May 6, 2025/[#Python](https://www.freecodecamp.org/news/tag/python/)

<https://www.freecodecamp.org/news/build-a-local-ai/>

**How to Build Your Own Local AI: Create Free RAG and AI Agents with Qwen 3 and Ollama**

**[Chaitanya Rahalkar](https://www.freecodecamp.org/news/author/chaitanyarahalkar/)**

The landscape of Artificial Intelligence is rapidly evolving, and one of the most exciting trends is the ability to run powerful Large Language Models (LLMs) directly on your local machine.

This shift away from reliance on cloud-based APIs offers significant advantages in terms of privacy, cost-effectiveness, and offline accessibility. Developers and enthusiasts can now experiment with and deploy sophisticated AI capabilities without sending data externally or incurring API fees.

This tutorial serves as a practical, hands-on guide to harnessing this local AI power. It focuses on leveraging the Qwen 3 family of LLMs, a state-of-the-art open-source offering from Alibaba, combined with Ollama, a tool that dramatically simplifies running LLMs locally.

**Prerequisites**

Before diving into this tutorial, you should have a foundational understanding of Python programming and be comfortable using the command line or terminal. Make sure you have Python 3 installed on your system.

While prior experience with AI or Large Language Models (LLMs) is beneficial, it's not essential, as I’ll introduce and explain core concepts like Retrieval-Augmented Generation (RAG) and AI agents throughout the guide.

This tutorial serves as a practical, hands-on guide to harnessing this local AI power. It focuses on leveraging the Qwen 3 family of LLMs, a state-of-the-art open-source offering from Alibaba, combined with Ollama, a tool that dramatically simplifies running LLMs locally.

**Table of Contents**

1. [Local AI Power with Qwen 3 and Ollama](https://www.freecodecamp.org/news/build-a-local-ai/" \l "heading-local-ai-power-with-qwen-3-and-ollama)
   * [Ollama: Your Local LLM Gateway](https://www.freecodecamp.org/news/build-a-local-ai/#heading-ollama-your-local-llm-gateway)
   * [Tutorial Roadmap](https://www.freecodecamp.org/news/build-a-local-ai/#heading-tutorial-roadmap)
2. [How to Set Up Your Local AI Lab](https://www.freecodecamp.org/news/build-a-local-ai/" \l "heading-how-to-set-up-your-local-ai-lab)
   * [Install Ollama](https://www.freecodecamp.org/news/build-a-local-ai/#heading-install-ollama)
   * [Choose Your Qwen 3 Model](https://www.freecodecamp.org/news/build-a-local-ai/#heading-choose-your-qwen-3-model)
   * [Pull and Run Qwen 3 with Ollama](https://www.freecodecamp.org/news/build-a-local-ai/" \l "heading-pull-and-run-qwen-3-with-ollama)
   * [Set Up Your Python Environment](https://www.freecodecamp.org/news/build-a-local-ai/#heading-set-up-your-python-environment)
3. [How to Build a Local RAG System with Qwen 3](https://www.freecodecamp.org/news/build-a-local-ai/" \l "heading-how-to-build-a-local-rag-system-with-qwen-3)
   * [Step 1: Prepare Your Data](https://www.freecodecamp.org/news/build-a-local-ai/#heading-step-1-prepare-your-data)
   * [Step 2: Load Documents in Python](https://www.freecodecamp.org/news/build-a-local-ai/#heading-step-2-load-documents-in-python)
   * [Step 3: Split Documents](https://www.freecodecamp.org/news/build-a-local-ai/#heading-step-3-split-documents)
   * [Step 4: Choose and Configure Embedding Model](https://www.freecodecamp.org/news/build-a-local-ai/" \l "heading-step-4-choose-and-configure-embedding-model)
   * [Step 5: Set Up Local Vector Store (ChromaDB)](https://www.freecodecamp.org/news/build-a-local-ai/#heading-step-5-set-up-local-vector-store-chromadb)
   * [Step 6: Index Documents (Embed and Store)](https://www.freecodecamp.org/news/build-a-local-ai/#heading-step-6-index-documents-embed-and-store)
   * [Step 7: Build the RAG Chain](https://www.freecodecamp.org/news/build-a-local-ai/#heading-step-7-build-the-rag-chain)
   * [Step 8: Query Your Documents](https://www.freecodecamp.org/news/build-a-local-ai/#heading-step-8-query-your-documents)
4. [How to Create Local AI Agents with Qwen 3](https://www.freecodecamp.org/news/build-a-local-ai/#heading-how-to-create-local-ai-agents-with-qwen-3)
   * [Step 1: Define Custom Tools](https://www.freecodecamp.org/news/build-a-local-ai/#heading-step-1-define-custom-tools)
   * [Step 2: Set up the Agent LLM](https://www.freecodecamp.org/news/build-a-local-ai/#heading-step-2-set-up-the-agent-llm)
   * [Step 3: Create the Agent Prompt](https://www.freecodecamp.org/news/build-a-local-ai/#heading-step-3-create-the-agent-prompt)
   * [Step 4: Build the Agent](https://www.freecodecamp.org/news/build-a-local-ai/#heading-step-4-build-the-agent)
   * [Step 5: Create the Agent Executor](https://www.freecodecamp.org/news/build-a-local-ai/#heading-step-5-create-the-agent-executor)
   * [Step 6: Run the Agent](https://www.freecodecamp.org/news/build-a-local-ai/#heading-step-6-run-the-agent)
5. [Advanced Considerations and Troubleshooting](https://www.freecodecamp.org/news/build-a-local-ai/#heading-advanced-considerations-and-troubleshooting)
   * [Controlling Qwen 3's Thinking Mode with Ollama](https://www.freecodecamp.org/news/build-a-local-ai/#heading-controlling-qwen-3s-thinking-mode-with-ollama)
   * [Managing Context Length (num\_ctx)](https://www.freecodecamp.org/news/build-a-local-ai/#heading-managing-context-length-numctx)
   * [Hardware Limitations and VRAM](https://www.freecodecamp.org/news/build-a-local-ai/#heading-hardware-limitations-and-vram)
6. [Conclusion and Next Steps](https://www.freecodecamp.org/news/build-a-local-ai/#heading-conclusion-and-next-steps)

**Local AI Power with Qwen 3 and Ollama**

Running LLMs locally addresses several key concerns associated with cloud-based AI services.

* Privacy is paramount – data processed locally never leaves the user's machine.
* Cost is another major factor – utilizing open-source models and tools like Ollama eliminates API subscription fees and pay-per-token charges, making advanced AI accessible to everyone.
* Local execution enables offline functionality – crucial for applications where internet connectivity is unreliable or undesirable.

**Ollama: Your Local LLM Gateway**

Ollama acts as a bridge, making the power of models like Qwen 3 accessible on local hardware. It's a command-line tool that simplifies the download, setup, and execution of various open-source LLMs across macOS, Linux, and Windows.

Ollama handles the complexities of model configuration and GPU utilization, providing a straightforward interface for developers and users. It also exposes an OpenAI-compatible API endpoint, allowing seamless integration with popular frameworks like LangChain.

**Tutorial Roadmap**

This tutorial will guide you through the process of:

1. **Setting up a local AI environment:** Installing Ollama and selecting/running appropriate Qwen 3 models.
2. **Building a local RAG system:** Creating a system that allows chatting with personal documents using Qwen 3, Ollama, LangChain, and ChromaDB for vector storage.
3. **Creating a basic local AI agent:** Developing a simple agent powered by Qwen 3 that can utilize custom-defined tools (functions).

**How to Set Up Your Local AI Lab**

The first step is to prepare your local machine with the necessary tools and models.

**Install Ollama**

Ollama provides the simplest path to running LLMs locally.

* **Linux / macOS:** Open a terminal and run the official installation script:
* curl -fsSL https://ollama.com/install.sh | sh
* **Windows:** Download the installer from the Ollama website (<https://ollama.com/download>) and follow the setup instructions.

After installation, verify it by opening a new terminal window and running:

ollama --version

Ollama typically stores downloaded models in ~/.ollama/models on macOS and /usr/share/ollama/.ollama/models on Linux/WSL.

**Choose Your Qwen 3 Model**

Selecting the right Qwen 3 model is crucial and depends on your intended task and available hardware, primarily system RAM and GPU VRAM. Running larger models requires more resources but generally offers better performance and reasoning capabilities.

Qwen 3 offers two main architectures available through Ollama:

* **Dense Models:** (like qwen3:0.6b, qwen3:4b, qwen3:8b, qwen3:14b, qwen3:32b) These models activate all their parameters during inference. Their performance is predictable, but resource requirements scale directly with parameter count.
* **Mixture-of-Experts (MoE) Models:** (like qwen3:30b-a3b) These models contain many "expert" sub-networks but only activate a small fraction for each input token. This allows them to achieve the performance characteristic of their large total parameter count (for example, 30 billion) while having inference costs closer to their smaller *active* parameter count (for example, 3 billion). They offer a compelling balance of capability and efficiency, especially for reasoning and coding tasks.

