. The problem with LLMs

So you have your favorite chatbot, and you use it for your daily job to boost your productivity. It can translate text, write nice emails, tell jokes, etc. And then comes the day when your colleague comes to you and asks :

"Do you know the current exchange rate between USD and EUR ? I wonder if I should sell my EUR..."

You ask your favorite chatbot, and the answer pops :

```
I am sorry, I cannot fulfill this request.
I do not have access to real-time information, including financial data
like exchange rates.
```

What is the problem here ?

The problem is that you have stumbled on one of the shortcomings of LLMs. Large Language Models (LLMs) are powerful at solving many types of problems, such as problem solving, text summarization, generation, etc.

However, they are constrained by the following limitations:

- **They are frozen after training, leading to stale knowledge.**
- **They can't query or modify external data.**

Same way as we are using search engines every day, reading books and documents or querying databases, we would ideally want to provide this knowledge to our LLM to make it more efficient.

Fortunately, there is a way to do that: Tools and Agents.

> *Foundational models, despite their impressive text and image generation, remain constrained by their inability to interact with the outside world. Tools bridge this gap, empowering agents to interact with external data and services while unlocking a wider range of actions beyond that of the underlying model alone*

(source : Google Agents whitepaper)

Using agents and tools, we could then be able to, from our chat interface:

- retrieve data from our own documents
- read / send emails
- interact with internal databases
- perform real time Google searches
- etc.

2. What are Agents, Tools and Chains ?

An *agent* is an application which attempts to achieve a goal (or a task) by having at its disposal a set of tools and taking decisions based on its observations of the environment.

A good example of an agent could be you, for example: if you need to compute a complex mathematical operation (goal), you could use a calculator (tool #1), or a programming language (tool #2). Maybe you would choose the calculator to do a simple addition, but choose tool #2 for more complex algorithms.

Agents are therefore made of :

- A model : The brain in our agent is the LLM. It will understand the query (the goal), and browse through its tools available to select the best.
- One or more *tools* : These are functions, or APIs, that are responsible for performing a specific action (ie: retrieving the current currency rate for USD vs EUR, adding numbers, etc.)
- An orchestration process: this is how the model will behave when asked to solve a task. It is a cognitive process that defines how the model will analyze the problem, refine inputs, choose a tool, etc. Examples of such processes are ReAct, CoT (Chain of Thought), ToT (Tree-of-Thought)

Here is below a workflow explanation

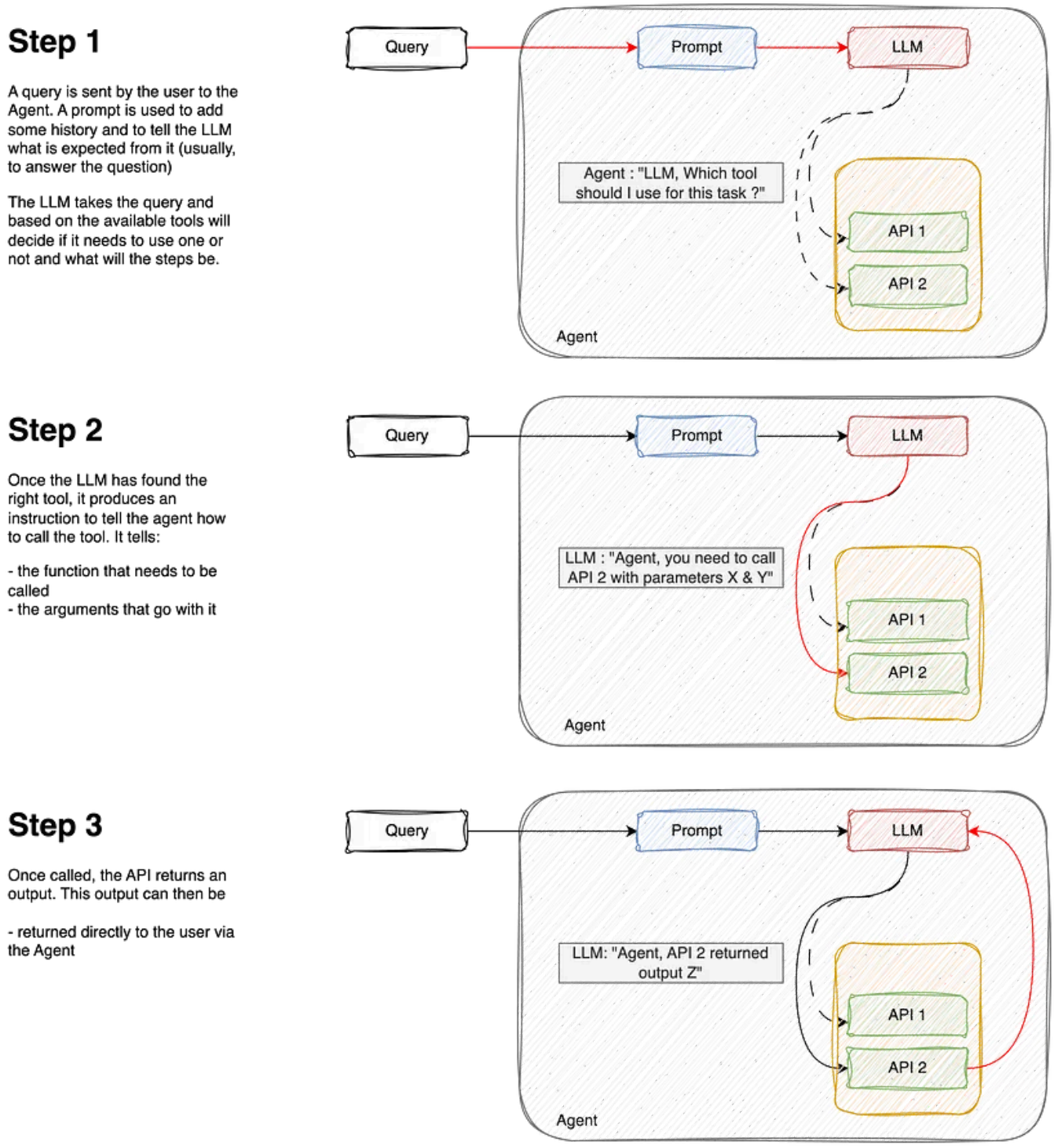


image by author

Chains are somehow different. Whereas agents can 'decide' by themselves what to do and which steps to take, chains are just a sequence of predefined steps. They can still rely on tools though, meaning that they can include a step in which they need to select from available tools. We'll cover that later.

3. Creating a simple chat without Tools

To illustrate our point, we will first of all see how our LLM performs as-is, without any help.

Let's install the needed libraries :

```
vertexai==1.65.0
langchain==0.2.16
langchain-community==0.2.16
langchain-core==0.2.38
langchain-google-community==1.0.8
langchain-google-vertexai==1.0.6
```

And create our very simple chat using Google's Gemini LLM:

```
from vertexai.generative_models import (
    GenerativeModel,
    GenerationConfig,
    Part
)

gemini_model = GenerativeModel(
    "gemini-1.5-flash",
    generation_config=GenerationConfig(temperature=0),
)
chat = gemini_model.start_chat()
```

If you run this simple chat and ask a question about the current exchange rate, you might probably get a similar answer:

```
response = chat.send_message("What is the current exchange rate for USD vs EUR ?
answer = response.candidates[0].content.parts[0].text

--- OUTPUT ---
"I am sorry, I cannot fulfill this request. I do not have access to real-time in
```

Not surprising, as we know LLMs do not have access to real-time data.