**Recommendation for this tutorial:** For the examples that follow, qwen3:8b strikes a good balance between capability and resource requirements for many modern machines. If resources are more constrained, qwen3:4b is a viable alternative. The MoE model qwen3:30b-a3b offers excellent performance, especially for coding and reasoning, and runs surprisingly well on systems with 16GB+ VRAM due to its sparse activation.

**Pull and Run Qwen 3 with Ollama**

Once you’ve chosen a model, you’ll need to download it (pull it) via Ollama.

**Pull the model:** Open the terminal and run (replace qwen3:8b with the desired tag):

ollama pull qwen3:8b

This command downloads the model weights and configuration.

**Run interactively (optional test):** To chat directly with the model from the command line:

ollama run qwen3:8b

Type prompts directly into the terminal. Use /bye to exit the session. Other useful commands within the interactive session include /? for help and /set parameter <name> <value> (for example, /set parameter num\_ctx 8192) to temporarily change model parameters for the current session. Use ollama list outside the session to see downloaded models.

**Run as a server:** For integration with Python scripts (using LangChain), Ollama needs to run as a background server process, exposing an API. Open a *separate* terminal window and run:

ollama serve

Keep this terminal window open while running the Python scripts. This command starts the server, typically listening on http://localhost:11434, providing an OpenAI-compatible API endpoint.

**Set Up Your Python Environment**

A dedicated Python environment is recommended for managing dependencies.

**Create a virtual environment:**

python -m venv venv

**Activate the environment:**

* macOS/Linux: source venv/bin/activate
* Windows: venv\Scripts\activate

**Install necessary libraries:**

pip install langchain langchain-community langchain-core langchain-ollama chromadb sentence-transformers pypdf python-dotenv unstructured[pdf] tiktoken

* langchain, langchain-community, langchain-core: The core LangChain framework for building LLM applications.
* langchain-ollama: Specific integration for using Ollama models with LangChain.
* chromadb: The local vector database for storing document embeddings.
* sentence-transformers: Used for an alternative local embedding method (explained later).
* pypdf: A library for loading PDF documents.
* python-dotenv: For managing environment variables (optional but good practice).
* unstructured[pdf]: An alternative, powerful document loader, especially for complex PDFs.
* tiktoken: Used by LangChain for token counting.

The local setup involves coordinating several independent components: Ollama itself, the specific Qwen 3 model weights, the Python environment, and various libraries like LangChain and ChromaDB. Ensuring compatibility between these pieces and correctly configuring parameters (like Ollama's context window size or selecting a model appropriate for the available VRAM) is key to a smooth experience.

While this modularity offers flexibility – allowing components like the LLM or vector store to be swapped – it also means the initial setup requires careful attention to detail. This tutorial aims to provide clear steps and sensible defaults to minimize potential friction points.

**How to Build a Local RAG System with Qwen 3**

Retrieval-Augmented Generation (RAG) is a powerful technique that enhances LLMs by providing them with external knowledge.

Instead of relying solely on its training data, the LLM can retrieve relevant information from a specified document set (like local PDFs) and uses that information to answer questions. This significantly reduces "hallucinations" (incorrect or fabricated information) and allows the LLM to answer questions about specific, private data without needing retraining.

The core RAG process involves:

1. Loading and splitting documents into manageable chunks.
2. Converting these chunks into numerical representations (embeddings) using an embedding model.
3. Storing these embeddings in a vector database for efficient searching.
4. When a query comes in, embedding the query and searching the vector database for the most similar document chunks.
5. Providing these relevant chunks (context) along with the original query to the LLM to generate an informed answer.

Let's build this locally using Qwen 3, Ollama, LangChain, and ChromaDB.

**Step 1: Prepare Your Data**

Create a directory named data in the project folder. Place the PDF document that you intend to query into this directory. For this tutorial, using a single, primarily text-based PDF (like a research paper or a report) for simplicity.

mkdir data

# Copy your PDF file into the 'data' directory

# e.g., cp ~/Downloads/some\_paper.pdf./data/mydocument.pdf

If you don’t have a PDF readily available that you’d like to use, you can download a sample PDF (the Llama 2 paper) for this tutorial using the following command in your terminal:

wget --user-agent "Mozilla" "https://arxiv.org/pdf/2307.09288.pdf" -O "data/llama2.pdf"

This command creates the data directory and downloads the PDF, saving it as llama2.pdf inside the data directory. If you prefer to use your own document, place your PDF file into the data directory and update the filename in the subsequent Python code.

**Step 2: Load Documents in Python**

Use LangChain's document loaders to read the PDF content. PyPDFLoader is straightforward for simple PDFs. UnstructuredPDFLoader (requires unstructured[pdf]) can handle more complex layouts but has more dependencies.

# rag\_local.py

import os

from dotenv import load\_dotenv

from langchain\_community.document\_loaders import PyPDFLoader # Or UnstructuredPDFLoader

load\_dotenv() # Optional: Loads environment variables from.env file

DATA\_PATH = "data/"

PDF\_FILENAME = "mydocument.pdf" # Replace with your PDF filename

def load\_documents():

"""Loads documents from the specified data path."""

pdf\_path = os.path.join(DATA\_PATH, PDF\_FILENAME)

loader = PyPDFLoader(pdf\_path)

# loader = UnstructuredPDFLoader(pdf\_path) # Alternative

documents = loader.load()

print(f"Loaded {len(documents)} page(s) from {pdf\_path}")

return documents

# documents = load\_documents() # Call this later

**Step 3: Split Documents**

Large documents need to be split into smaller chunks suitable for embedding and retrieval. The RecursiveCharacterTextSplitter attempts to split text semantically (at paragraphs, sentences, and so on) before resorting to fixed-size splits. chunk\_size determines the maximum size of each chunk (in characters), and chunk\_overlap specifies how many characters should overlap between consecutive chunks to maintain context.

# rag\_local.py (continued)

from langchain\_text\_splitters import RecursiveCharacterTextSplitter

def split\_documents(documents):

"""Splits documents into smaller chunks."""

text\_splitter = RecursiveCharacterTextSplitter(

chunk\_size=1000,

chunk\_overlap=200,

length\_function=len,

is\_separator\_regex=False,

)

all\_splits = text\_splitter.split\_documents(documents)

print(f"Split into {len(all\_splits)} chunks")

return all\_splits

# loaded\_docs = load\_documents()

# chunks = split\_documents(loaded\_docs) # Call this later

**Step 4: Choose and Configure Embedding Model**

Embeddings transform text into vectors (lists of numbers) such that semantically similar text chunks have vectors that are close together in multi-dimensional space.

**Option A (Recommended for Simplicity): Ollama Embeddings**

This approach uses Ollama to serve a dedicated embedding model. nomic-embed-text is a capable open-source model available via Ollama.

First, ensure the embedding model is pulled:

ollama pull nomic-embed-text

Then, use OllamaEmbeddings in Python:

# rag\_local.py (continued)

from langchain\_ollama import OllamaEmbeddings

def get\_embedding\_function(model\_name="nomic-embed-text"):

"""Initializes the Ollama embedding function."""

# Ensure Ollama server is running (ollama serve)

embeddings = OllamaEmbeddings(model=model\_name)

print(f"Initialized Ollama embeddings with model: {model\_name}")

return embeddings

# embedding\_function = get\_embedding\_function() # Call this later

**Option B (Alternative): Sentence Transformers**

This uses the sentence-transformers library directly within the Python script. It requires installing the library (pip install sentence-transformers) but doesn't need a separate Ollama process for embeddings. Models like all-MiniLM-L6-v2 are fast and lightweight, while all-mpnet-base-v2 offers higher quality.

# Alternative embedding function using Sentence Transformers

from langchain\_community.embeddings import HuggingFaceEmbeddings

def get\_embedding\_function\_hf(model\_name="all-MiniLM-L6-v2"):

"""Initializes HuggingFace embeddings (runs locally)."""

embeddings = HuggingFaceEmbeddings(model\_name=model\_name)

print(f"Initialized HuggingFace embeddings with model: {model\_name}")

return embeddings

embedding\_function = get\_embedding\_function\_hf() # Use this if choosing Option B

For this tutorial, we’ll use Option A (Ollama Embeddings with nomic-embed-text) to keep the toolchain consistent.

**Step 5: Set Up Local Vector Store (ChromaDB)**

ChromaDB provides an efficient way to store and search vector embeddings locally. Using a persistent client ensures the indexed data is saved to disk and can be reloaded without re-processing the documents every time.