Let's add a tool for that. Our tool will be little function that calls an API to retrieve exchange rate data in real time.

```
def get_exchange_rate_from_api(params):
    url = f"https://api.frankfurter.app/latest?from={params['currency_from']}&to
    print(url)
    api_response = requests.get(url)
    return api_response.text

# Try it out !
get_exchange_rate_from_api({'currency_from': 'USD', 'currency_to': 'EUR'})
---
'{"amount":1.0,"base":"USD","date":"2024-11-20","rates":{"EUR":0.94679}}'
```

Now we know how our tools works, we would like to tell our chat LLM to use this function to answer our question. We will therefore create a mono-tool agent. To do that, we have several options which I will list here:

- Use Google's Gemini chat API with Function Calling
- Use LangChain's API with Agents and Tools

Both have their advantages and drawbacks. The purpose of this article is also to show you the possibilities and let you decide which one you prefer.

4. Adding Tools to our chat: The Google way with Function Calling

There are basically two ways of creating a tool out of a function.

The 1st one is a “dictionary” approach where you specify inputs and description of the function in the Tool. The imporant parameters are:

- Name of the function (be explicit)
- Description : be verbose here, as a solid and exhaustive description will help the LLM select the right tool
- Parameters : this is where you specify your arguments (type and description). Again, be verbose in the description of your arguments to help the LLM know how to pass value to your function

```
import requests

from vertexai.generative_models import FunctionDeclaration

get_exchange_rate_func = FunctionDeclaration(
    name="get_exchange_rate",
    description="Get the exchange rate for currencies between countries",
    parameters={
    "type": "object",
    "properties": {
        "currency_from": {
            "type": "string",
            "description": "The currency to convert from in ISO 4217 format"
        },
        "currency_to": {
            "type": "string",
            "description": "The currency to convert to in ISO 4217 format"
        }
    },
        "required": [
```

```
                "currency_from",
                "currency_to",
            ]
        },
    )
```

The 2nd way of adding a tool using Google's SDK is with a `from_func` instantiation. This requires editing our original function to be more explicit, with a docstring, etc. Instead of being verbose in the Tool creation, we are being verbose in the function creation.

```
# Edit our function
def get_exchange_rate_from_api(currency_from: str, currency_to: str):
    """
    Get the exchange rate for currencies

    Args:
        currency_from (str): The currency to convert from in ISO 4217 format
        currency_to (str): The currency to convert to in ISO 4217 format
    """
    url = f"https://api.frankfurter.app/latest?from={currency_from}&to={currency
    api_response = requests.get(url)
    return api_response.text

# Create the tool
get_exchange_rate_func = FunctionDeclaration.from_func(
    get_exchange_rate_from_api
)
```

The next step is really about creating the tool. For that, we will add our FunctionDeclaration to a list to create our Tool object:

```
from vertexai.generative_models import Tool as VertexTool

tool = VertexTool(
    function_declarations=[
        get_exchange_rate_func,
        # add more functions here !
    ]
)
```

Let's now pass that to our chat and see if it now can answer our query about exchange rates ! Remember, without tools, our chat answered:

```
response = chat.send_message("What is the current exchange rate for USD vs EUR ?")
response.candidates[0].content.parts[0].text
```

```
'I am sorry, I cannot fulfill this request. I do not have access to real-time information, including financial data like exchange rates.
\n'
```

Let's try Google's Function calling tool and see if this helps ! First, let's send our query to the chat:

```
from vertexai.generative_models import GenerativeModel

gemini_model = GenerativeModel(
    "gemini-1.5-flash",
    generation_config=GenerationConfig(temperature=0),
    tools=[tool] #We add the tool here !
)
chat = gemini_model.start_chat()

response = chat.send_message(prompt)

# Extract the function call response
response.candidates[0].content.parts[0].function_call

--- OUTPUT ---
"""
name: "get_exchange_rate"
args {
```

```
  fields {
    key: "currency_to"
    value {
      string_value: "EUR"
    }
  }
  fields {
    key: "currency_from"
    value {
      string_value: "USD"
    }
  }
  fields {
    key: "currency_date"
    value {
      string_value: "latest"
    }
  }
}"""
```

The LLM correctly guessed it needed to use the `get_exchange_rate` function, and also correctly guessed the 2 parameters were `USD` and `EUR` .

But this is not enough. What we want now is to actually run this function to get our results!

```
# mapping dictionnary to map function names and function
function_handler = {
    "get_exchange_rate": get_exchange_rate_from_api,
}

# Extract the function call name
function_name = function_call.name
print("#### Predicted function name")
print(function_name, "\n")

# Extract the function call parameters
params = {key: value for key, value in function_call.args.items()}
print("#### Predicted function parameters")
print(params, "\n")
```

```
function_api_response = function_handler[function_name](params)
print("#### API response")
print(function_api_response)
response = chat.send_message(
    Part.from_function_response(
        name=function_name,
        response={"content": function_api_response},
    ),
)
print("\n#### Final Answer")
print(response.candidates[0].content.parts[0].text)

--- OUTPUT ---
"""
#### Predicted function name
get_exchange_rate

#### Predicted function parameters
{'currency_from': 'USD', 'currency_date': 'latest', 'currency_to': 'EUR'}


#### API response
{"amount":1.0,"base":"USD","date":"2024-11-20","rates":{"EUR":0.94679}}

#### Final Answer
The current exchange rate for USD vs EUR is 0.94679. This means that 1 USD is eq
"""
```

We can now see our chat is able to answer our question! It:

- Correctly guessed to function to call, `get_exchange_rate`
- Correctly assigned the parameters to call the function `{'currency_from': 'USD', 'currency_to': 'EUR'}`
- Got results from the API
- And nicely formatted the answer to be human-readable!

Let's now see another way of doing with LangChain.

5. Adding Tools to our chat: The Langchain way with Agents

LangChain is a composable framework to build with LLMs. It is the orchestration framework for controllable agentic workflows.

Similar to what we did before the “Google” way, we will build tools in the Langchain way. Let’s begin with defining our functions. Same as for Google, we need to be exhaustive and verbose in the docstrings:

```
from langchain_core.tools import tool

@tool
def get_exchange_rate_from_api(currency_from: str, currency_to: str) -> str:
    """
    Return the exchange rate between currencies
    Args:
        currency_from: str
        currency_to: str
    """
    url = f"https://api.frankfurter.app/latest?from={currency_from}&to={currency
    api_response = requests.get(url)
    return api_response.text
```

In order to spice things up, I will add another tool which can list tables in a BigQuery dataset. Here is the code:

```
@tool
def list_tables(project: str, dataset_id: str) -> list:
    """
    Return a list of Bigquery tables
    Args:
        project: GCP project id
        dataset_id: ID of the dataset
    """
    client = bigquery.Client(project=project)
```

```
        try:
            response = client.list_tables(dataset_id)
            return [table.table_id for table in response]
        except Exception as e:
            return f"The dataset {params['dataset_id']} is not found in the {params[
```

Add once done, we add our functions to our LangChain toolbox !

```
langchain_tool = [
    list_tables,
    get_exchange_rate_from_api
]
```

To build our agent, we will use the `AgentExecutor` object from LangChain. This object will basically take 3 components, which are the ones we defined earlier :

- A LLM
- A prompt
- And tools.

Let's first choose our LLM:

```
gemini_llm = ChatVertexAI(model="gemini-1.5-flash")
```

Then we create a prompt to manage the conversation:

```
prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "You are a helpful assistant"),
        ("human", "{input}"),
        # Placeholders fill up a **list** of messages
        ("placeholder", "{agent_scratchpad}"),
    ]
)
```

And finally, we create the `AgentExecutor` and run a query:

```
agent = create_tool_calling_agent(gemini_llm, langchain_tools, prompt)
agent_executor = AgentExecutor(agent=agent, tools=langchain_tools)
agent_executor.invoke({
    "input": "Which tables are available in the thelook_ecommerce dataset ?"
})

--- OUTPUT ---
"""
{'input': 'Which tables are available in the thelook_ecommerce dataset ?',
 'output': 'The dataset `thelook_ecommerce` is not found in the `gcp-project-id`
            Please specify the correct dataset and project. \n'}
"""
```