# rag\_local.py (continued)

from langchain\_community.vectorstores import Chroma

CHROMA\_PATH = "chroma\_db" # Directory to store ChromaDB data

def get\_vector\_store(embedding\_function, persist\_directory=CHROMA\_PATH):

"""Initializes or loads the Chroma vector store."""

vectorstore = Chroma(

persist\_directory=persist\_directory,

embedding\_function=embedding\_function

)

print(f"Vector store initialized/loaded from: {persist\_directory}")

return vectorstore

embedding\_function = get\_embedding\_function()

vector\_store = get\_vector\_store(embedding\_function) # Call this later

**Step 6: Index Documents (Embed and Store)**

This is the core indexing step where document chunks are converted to embeddings and saved in ChromaDB. The Chroma.from\_documents function is convenient for the initial creation and indexing. If the database already exists, subsequent additions can use vectorstore.add\_documents.

# rag\_local.py (continued)

def index\_documents(chunks, embedding\_function, persist\_directory=CHROMA\_PATH):

"""Indexes document chunks into the Chroma vector store."""

print(f"Indexing {len(chunks)} chunks...")

# Use from\_documents for initial creation.

# This will overwrite existing data if the directory exists but isn't a valid Chroma DB.

# For incremental updates, initialize Chroma first and use vectorstore.add\_documents().

vectorstore = Chroma.from\_documents(

documents=chunks,

embedding=embedding\_function,

persist\_directory=persist\_directory

)

vectorstore.persist() # Ensure data is saved

print(f"Indexing complete. Data saved to: {persist\_directory}")

return vectorstore

#... (previous function calls)

vector\_store = index\_documents(chunks, embedding\_function) # Call this for initial indexing

To load an existing persistent database later:

embedding\_function = get\_embedding\_function()

vector\_store = Chroma(persist\_directory=CHROMA\_PATH, embedding\_function=embedding\_function)

**Step 7: Build the RAG Chain**

Now, assemble the components into a LangChain Expression Language (LCEL) chain. This involves initializing the Qwen 3 LLM via Ollama, creating a retriever from the vector store, defining a suitable prompt, and chaining them together.

A critical parameter when initializing ChatOllama for RAG is num\_ctx. This defines the context window size (in tokens) that the LLM can handle. Ollama's default (often 2048 or 4096 tokens) might be too small to accommodate both the retrieved document context and the user's query/prompt.

Qwen 3 models (8B and larger) support much larger context windows (for example, 128k tokens), but practical limits depend on your available RAM/VRAM. Setting num\_ctx to a value like 8192 or higher is often necessary for effective RAG.

# rag\_local.py (continued)

from langchain\_ollama import ChatOllama

from langchain\_core.prompts import ChatPromptTemplate

from langchain\_core.runnables import RunnablePassthrough

from langchain\_core.output\_parsers import StrOutputParser

def create\_rag\_chain(vector\_store, llm\_model\_name="qwen3:8b", context\_window=8192):

"""Creates the RAG chain."""

# Initialize the LLM

llm = ChatOllama(

model=llm\_model\_name,

temperature=0, # Lower temperature for more factual RAG answers

num\_ctx=context\_window # IMPORTANT: Set context window size

)

print(f"Initialized ChatOllama with model: {llm\_model\_name}, context window: {context\_window}")

# Create the retriever

retriever = vector\_store.as\_retriever(

search\_type="similarity", # Or "mmr"

search\_kwargs={'k': 3} # Retrieve top 3 relevant chunks

)

print("Retriever initialized.")

# Define the prompt template

template = """Answer the question based ONLY on the following context:

{context}

Question: {question}

"""

prompt = ChatPromptTemplate.from\_template(template)

print("Prompt template created.")

# Define the RAG chain using LCEL

rag\_chain = (

{"context": retriever, "question": RunnablePassthrough()}

| prompt

| llm

| StrOutputParser()

)

print("RAG chain created.")

return rag\_chain

#... (previous function calls)

vector\_store = get\_vector\_store(embedding\_function) # Assuming DB is already indexed

rag\_chain = create\_rag\_chain(vector\_store) # Call this later

The effectiveness of the RAG system hinges on the proper configuration of each component. The chunk\_size and chunk\_overlap in the splitter affect what the retriever finds. Your choice of embedding\_function must be consistent between indexing and querying. The num\_ctx parameter for the ChatOllama LLM must be large enough to hold the retrieved context and the prompt itself. A poorly designed prompt template can also lead the LLM astray. Make sure you carefully tune these elements for optimal performance.

**Step 8: Query Your Documents**

Finally, invoke the RAG chain with a question related to the content of the indexed PDF.

# rag\_local.py (continued)

def query\_rag(chain, question):

"""Queries the RAG chain and prints the response."""

print("\nQuerying RAG chain...")

print(f"Question: {question}")

response = chain.invoke(question)

print("\nResponse:")

print(response)

# --- Main Execution ---

if \_\_name\_\_ == "\_\_main\_\_":

# 1. Load Documents

docs = load\_documents()

# 2. Split Documents

chunks = split\_documents(docs)

# 3. Get Embedding Function

embedding\_function = get\_embedding\_function() # Using Ollama nomic-embed-text

# 4. Index Documents (Only needs to be done once per document set)

# Check if DB exists, if not, index. For simplicity, we might re-index here.

# A more robust approach would check if indexing is needed.

print("Attempting to index documents...")

vector\_store = index\_documents(chunks, embedding\_function)

# To load existing DB instead:

# vector\_store = get\_vector\_store(embedding\_function)

# 5. Create RAG Chain

rag\_chain = create\_rag\_chain(vector\_store, llm\_model\_name="qwen3:8b") # Use the chosen Qwen 3 model

# 6. Query

query\_question = "What is the main topic of the document?" # Replace with a specific question

query\_rag(rag\_chain, query\_question)

query\_question\_2 = "Summarize the introduction section." # Another example

query\_rag(rag\_chain, query\_question\_2)

Run the complete script (python rag\_local.py). Make sure that the ollama serve command is running in another terminal. The script will load the PDF, split it, embed the chunks using nomic-embed-text via Ollama, store them in ChromaDB, build the RAG chain using qwen3:8b via Ollama, and finally execute the queries. It’ll print the LLM's responses based on the document content.

**How to Create Local AI Agents with Qwen 3**

Beyond answering questions based on provided text, LLMs can act as the reasoning engine for AI agents. Agents can plan sequences of actions, interact with external tools (like functions or APIs), and work towards accomplishing more complex goals assigned by the user.

Qwen 3 models were specifically designed with strong tool-calling and agentic capabilities. While Alibaba provides the Qwen-Agent framework, this tutorial will continue using LangChain for consistency and because its integration with Ollama for agent tasks is more readily documented in the provided materials.

We will build a simple agent that can use a custom Python function as a tool.

**Step 1: Define Custom Tools**

Tools are standard Python functions that the agent can choose to execute. The function's docstring is crucial, as the LLM uses it to understand what the tool does and what arguments it requires. LangChain's @tool decorator simplifies wrapping functions for agent use.

# agent\_local.py

import os

from dotenv import load\_dotenv

from langchain.agents import tool

import datetime

load\_dotenv() # Optional

@tool

def get\_current\_datetime(format: str = "%Y-%m-%d %H:%M:%S") -> str:

"""

Returns the current date and time, formatted according to the provided Python strftime format string.

Use this tool whenever the user asks for the current date, time, or both.

Example format strings: '%Y-%m-%d' for date, '%H:%M:%S' for time.

If no format is specified, defaults to '%Y-%m-%d %H:%M:%S'.

"""

try:

return datetime.datetime.now().strftime(format)

except Exception as e:

return f"Error formatting date/time: {e}"

# List of tools the agent can use

tools = [get\_current\_datetime]

print("Custom tool defined.")

**Step 2: Set Up the Agent LLM**

Instantiate the ChatOllama model again, using a Qwen 3 variant suitable for tool calling. The qwen3:8b model should be capable of handling simple tool use cases.

It's important to note that tool calling reliability with local models served via Ollama can sometimes be less consistent than with large commercial APIs like GPT-4 or Claude. The LLM might fail to recognize when a tool is needed, hallucinate arguments, or misinterpret the tool's output. Starting with clear prompts and simple tools is recommended.

# agent\_local.py (continued)

from langchain\_ollama import ChatOllama

def get\_agent\_llm(model\_name="qwen3:8b", temperature=0):

"""Initializes the ChatOllama model for the agent."""