Hmmm. Seems like the agent is missing one argument, or at least asking for more information…Let's reply by giving this information:

```
agent_executor.invoke({"input": f"Project id is bigquery-public-data"})

--- OUPTUT ---
"""
{'input': 'Project id is bigquery-public-data',
```

```
 'output': 'OK. What else can I do for you? \n'}
"""
```

Well, seems we're back to square one. The LLM has been told the project id but forgot about the question. Our agent seems to be lacking memory to remember previous questions and answers. Maybe we should think of...

6. Adding Memory to our Agent

Memory is another concept in Agents, which basically helps the system to remember the conversation history and avoid endless loops like above. Think of memory as being a notepad where the LLM keeps track of previous questions and answers to build context around the conversation.

We will modify our prompt (instructions) to the model to include memory:

```
from langchain_core.chat_history import InMemoryChatMessageHistory
from langchain_core.runnables.history import RunnableWithMessageHistory

# Different types of memory can be found in Langchain
memory = InMemoryChatMessageHistory(session_id="foo")

prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "You are a helpful assistant."),
        # First put the history
        ("placeholder", "{chat_history}"),
        # Then the new input
        ("human", "{input}"),
        # Finally the scratchpad
        ("placeholder", "{agent_scratchpad}"),
    ]
)

# Remains unchanged
agent = create_tool_calling_agent(gemini_llm, langchain_tools, prompt)
agent_executor = AgentExecutor(agent=agent, tools=langchain_tools)
```

```
# We add the memory part and the chat history
agent_with_chat_history = RunnableWithMessageHistory(
    agent_executor,
    lambda session_id: memory, #<-- NEW
    input_messages_key="input",
    history_messages_key="chat_history", #<-- NEW
)

config = {"configurable": {"session_id": "foo"}}
```

We will now rerun our query from the beginning:

```
agent_with_chat_history.invoke({
    "input": "Which tables are available in the thelook_ecommerce dataset ?"
    },
    config
)

--- OUTPUT ---
"""
{'input': 'Which tables are available in the thelook_ecommerce dataset ?',
 'chat_history': [],
 'output': 'The dataset `thelook_ecommerce` is not found in the `gcp-project-id`
"""
```

With an empty chat history, the model still asks for the project id. Pretty consistent with what we had before with a memoryless agent. Let's reply to the agent and add the missing information:

```
reply = "Project id is bigquery-public-data"
agent_with_chat_history.invoke({"input": reply}, config)

--- OUTPUT ---
"""
{'input': 'Project id is bigquery-public-data',
```

```
    'chat_history': [HumanMessage(content='Which tables are available in the theloo
     AIMessage(content='The dataset `thelook_ecommerce` is not found in the `gcp-pr
    'output': 'The following tables are available in the `thelook_ecommerce` datase
   """
```

Notice how, in the output:

- The `chat history` keeps track of the previous Q&A
- The output now returns the list of the tables!

```
   'output': 'The following tables are available in the `thelook_ecommerce` dataset
```

In some use cases however, certain actions might require special attention because of their nature (ie deleting an entry in a database, editing information, sending an email, etc.). Full automation without control might leads to situations where the agent takes wrong decisions and creates damage.

One way to secure our workflows is to add a human-in-the-loop step.

7. Creating a Chain with a Human Validation step

A chain is somehow different from an agent. Whereas the agent can decide to use or not to use tools, a chain is more static. It is a sequence of steps, for which we can still include a step where the LLM will choose from a set of tools.

To build chains in LangChain, we use LCEL.
LangChain Expression Language, or LCEL, is a declarative way to easily compose chains together. Chains in LangChain use the pipe `|` operator to indicate the orders in which steps have to be executed, such as `step 1 | step 2 | step 3 etc`. The difference with Agents is that Chains will always follow those steps, whereas Agents can "decide" by themselves and are autonomous in their decision-making process.

In our case, we will proceed as follows to build a simple `prompt | llm` chain.

```
# define the prompt with memory
prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "You are a helpful assistant."),
        # First put the history
        ("placeholder", "{chat_history}"),
        # Then the new input
        ("human", "{input}"),
        # Finally the scratchpad
        ("placeholder", "{agent_scratchpad}"),
    ]
)

# bind the tools to the LLM
gemini_with_tools = gemini_llm.bind_tools(langchain_tool)

# build the chain
chain = prompt | gemini_with_tools
```

Remember how in the previous step we passed an agent to our `RunnableWithMessageHistory`? Well, we will do the same here, but...

```
# With AgentExecutor
```

```
# agent = create_tool_calling_agent(gemini_llm, langchain_tool, prompt)
# agent_executor = AgentExecutor(agent=agent, tools=langchain_tool)

# agent_with_chat_history = RunnableWithMessageHistory(
#     agent_executor,
#     lambda session_id: memory,
#     input_messages_key="input",
#     history_messages_key="chat_history",
# )

config = {"configurable": {"session_id": "foo"}}

# With Chains
memory = InMemoryChatMessageHistory(session_id="foo")
chain_with_history = RunnableWithMessageHistory(
    chain,
    lambda session_id: memory,
    input_messages_key="input",
    history_messages_key="chat_history",
)

response = chain_with_history.invoke(
    {"input": "What is the current CHF EUR exchange rate ?"}, config)

--- OUTPUT
"""
content='',
additional_kwargs={
    'function_call': {
        'name': 'get_exchange_rate_from_api',
        'arguments': '{"currency_from": "CHF", "currency_to": "EUR"}'
    }
}
"""
```

Unlike the agent, a chain does not provide the answer unless we tell it to. In our case, it stopped at the step where the LLM returns the function that needs to be called.

We need to add an extra step to actually *call* the tool. Let's add another function to call the tools:

```
from langchain_core.messages import AIMessage

def call_tools(msg: AIMessage) -> list[dict]:
    """Simple sequential tool calling helper."""
    tool_map = {tool.name: tool for tool in langchain_tool}
    tool_calls = msg.tool_calls.copy()
    for tool_call in tool_calls:
        tool_call["output"] = tool_map[tool_call["name"]].invoke(tool_call["args
    return tool_calls

chain = prompt | gemini_with_tools | call_tools #<-- Extra step

chain_with_history = RunnableWithMessageHistory(
    chain,
    lambda session_id: memory,
    input_messages_key="input",
    history_messages_key="chat_history",
)

# Rerun the chain
chain_with_history.invoke({"input": "What is the current CHF EUR exchange rate ?
```

We now get the following output, which shows the API has been successfully called:

```
[{'name': 'get_exchange_rate_from_api',
  'args': {'currency_from': 'CHF', 'currency_to': 'EUR'},
  'id': '81bc85ea-dfd4-4c01-85e8-f3ca592fff5b',
  'type': 'tool_call',
  'output': '{"amount":1.0,"base":"USD","date":"2024-11-20","rates":{"EUR":0.946
}]
```

Now we understood how to chain steps, let's add our human-in-the-loop step ! We want this step to check that the LLM has understood our requests and

will make the right call to an API. If the LLM has misunderstood the request or will use the function incorrectly, we can decide to interrupt the process.

```
def human_approval(msg: AIMessage) -> AIMessage:
    """Responsible for passing through its input or raising an exception.

    Args:
        msg: output from the chat model

    Returns:
        msg: original output from the msg
    """
    for tool_call in msg.tool_calls:
        print(f"I want to use function [{tool_call.get('name')}] with the follow
        for k,v in tool_call.get('args').items():
            print(" {} = {}".format(k, v))

    print("")
    input_msg = (
        f"Do you approve (Y|y)?\n\n"
        ">>>"
    )
    resp = input(input_msg)
    if resp.lower() not in ("yes", "y"):
        raise NotApproved(f"Tool invocations not approved:\n\n{tool_strs}")
    return msg
```