# Ensure Ollama server is running (ollama serve)

llm = ChatOllama(

model=model\_name,

temperature=temperature # Lower temperature for more predictable tool use

# Consider increasing num\_ctx if expecting long conversations or complex reasoning

# num\_ctx=8192

)

print(f"Initialized ChatOllama agent LLM with model: {model\_name}")

return llm

# agent\_llm = get\_agent\_llm() # Call this later

**Step 3: Create the Agent Prompt**

Agents require specific prompt structures that guide their reasoning and tool use. The prompt typically includes placeholders for user input (input), conversation history (chat\_history), and the agent\_scratchpad. The scratchpad is where the agent records its internal "thought" process, the tools it decides to call, and the results (observations) it gets back from those tools. LangChain Hub provides pre-built prompts suitable for tool-calling agents.

# agent\_local.py (continued)

from langchain import hub

def get\_agent\_prompt(prompt\_hub\_name="hwchase17/openai-tools-agent"):

"""Pulls the agent prompt template from LangChain Hub."""

# This prompt is designed for OpenAI but often works well with other tool-calling models.

# Alternatively, define a custom ChatPromptTemplate.

prompt = hub.pull(prompt\_hub\_name)

print(f"Pulled agent prompt from Hub: {prompt\_hub\_name}")

# print("Prompt Structure:")

# prompt.pretty\_print() # Uncomment to see the prompt structure

return prompt

# agent\_prompt = get\_agent\_prompt() # Call this later

**Step 4: Build the Agent**

The create\_tool\_calling\_agent function combines the LLM, the defined tools, and the prompt into a runnable unit that represents the agent's core logic.

# agent\_local.py (continued)

from langchain.agents import create\_tool\_calling\_agent

def build\_agent(llm, tools, prompt):

"""Builds the tool-calling agent runnable."""

agent = create\_tool\_calling\_agent(llm, tools, prompt)

print("Agent runnable created.")

return agent

# agent\_runnable = build\_agent(agent\_llm, tools, agent\_prompt) # Call this later

**Step 5: Create the Agent Executor**

The AgentExecutor is responsible for running the agent loop. It takes the agent runnable and the tools, invokes the agent with the input, parses the agent's output (which could be a final answer or a tool call request), executes any requested tool calls, and feeds the results back to the agent until a final answer is reached. Setting verbose=True is highly recommended during development to observe the agent's step-by-step execution flow.

# agent\_local.py (continued)

from langchain.agents import AgentExecutor

def create\_agent\_executor(agent, tools):

"""Creates the agent executor."""

agent\_executor = AgentExecutor(

agent=agent,

tools=tools,

verbose=True # Set to True to see agent thoughts and tool calls

)

print("Agent executor created.")

return agent\_executor

# agent\_executor = create\_agent\_executor(agent\_runnable, tools) # Call this later

**Step 6: Run the Agent**

Invoke the agent executor with a user query that should trigger the use of the defined tool.

# agent\_local.py (continued)

def run\_agent(executor, user\_input):

"""Runs the agent executor with the given input."""

print("\nInvoking agent...")

print(f"Input: {user\_input}")

response = executor.invoke({"input": user\_input})

print("\nAgent Response:")

print(response['output'])

# --- Main Execution ---

if \_\_name\_\_ == "\_\_main\_\_":

# 1. Define Tools (already done above)

# 2. Get Agent LLM

agent\_llm = get\_agent\_llm(model\_name="qwen3:8b") # Use the chosen Qwen 3 model

# 3. Get Agent Prompt

agent\_prompt = get\_agent\_prompt()

# 4. Build Agent Runnable

agent\_runnable = build\_agent(agent\_llm, tools, agent\_prompt)

# 5. Create Agent Executor

agent\_executor = create\_agent\_executor(agent\_runnable, tools)

# 6. Run Agent

run\_agent(agent\_executor, "What is the current date?")

run\_agent(agent\_executor, "What time is it right now? Use HH:MM format.")

run\_agent(agent\_executor, "Tell me a joke.") # Should not use the tool

Running python agent\_local.py (with ollama serve active) will execute the agent. The verbose=True setting will print output resembling the ReAct (Reasoning and Acting) framework, showing the agent's internal "Thoughts" on how to proceed, the "Action" it decides to take (calling a specific tool with arguments), and the "Observation" (the result returned by the tool).

Building reliable agents with local models presents unique challenges. The LLM's ability to correctly interpret the prompt, understand when to use tools, select the right tool, generate valid arguments, and process the tool's output is critical.

Local models, especially smaller or heavily quantized ones, might struggle with these reasoning steps compared to larger, cloud-based counterparts. If the qwen3:8b model proves unreliable for more complex agentic tasks, consider trying qwen3:14b or the efficient qwen3:30b-a3b if hardware permits.

For highly complex or stateful agent workflows, exploring frameworks like LangGraph, which offers more control over the agent's execution flow, might be beneficial.

**Advanced Considerations and Troubleshooting**

Running LLMs locally offers great flexibility but also introduces specific configuration aspects and potential issues.

**Controlling Qwen 3's Thinking Mode with Ollama**

Qwen 3's unique hybrid inference allows switching between a deep "thinking" mode for complex reasoning and a faster "non-thinking" mode for general chat. While frameworks like Hugging Face Transformers or vLLM might offer explicit parameters (enable\_thinking), the primary way to control this when using Ollama appears to be through "soft switches" embedded in the prompt.

Append /think to the end of a user prompt to encourage step-by-step reasoning, or /no\_think to request a faster, direct response. You can do this via the Ollama CLI or potentially within the prompts sent via the API/LangChain.

# Example using LangChain's ChatOllama

from langchain\_ollama import ChatOllama

llm\_think = ChatOllama(model="qwen3:8b")

llm\_no\_think = ChatOllama(model="qwen3:8b") # Could also set system prompt

# Invoke with prompt modification

response\_think = llm\_think.invoke("Solve the equation 2x + 5 = 15 /think")

print("Thinking Response:", response\_think)

response\_no\_think = llm\_no\_think.invoke("What is the capital of France? /no\_think")

print("Non-Thinking Response:", response\_no\_think)

# Alternatively, set via system message (might be less reliable turn-by-turn)

llm\_system\_no\_think = ChatOllama(model="qwen3:8b", system="/no\_think")

response\_system = llm\_system\_no\_think.invoke("What is 2+2?")

print("System No-Think Response:", response\_system)

Note that the persistence of these tags across multiple turns in a conversation might require careful prompt management.

**Managing Context Length (num\_ctx)**

The context window (num\_ctx) determines how much information (prompt, history, retrieved documents) the LLM can consider at once. Qwen 3 models (8B+) support large native context lengths (for example, 128k tokens), but Ollama often defaults to a much smaller window (like 2048 or 4096). For RAG or conversations requiring memory of earlier turns, this default is often insufficient.

Set num\_ctx when initializing ChatOllama or OllamaLLM in LangChain:

# Example setting context window to 8192 tokens

llm = ChatOllama(model="qwen3:8b", num\_ctx=8192)

Be mindful that larger num\_ctx values significantly increase RAM and VRAM consumption. But setting it too low can lead to the model "forgetting" context or even entering repetitive loops. Choose a value that balances the task requirements with hardware capabilities.

**Hardware Limitations and VRAM**

Running LLMs locally is resource-intensive.

* **VRAM:** A dedicated GPU (NVIDIA or Apple Silicon) with sufficient VRAM is highly recommended for acceptable performance. The amount of VRAM dictates the largest model size that can run efficiently. Refer to the table in Section 2 for estimates.
* **RAM:** System RAM is also crucial, especially if the model doesn't fit entirely in VRAM. Ollama can utilize system RAM as a fallback, but this is significantly slower.
* **Quantization:** Ollama typically serves quantized models (for example., 4-bit or 5-bit), which reduce the model size and VRAM requirements significantly compared to full-precision models, often with minimal performance degradation for many tasks. The tags like :4b, :8b usually imply a default quantization level.

If performance is slow or errors occur due to resource constraints, consider:

* Using a smaller Qwen 3 model (like 4B instead of 8B).
* Ensuring Ollama is correctly detecting and utilizing the GPU (check Ollama logs or system monitoring tools).
* Closing other resource-intensive applications.

**Conclusion and Next Steps**

This tutorial gave you a practical walkthrough for setting up your local AI environment using the powerful and open Qwen 3 LLM family with the user-friendly Ollama tool.

If you’ve followed these steps, you should have successfully:

1. Installed Ollama and downloaded/run Qwen 3 models locally.
2. Built a functional Retrieval-Augmented Generation (RAG) pipeline using LangChain and ChromaDB to query local documents.
3. Created a basic AI agent capable of reasoning and utilizing custom Python tools.

Running these systems locally unlocks significant advantages in privacy, cost, and customization, making advanced AI capabilities more accessible than ever. The combination of Qwen 3's performance and open license with Ollama's ease of use creates a potent platform for experimentation and development.