Next, add this step to the chain before the function call:

```
chain = prompt | gemini_with_tools | human_approval | call_tools

memory = InMemoryChatMessageHistory(session_id="foo")

chain_with_history = RunnableWithMessageHistory(
    chain,
    lambda session_id: memory,
    input_messages_key="input",
    history_messages_key="chat_history",
```

```
    )

    chain_with_history.invoke({"input": "What is the current CHF EUR exchange rate ?
```

You will then be asked to confirm that the LLM understood correctly:

```
 I want to use function [get_exchange_rate_from_api] with the following parameters :
  currency_from = CHF
  currency_to = EUR

Do you approve (Y|y)?

>>> ↑↓ for history. Search history with c-↑/c-↓
```

This human-in-the-loop step can be very helpful for critical workflows where a misinterpretation from the LLM could have dramatic consequences.

8. Using search tools

One of the most convenient tools to retrieve information in real-time are search engines . One way to do that is to use `GoogleSerperAPIWrapper` (you will need to register to get an API key in order to use it), which provides a nice interface to query Google Search and get results quickly.

Luckily, LangChain already provides a tool for you, so we won't have to write the function ourselves.

Let's therefore try to ask a question on yesterday's event (Nov 20th) and see if our agent can answer. Our question is about Rafael Nadal's last official game (which he lost to van de Zandschulp).

```
agent_with_chat_history.invoke(
    {"input": "What was the result of Rafael Nadal's latest game ?"}, config)

--- OUTPUT ---
"""
{'input': "What was the result of Rafael Nadal's latest game ?",
 'chat_history': [],
 'output': "I do not have access to real-time information, including sports resu
"""
```

Without being able to access Google Search, our model is unable to answer because this information was not available at the time it was trained.

Let's now add our Serper tool to our toolbox and see if our model can use Google Search to find the information:

```
from langchain_community.utilities import GoogleSerperAPIWrapper

# Create our new search tool here
search = GoogleSerperAPIWrapper(serper_api_key="...")

@tool
def google_search(query: str):
    """
    Perform a search on Google
    Args:
        query: the information to be retrieved with google search
    """
    return search.run(query)

# Add it to our existing tools
langchain_tool = [
    list_datasets,
    list_tables,
    get_exchange_rate_from_api,
    google_search
]
```

```
# Create agent
agent = create_tool_calling_agent(gemini_llm, langchain_tool, prompt)
agent_executor = AgentExecutor(agent=agent, tools=langchain_tool)

# Add memory
memory = InMemoryChatMessageHistory()
agent_with_chat_history = RunnableWithMessageHistory(
    agent_executor,
    lambda session_id: memory,
    input_messages_key="input",
    history_messages_key="chat_history",
)
```

And rerun our query :

```
agent_with_chat_history.invoke({"input": "What was the result of Rafael Nadal's

--- OUTPUT ---
"""
{'input': "What was the result of Rafael Nadal's latest game ?",
 'chat_history': [],
 'output': "Rafael Nadal's last match was a loss to Botic van de Zandschulp in t
"""
```

Conclusion

LLMs alone often hit a blocker when it comes to using personal, corporate, private or real-data. Indeed, such information is generally not available at training time. Agents and tools are a powerful way to augment these models by allowing them to interact with systems and APIs, and orchestrate workflows to boost productivity.

Agents Vertex Llm Genai Hands On Tutorials

Published in Towards Data Science

769K Followers · Last published just now

Follow

Your home for data science and AI. The world's leading publication for data science, data analytics, data engineering, machine learning, and artificial intelligence professionals.