**Official Resources:**

* **Qwen 3:** [GitHub](https://github.com/QwenLM/Qwen3), [Documentation](https://qwen.readthedocs.io/en/latest/" \t "_blank)

**Ollama:** [Website](https://ollama.com/), [Model Library](https://ollama.com/library), [GitHub](https://github.com/ollama/ollama)

* **LangChain:** [Python Documentation](https://python.langchain.com/docs/get_started/introduction)
* **ChromaDB:** [Documentation](https://docs.trychroma.com/" \t "_blank)
* **Sentence Transformers:** [Documentation](https://www.sbert.net/" \t "_blank)

By leveraging these powerful, free, and open-source tools, you can continue to push the boundaries of what's possible with AI running directly on your own hardware.

# Build an Agent

LangChain supports the creation of [agents](https://github.com/langchain-ai/langchain/blob/5ecbb5f2772a216c1d6604793fa50348c85d1c52/docs/docs/tutorials/docs/concepts/agents), or systems that use [LLMs](https://github.com/langchain-ai/langchain/blob/5ecbb5f2772a216c1d6604793fa50348c85d1c52/docs/docs/tutorials/docs/concepts/chat_models) as reasoning engines to determine which actions to take and the inputs necessary to perform the action. After executing actions, the results can be fed back into the LLM to determine whether more actions are needed, or whether it is okay to finish. This is often achieved via [tool-calling](https://github.com/langchain-ai/langchain/blob/5ecbb5f2772a216c1d6604793fa50348c85d1c52/docs/docs/tutorials/docs/concepts/tool_calling).

In this tutorial we will build an agent that can interact with a search engine. You will be able to ask this agent questions, watch it call the search tool, and have conversations with it.

## End-to-end agent

The code snippet below represents a fully functional agent that uses an LLM to decide which tools to use. It is equipped with a generic search tool. It has conversational memory - meaning that it can be used as a multi-turn chatbot.

In the rest of the guide, we will walk through the individual components and what each part does - but if you want to just grab some code and get started, feel free to use this!

*# Import relevant functionality*

**from** langchain.chat\_models **import** init\_chat\_model

**from** langchain\_tavily **import** TavilySearch

**from** langgraph.checkpoint.memory **import** MemorySaver

**from** langgraph.prebuilt **import** create\_react\_agent

*# Create the agent*

memory **=** MemorySaver()

model **=** init\_chat\_model("anthropic:claude-3-5-sonnet-latest")

search **=** TavilySearch(max\_results**=**2)

tools **=** [search]

agent\_executor **=** create\_react\_agent(model, tools, checkpointer**=**memory)

*# Use the agent*

config **=** {"configurable": {"thread\_id": "abc123"}}

input\_message **=** {

"role": "user",

"content": "Hi, I'm Bob and I live in SF.",

}

**for** step **in** agent\_executor**.**stream(

{"messages": [input\_message]}, config, stream\_mode**=**"values"

):

step["messages"][**-**1]**.**pretty\_print()

================================ **Human Message** =================================

Hi, I'm Bob and I live in SF.

================================== **Ai Message** ==================================

Hello Bob! I notice you've introduced yourself and mentioned you live in SF (San Francisco), but you haven't asked a specific question or made a request that requires the use of any tools. Is there something specific you'd like to know about San Francisco or any other topic? I'd be happy to help you find information using the available search tools.

input\_message **=** {

"role": "user",

"content": "What's the weather where I live?",

}

**for** step **in** agent\_executor**.**stream(

{"messages": [input\_message]}, config, stream\_mode**=**"values"

):

step["messages"][**-**1]**.**pretty\_print()

================================ **Human Message** =================================

What's the weather where I live?

================================== **Ai Message** ==================================

[{'text': 'Let me search for current weather information in San Francisco.', 'type': 'text'}, {'id': 'toolu\_011kSdheoJp8THURoLmeLtZo', 'input': {'query': 'current weather San Francisco CA'}, 'name': 'tavily\_search', 'type': 'tool\_use'}]

Tool Calls:

tavily\_search (toolu\_011kSdheoJp8THURoLmeLtZo)

Call ID: toolu\_011kSdheoJp8THURoLmeLtZo

Args:

query: current weather San Francisco CA

================================= **Tool Message** =================================

Name: tavily\_search

{"query": "current weather San Francisco CA", "follow\_up\_questions": null, "answer": null, "images": [], "results": [{"title": "Weather in San Francisco, CA", "url": "https://www.weatherapi.com/", "content": "{'location': {'name': 'San Francisco', 'region': 'California', 'country': 'United States of America', 'lat': 37.775, 'lon': -122.4183, 'tz\_id': 'America/Los\_Angeles', 'localtime\_epoch': 1750168606, 'localtime': '2025-06-17 06:56'}, 'current': {'last\_updated\_epoch': 1750167900, 'last\_updated': '2025-06-17 06:45', 'temp\_c': 11.7, 'temp\_f': 53.1, 'is\_day': 1, 'condition': {'text': 'Fog', 'icon': '//cdn.weatherapi.com/weather/64x64/day/248.png', 'code': 1135}, 'wind\_mph': 4.0, 'wind\_kph': 6.5, 'wind\_degree': 215, 'wind\_dir': 'SW', 'pressure\_mb': 1017.0, 'pressure\_in': 30.02, 'precip\_mm': 0.0, 'precip\_in': 0.0, 'humidity': 86, 'cloud': 0, 'feelslike\_c': 11.3, 'feelslike\_f': 52.4, 'windchill\_c': 8.7, 'windchill\_f': 47.7, 'heatindex\_c': 9.8, 'heatindex\_f': 49.7, 'dewpoint\_c': 9.6, 'dewpoint\_f': 49.2, 'vis\_km': 16.0, 'vis\_miles': 9.0, 'uv': 0.0, 'gust\_mph': 6.3, 'gust\_kph': 10.2}}", "score": 0.944705, "raw\_content": null}, {"title": "Weather in San Francisco in June 2025", "url": "https://world-weather.info/forecast/usa/san\_francisco/june-2025/", "content": "Detailed ⚡ San Francisco Weather Forecast for June 2025 - day/night 🌡️ temperatures, precipitations - World-Weather.info. Add the current city. Search. Weather; Archive; Weather Widget °F. World; United States; California; Weather in San Francisco; ... 17 +64° +54° 18 +61° +54° 19", "score": 0.86441374, "raw\_content": null}], "response\_time": 2.34}

================================== **Ai Message** ==================================

Based on the search results, here's the current weather in San Francisco:

- Temperature: 53.1°F (11.7°C)

- Condition: Foggy

- Wind: 4.0 mph from the Southwest

- Humidity: 86%

- Visibility: 9 miles

This is quite typical weather for San Francisco, with the characteristic fog that the city is known for. Would you like to know anything else about the weather or San Francisco in general?

## Setup

### Jupyter Notebook

This guide (and most of the other guides in the documentation) uses [Jupyter notebooks](https://jupyter.org/) and assumes the reader is as well. Jupyter notebooks are perfect interactive environments for learning how to work with LLM systems because oftentimes things can go wrong (unexpected output, API down, etc), and observing these cases is a great way to better understand building with LLMs.

This and other tutorials are perhaps most conveniently run in a Jupyter notebook. See [here](https://jupyter.org/install) for instructions on how to install.

### Installation

To install LangChain run:

**%pip** install -U langgraph langchain-tavily langgraph-checkpoint-sqlite

For more details, see our [Installation guide](https://github.com/langchain-ai/langchain/blob/5ecbb5f2772a216c1d6604793fa50348c85d1c52/docs/docs/tutorials/docs/how_to/installation).

### LangSmith

Many of the applications you build with LangChain will contain multiple steps with multiple invocations of LLM calls. As these applications get more and more complex, it becomes crucial to be able to inspect what exactly is going on inside your chain or agent. The best way to do this is with [LangSmith](https://smith.langchain.com/).

After you sign up at the link above, make sure to set your environment variables to start logging traces:

export LANGSMITH\_TRACING**=**"true"

export LANGSMITH\_API\_KEY**=**"..."

Or, if in a notebook, you can set them with:

**import** getpass

**import** os

os**.**environ["LANGSMITH\_TRACING"] **=** "true"

os**.**environ["LANGSMITH\_API\_KEY"] **=** getpass**.**getpass()

### Tavily

We will be using [Tavily](https://github.com/langchain-ai/langchain/blob/5ecbb5f2772a216c1d6604793fa50348c85d1c52/docs/docs/tutorials/docs/integrations/tools/tavily_search) (a search engine) as a tool. In order to use it, you will need to get and set an API key:

export TAVILY\_API\_KEY**=**"..."

Or, if in a notebook, you can set it with:

**import** getpass

**import** os

os**.**environ["TAVILY\_API\_KEY"] **=** getpass**.**getpass()

## Define tools

We first need to create the tools we want to use. Our main tool of choice will be [Tavily](https://github.com/langchain-ai/langchain/blob/5ecbb5f2772a216c1d6604793fa50348c85d1c52/docs/docs/tutorials/docs/integrations/tools/tavily_search) - a search engine. We can use the dedicated [langchain-tavily](https://pypi.org/project/langchain-tavily/) [integration package](https://github.com/langchain-ai/langchain/blob/5ecbb5f2772a216c1d6604793fa50348c85d1c52/docs/docs/tutorials/docs/concepts/architecture/" \l "integration-packages) to easily use Tavily search engine as tool with LangChain.

**from** langchain\_tavily **import** TavilySearch

search **=** TavilySearch(max\_results**=**2)

search\_results **=** search**.**invoke("What is the weather in SF")

print(search\_results)

*# If we want, we can create other tools.*

*# Once we have all the tools we want, we can put them in a list that we will reference later.*

tools **=** [search]

{'query': 'What is the weather in SF', 'follow\_up\_questions': None, 'answer': None, 'images': [], 'results': [{'title': 'Weather in San Francisco, CA', 'url': 'https://www.weatherapi.com/', 'content': "{'location': {'name': 'San Francisco', 'region': 'California', 'country': 'United States of America', 'lat': 37.775, 'lon': -122.4183, 'tz\_id': 'America/Los\_Angeles', 'localtime\_epoch': 1750168606, 'localtime': '2025-06-17 06:56'}, 'current': {'last\_updated\_epoch': 1750167900, 'last\_updated': '2025-06-17 06:45', 'temp\_c': 11.7, 'temp\_f': 53.1, 'is\_day': 1, 'condition': {'text': 'Fog', 'icon': '//cdn.weatherapi.com/weather/64x64/day/248.png', 'code': 1135}, 'wind\_mph': 4.0, 'wind\_kph': 6.5, 'wind\_degree': 215, 'wind\_dir': 'SW', 'pressure\_mb': 1017.0, 'pressure\_in': 30.02, 'precip\_mm': 0.0, 'precip\_in': 0.0, 'humidity': 86, 'cloud': 0, 'feelslike\_c': 11.3, 'feelslike\_f': 52.4, 'windchill\_c': 8.7, 'windchill\_f': 47.7, 'heatindex\_c': 9.8, 'heatindex\_f': 49.7, 'dewpoint\_c': 9.6, 'dewpoint\_f': 49.2, 'vis\_km': 16.0, 'vis\_miles': 9.0, 'uv': 0.0, 'gust\_mph': 6.3, 'gust\_kph': 10.2}}", 'score': 0.9185379, 'raw\_content': None}, {'title': 'Weather in San Francisco in June 2025', 'url': 'https://world-weather.info/forecast/usa/san\_francisco/june-2025/', 'content': "Weather in San Francisco in June 2025 (California) - Detailed Weather Forecast for a Month \* Weather in San Francisco Weather in San Francisco in June 2025 \* 1 +63° +55° \* 2 +66° +54° \* 3 +66° +55° \* 4 +66° +54° \* 5 +66° +55° \* 6 +66° +57° \* 7 +64° +55° \* 8 +63° +55° \* 9 +63° +54° \* 10 +59° +54° \* 11 +59° +54° \* 12 +61° +54° Weather in Washington, D.C.\*\*+68°\*\* Sacramento\*\*+81°\*\* Pleasanton\*\*+72°\*\* Redwood City\*\*+68°\*\* San Leandro\*\*+61°\*\* San Mateo\*\*+64°\*\* San Rafael\*\*+70°\*\* San Ramon\*\*+64°\*\* South San Francisco\*\*+61°\*\* Daly City\*\*+59°\*\* Wilder\*\*+66°\*\* Woodacre\*\*+70°\*\* world's temperature today Colchani day+50°F night+16°F Az Zubayr day+124°F night+93°F Weather forecast on your site Install \_San Francisco\_ +61° Temperature units", 'score': 0.7978881, 'raw\_content': None}], 'response\_time': 2.62}

:::tip

In many applications, you may want to define custom tools. LangChain supports custom tool creation via Python functions and other means. Refer to the [How to create tools](https://github.com/langchain-ai/langchain/blob/5ecbb5f2772a216c1d6604793fa50348c85d1c52/docs/docs/tutorials/docs/how_to/custom_tools/) guide for details.

:::

## Using Language Models

Next, let's learn how to use a language model to call tools. LangChain supports many different language models that you can use interchangably - select the one you want to use below!

import ChatModelTabs from "@theme/ChatModelTabs";

<ChatModelTabs overrideParams={{openai: {model: "gpt-4.1"}}} />

*# | output: false*

*# | echo: false*

**from** langchain\_anthropic **import** ChatAnthropic

model **=** ChatAnthropic(model**=**"claude-3-5-sonnet-latest")

You can call the language model by passing in a list of messages. By default, the response is a content string.

query **=** "Hi!"

response **=** model**.**invoke([{"role": "user", "content": query}])

response**.**text()

'Hello! How can I help you today?'

We can now see what it is like to enable this model to do tool calling. In order to enable that we use .bind\_tools to give the language model knowledge of these tools

model\_with\_tools **=** model**.**bind\_tools(tools)

We can now call the model. Let's first call it with a normal message, and see how it responds. We can look at both the content field as well as the tool\_calls field.

query **=** "Hi!"

response **=** model\_with\_tools**.**invoke([{"role": "user", "content": query}])

print(f"Message content: {response**.**text()}\n")

print(f"Tool calls: {response**.**tool\_calls}")

Message content: Hello! I'm here to help you. I have access to a powerful search tool that can help answer questions and find information about various topics. What would you like to know about?

Feel free to ask any question or request information, and I'll do my best to assist you using the available tools.

Tool calls: []

Now, let's try calling it with some input that would expect a tool to be called.

query **=** "Search for the weather in SF"

response **=** model\_with\_tools**.**invoke([{"role": "user", "content": query}])

print(f"Message content: {response**.**text()}\n")

print(f"Tool calls: {response**.**tool\_calls}")

Message content: I'll help you search for information about the weather in San Francisco.

Tool calls: [{'name': 'tavily\_search', 'args': {'query': 'current weather San Francisco'}, 'id': 'toolu\_015gdPn1jbB2Z21DmN2RAnti', 'type': 'tool\_call'}]

We can see that there's now no text content, but there is a tool call! It wants us to call the Tavily Search tool.

This isn't calling that tool yet - it's just telling us to. In order to actually call it, we'll want to create our agent.

## Create the agent

Now that we have defined the tools and the LLM, we can create the agent. We will be using [LangGraph](https://github.com/langchain-ai/langchain/blob/5ecbb5f2772a216c1d6604793fa50348c85d1c52/docs/docs/tutorials/docs/concepts/architecture/" \l "langgraph) to construct the agent. Currently, we are using a high level interface to construct the agent, but the nice thing about LangGraph is that this high-level interface is backed by a low-level, highly controllable API in case you want to modify the agent logic.

Now, we can initialize the agent with the LLM and the tools.

Note that we are passing in the model, not model\_with\_tools. That is because create\_react\_agent will call .bind\_tools for us under the hood.

**from** langgraph.prebuilt **import** create\_react\_agent

agent\_executor **=** create\_react\_agent(model, tools)

## Run the agent

We can now run the agent with a few queries! Note that for now, these are all **stateless** queries (it won't remember previous interactions). Note that the agent will return the **final** state at the end of the interaction (which includes any inputs, we will see later on how to get only the outputs).

First up, let's see how it responds when there's no need to call a tool:

input\_message **=** {"role": "user", "content": "Hi!"}

response **=** agent\_executor**.**invoke({"messages": [input\_message]})

**for** message **in** response["messages"]:

message**.**pretty\_print()

================================ **Human Message** =================================

Hi!

================================== **Ai Message** ==================================

Hello! I'm here to help you with your questions using the available search tools. Please feel free to ask any question, and I'll do my best to find relevant and accurate information for you.