Written by Benjamin Etienne

1.1K Followers · 9 Following

Follow

Head of Data Science at Cartier

More from Benjamin Etienne and Towards Data Science

In Towards Data Science by Benjamin Etienne

Time Series in Python — Exponential Smoothing and ARIM...

TL;DR: In this article you'll learn the basics steps to performing time-series analysis and...

In Towards Data Science by Claudia Ng

The Most Expensive Data Science Mistake I've Witnessed in My...

Why true success in machine learning goes beyond optimizing a single metric

Feb 9, 2019 1.8K 18 1d ago 632 18

In Towards Data Science by Dima Sergeev

Dog Poop Compass: Bayesian Analysis of Canine Business

A Bayesian analysis of canine business

4d ago 119

In Towards Data Science by Benjamin Etienne

A Complete Guide to Write your own Transformers

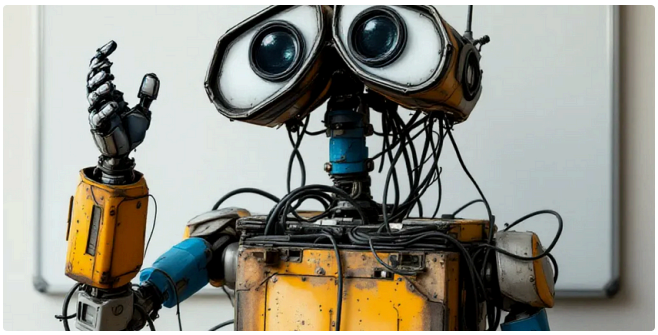
An end-to-end implementation of a Pytorch Transformer, in which we will cover key...

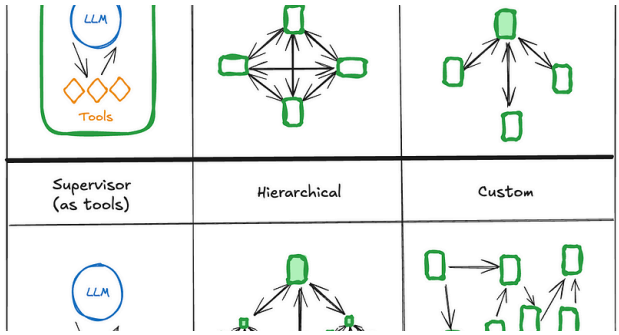
Feb 24 1.2K 9

See all from Benjamin Etienne

See all from Towards Data Science

Recommended from Medium





tds In Towards Data Science by Heiko Hotz

Productionising GenAI Agents: Evaluating Tool Selection with...

How to create reliable and scalable GenAI Agents for real-world applications

6d ago 260 4

A Alfredo Sone

AI Agents: How to build Digital Workers

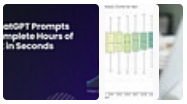
Key learnings to understand and design intelligent digital workers.

Nov 18 368 9

Lists

Natural Language Processing
1839 stories · 1461 saves

Stories to Help You Grow as a Software Developer
19 stories · 1495 saves



ChatGPT prompts
50 stories · 2297 saves

Generative AI Recommended Reading
52 stories · 1526 saves

In Towards AI by Gao Dalie (高達烈)

Browser-use + LightRAG Agent That Can Scrape 99% websites wi...

In this story, I have a quick tutorial showing how to create a powerful chatbot using...

Nov 20 785 4

In DataDrivenInvestor by Bex T.

Firecrawl: How to Scrape Entire Websites With a Single Command...

And turn them into LLM-ready data

5d ago 362 3

In Stackademic by Crafting-Code

I Stopped Using Kubernetes. Our DevOps Team Is Happier Than Ever

Why Letting Go of Kubernetes Worked for Us

Nov 19 2.9K 94

In Level Up Coding by Yeyu Huang

How to Create an Interactive Web UI for CrewAI Applications (New...

A Quick Update for the Implementation of CrewAI + Panel UI

Nov 21 315 2

See more recommendations