In order to see exactly what is happening under the hood (and to make sure it's not calling a tool) we can take a look at the [LangSmith trace](https://smith.langchain.com/public/28311faa-e135-4d6a-ab6b-caecf6482aaa/r)

Let's now try it out on an example where it should be invoking the tool

input\_message **=** {"role": "user", "content": "Search for the weather in SF"}

response **=** agent\_executor**.**invoke({"messages": [input\_message]})

**for** message **in** response["messages"]:

message**.**pretty\_print()

================================ **Human Message** =================================

Search for the weather in SF

================================== **Ai Message** ==================================

[{'text': "I'll help you search for weather information in San Francisco. Let me use the search engine to find current weather conditions.", 'type': 'text'}, {'id': 'toolu\_01WWcXGnArosybujpKzdmARZ', 'input': {'query': 'current weather San Francisco SF'}, 'name': 'tavily\_search', 'type': 'tool\_use'}]

Tool Calls:

tavily\_search (toolu\_01WWcXGnArosybujpKzdmARZ)

Call ID: toolu\_01WWcXGnArosybujpKzdmARZ

Args:

query: current weather San Francisco SF

================================= **Tool Message** =================================

Name: tavily\_search

{"query": "current weather San Francisco SF", "follow\_up\_questions": null, "answer": null, "images": [], "results": [{"title": "Weather in San Francisco, CA", "url": "https://www.weatherapi.com/", "content": "{'location': {'name': 'San Francisco', 'region': 'California', 'country': 'United States of America', 'lat': 37.775, 'lon': -122.4183, 'tz\_id': 'America/Los\_Angeles', 'localtime\_epoch': 1750168606, 'localtime': '2025-06-17 06:56'}, 'current': {'last\_updated\_epoch': 1750167900, 'last\_updated': '2025-06-17 06:45', 'temp\_c': 11.7, 'temp\_f': 53.1, 'is\_day': 1, 'condition': {'text': 'Fog', 'icon': '//cdn.weatherapi.com/weather/64x64/day/248.png', 'code': 1135}, 'wind\_mph': 4.0, 'wind\_kph': 6.5, 'wind\_degree': 215, 'wind\_dir': 'SW', 'pressure\_mb': 1017.0, 'pressure\_in': 30.02, 'precip\_mm': 0.0, 'precip\_in': 0.0, 'humidity': 86, 'cloud': 0, 'feelslike\_c': 11.3, 'feelslike\_f': 52.4, 'windchill\_c': 8.7, 'windchill\_f': 47.7, 'heatindex\_c': 9.8, 'heatindex\_f': 49.7, 'dewpoint\_c': 9.6, 'dewpoint\_f': 49.2, 'vis\_km': 16.0, 'vis\_miles': 9.0, 'uv': 0.0, 'gust\_mph': 6.3, 'gust\_kph': 10.2}}", "score": 0.885373, "raw\_content": null}, {"title": "Weather in San Francisco in June 2025", "url": "https://world-weather.info/forecast/usa/san\_francisco/june-2025/", "content": "Detailed ⚡ San Francisco Weather Forecast for June 2025 - day/night 🌡️ temperatures, precipitations - World-Weather.info. Add the current city. Search. Weather; Archive; Weather Widget °F. World; United States; California; Weather in San Francisco; ... 17 +64° +54° 18 +61° +54° 19", "score": 0.8830044, "raw\_content": null}], "response\_time": 2.6}

================================== **Ai Message** ==================================

Based on the search results, here's the current weather in San Francisco:

- Temperature: 53.1°F (11.7°C)

- Conditions: Foggy

- Wind: 4.0 mph from the SW

- Humidity: 86%

- Visibility: 9.0 miles

The weather appears to be typical for San Francisco, with morning fog and mild temperatures. The "feels like" temperature is 52.4°F (11.3°C).

We can check out the [LangSmith trace](https://smith.langchain.com/public/f520839d-cd4d-4495-8764-e32b548e235d/r) to make sure it's calling the search tool effectively.

## Streaming Messages

We've seen how the agent can be called with .invoke to get a final response. If the agent executes multiple steps, this may take a while. To show intermediate progress, we can stream back messages as they occur.

**for** step **in** agent\_executor**.**stream({"messages": [input\_message]}, stream\_mode**=**"values"):

step["messages"][**-**1]**.**pretty\_print()

================================ **Human Message** =================================

Search for the weather in SF

================================== **Ai Message** ==================================

[{'text': "I'll help you search for information about the weather in San Francisco.", 'type': 'text'}, {'id': 'toolu\_01DCPnJES53Fcr7YWnZ47kDG', 'input': {'query': 'current weather San Francisco'}, 'name': 'tavily\_search', 'type': 'tool\_use'}]

Tool Calls:

tavily\_search (toolu\_01DCPnJES53Fcr7YWnZ47kDG)

Call ID: toolu\_01DCPnJES53Fcr7YWnZ47kDG

Args:

query: current weather San Francisco

================================= **Tool Message** =================================

Name: tavily\_search

{"query": "current weather San Francisco", "follow\_up\_questions": null, "answer": null, "images": [], "results": [{"title": "Weather in San Francisco", "url": "https://www.weatherapi.com/", "content": "{'location': {'name': 'San Francisco', 'region': 'California', 'country': 'United States of America', 'lat': 37.775, 'lon': -122.4183, 'tz\_id': 'America/Los\_Angeles', 'localtime\_epoch': 1750168506, 'localtime': '2025-06-17 06:55'}, 'current': {'last\_updated\_epoch': 1750167900, 'last\_updated': '2025-06-17 06:45', 'temp\_c': 11.7, 'temp\_f': 53.1, 'is\_day': 1, 'condition': {'text': 'Fog', 'icon': '//cdn.weatherapi.com/weather/64x64/day/248.png', 'code': 1135}, 'wind\_mph': 4.0, 'wind\_kph': 6.5, 'wind\_degree': 215, 'wind\_dir': 'SW', 'pressure\_mb': 1017.0, 'pressure\_in': 30.02, 'precip\_mm': 0.0, 'precip\_in': 0.0, 'humidity': 86, 'cloud': 0, 'feelslike\_c': 11.3, 'feelslike\_f': 52.4, 'windchill\_c': 8.7, 'windchill\_f': 47.7, 'heatindex\_c': 9.8, 'heatindex\_f': 49.7, 'dewpoint\_c': 9.6, 'dewpoint\_f': 49.2, 'vis\_km': 16.0, 'vis\_miles': 9.0, 'uv': 0.0, 'gust\_mph': 6.3, 'gust\_kph': 10.2}}", "score": 0.9542825, "raw\_content": null}, {"title": "Weather in San Francisco in June 2025", "url": "https://world-weather.info/forecast/usa/san\_francisco/june-2025/", "content": "Detailed ⚡ San Francisco Weather Forecast for June 2025 - day/night 🌡️ temperatures, precipitations - World-Weather.info. Add the current city. Search. Weather; Archive; Weather Widget °F. World; United States; California; Weather in San Francisco; ... 17 +64° +54° 18 +61° +54° 19", "score": 0.8638634, "raw\_content": null}], "response\_time": 2.57}

================================== **Ai Message** ==================================

Based on the search results, here's the current weather in San Francisco:

- Temperature: 53.1°F (11.7°C)

- Condition: Foggy

- Wind: 4.0 mph from the Southwest

- Humidity: 86%

- Visibility: 9.0 miles

- Feels like: 52.4°F (11.3°C)

This is quite typical weather for San Francisco, which is known for its fog, especially during the morning hours. The city's proximity to the ocean and unique geographical features often result in mild temperatures and foggy conditions.

## Streaming tokens

In addition to streaming back messages, it is also useful to stream back tokens. We can do this by specifying stream\_mode="messages".

::: note

Below we use message.text(), which requires langchain-core>=0.3.37.

:::

**for** step, metadata **in** agent\_executor**.**stream(

{"messages": [input\_message]}, stream\_mode**=**"messages"

):

**if** metadata["langgraph\_node"] **==** "agent" **and** (text **:=** step**.**text()):

print(text, end**=**"|")

I|'ll help you search for information| about the weather in San Francisco.|Base|d on the search results, here|'s the current weather in| San Francisco:

-| Temperature: 53.1°F (|11.7°C)

-| Condition: Foggy

- Wind:| 4.0 mph from| the Southwest

- Humidity|: 86%|

- Visibility: 9|.0 miles

- Pressure: |30.02 in|Hg

The weather| is characteristic of San Francisco, with| foggy conditions and mild temperatures|. The "feels like" temperature is slightly| lower at 52.4|°F (11.|3°C)| due to the wind chill effect|.|

## Adding in memory

As mentioned earlier, this agent is stateless. This means it does not remember previous interactions. To give it memory we need to pass in a checkpointer. When passing in a checkpointer, we also have to pass in a thread\_id when invoking the agent (so it knows which thread/conversation to resume from).

**from** langgraph.checkpoint.memory **import** MemorySaver

memory **=** MemorySaver()

agent\_executor **=** create\_react\_agent(model, tools, checkpointer**=**memory)

config **=** {"configurable": {"thread\_id": "abc123"}}

input\_message **=** {"role": "user", "content": "Hi, I'm Bob!"}

**for** step **in** agent\_executor**.**stream(

{"messages": [input\_message]}, config, stream\_mode**=**"values"

):

step["messages"][**-**1]**.**pretty\_print()

================================ **Human Message** =================================

Hi, I'm Bob!

================================== **Ai Message** ==================================

Hello Bob! I'm an AI assistant who can help you search for information using specialized search tools. Is there anything specific you'd like to know about or search for? I'm happy to help you find accurate and up-to-date information on various topics.

input\_message **=** {"role": "user", "content": "What's my name?"}

**for** step **in** agent\_executor**.**stream(

{"messages": [input\_message]}, config, stream\_mode**=**"values"

):

step["messages"][**-**1]**.**pretty\_print()

================================ **Human Message** =================================

What's my name?

================================== **Ai Message** ==================================

Your name is Bob, as you introduced yourself earlier. I can remember information shared within our conversation without needing to search for it.

Example [LangSmith trace](https://smith.langchain.com/public/fa73960b-0f7d-4910-b73d-757a12f33b2b/r)

If you want to start a new conversation, all you have to do is change the thread\_id used

*# highlight-next-line*

config **=** {"configurable": {"thread\_id": "xyz123"}}

input\_message **=** {"role": "user", "content": "What's my name?"}

**for** step **in** agent\_executor**.**stream(

{"messages": [input\_message]}, config, stream\_mode**=**"values"

):

step["messages"][**-**1]**.**pretty\_print()

================================ **Human Message** =================================

What's my name?

================================== **Ai Message** ==================================

I apologize, but I don't have access to any tools that would tell me your name. I can only assist you with searching for publicly available information using the tavily\_search function. I don't have access to personal information about users. If you'd like to tell me your name, I'll be happy to address you by it.

## Conclusion

That's a wrap! In this quick start we covered how to create a simple agent. We've then shown how to stream back a response - not only with the intermediate steps, but also tokens! We've also added in memory so you can have a conversation with them. Agents are a complex topic with lots to learn!

For more information on Agents, please check out the [LangGraph](https://github.com/langchain-ai/langchain/blob/5ecbb5f2772a216c1d6604793fa50348c85d1c52/docs/docs/tutorials/docs/concepts/architecture/" \l "langgraph) documentation. This has it's own set of concepts, tutorials, and how-to guides.

# Run models locally

## Use case

The popularity of projects like [llama.cpp](https://github.com/ggerganov/llama.cpp), [Ollama](https://github.com/ollama/ollama), [GPT4All](https://github.com/nomic-ai/gpt4all), [llamafile](https://github.com/Mozilla-Ocho/llamafile), and others underscore the demand to run LLMs locally (on your own device).

This has at least two important benefits:

1. **Privacy**: Your data is not sent to a third party, and it is not subject to the terms of service of a commercial service
2. **Cost**: There is no inference fee, which is important for token-intensive applications (e.g., [long-running simulations](https://twitter.com/RLanceMartin/status/1691097659262820352?s=20), summarization)

## Overview

Running an LLM locally requires a few things:

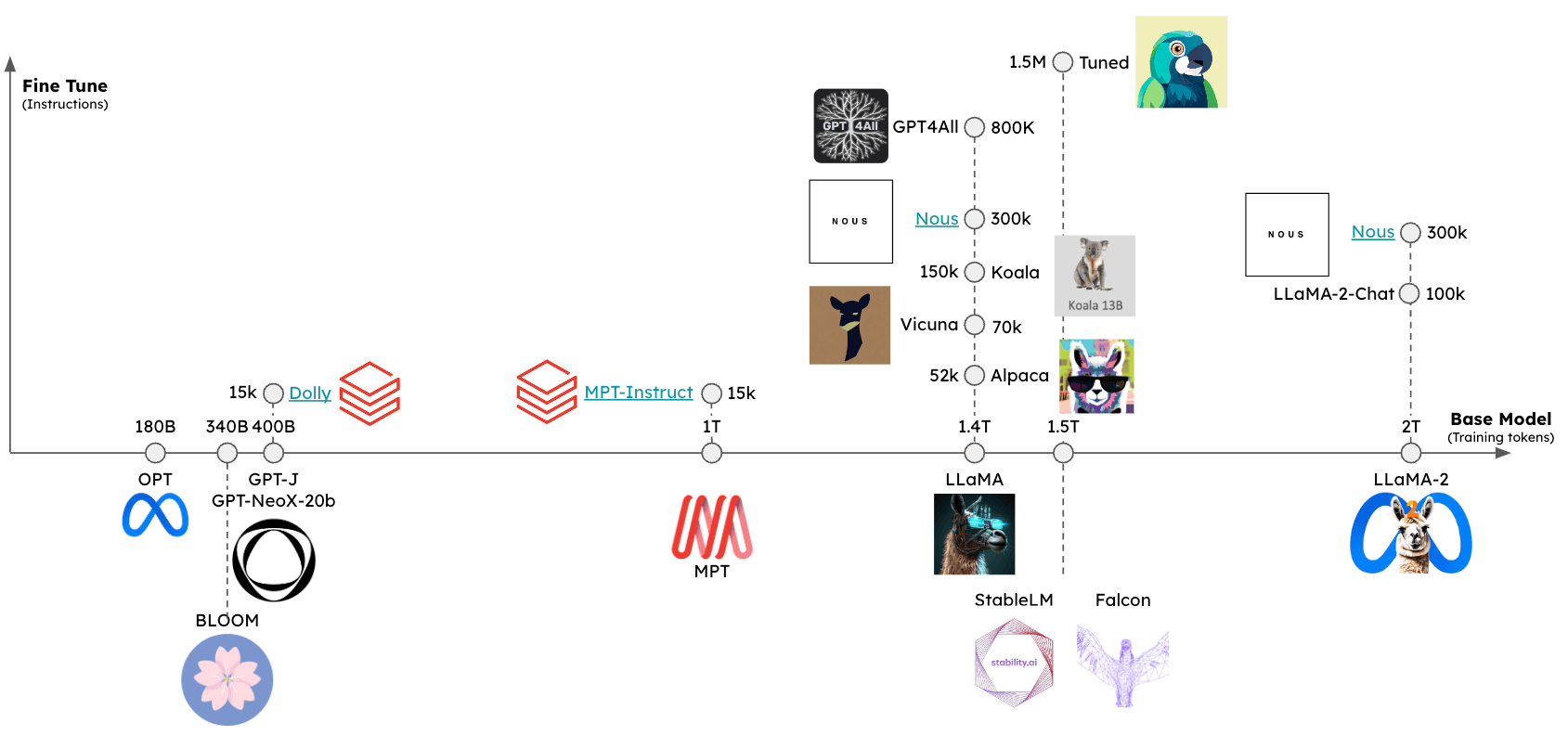
1. **Open-source LLM**: An open-source LLM that can be freely modified and shared
2. **Inference**: Ability to run this LLM on your device w/ acceptable latency

### Open-source LLMs

Users can now gain access to a rapidly growing set of [open-source LLMs](https://cameronrwolfe.substack.com/p/the-history-of-open-source-llms-better).

These LLMs can be assessed across at least two dimensions (see figure):

1. **Base model**: What is the base-model and how was it trained?
2. **Fine-tuning approach**: Was the base-model fine-tuned and, if so, what [set of instructions](https://cameronrwolfe.substack.com/p/beyond-llama-the-power-of-open-llms" \l "%C2%A7alpaca-an-instruction-following-llama-model) was used?



The relative performance of these models can be assessed using several leaderboards, including:

1. [LmSys](https://chat.lmsys.org/?arena)
2. [GPT4All](https://gpt4all.io/index.html)
3. [HuggingFace](https://huggingface.co/spaces/lmsys/chatbot-arena-leaderboard)

### Inference

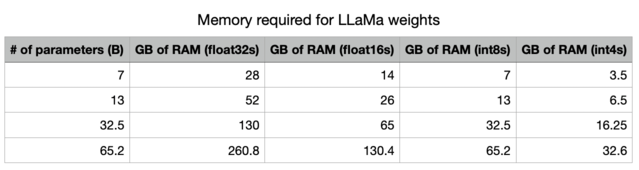
A few frameworks for this have emerged to support inference of open-source LLMs on various devices:

1. [llama.cpp](https://github.com/ggerganov/llama.cpp): C++ implementation of llama inference code with [weight optimization / quantization](https://finbarr.ca/how-is-llama-cpp-possible/)
2. [gpt4all](https://docs.gpt4all.io/index.html): Optimized C backend for inference
3. [Ollama](https://ollama.ai/): Bundles model weights and environment into an app that runs on device and serves the LLM
4. [llamafile](https://github.com/Mozilla-Ocho/llamafile): Bundles model weights and everything needed to run the model in a single file, allowing you to run the LLM locally from this file without any additional installation steps

In general, these frameworks will do a few things:

1. **Quantization**: Reduce the memory footprint of the raw model weights
2. **Efficient implementation for inference**: Support inference on consumer hardware (e.g., CPU or laptop GPU)

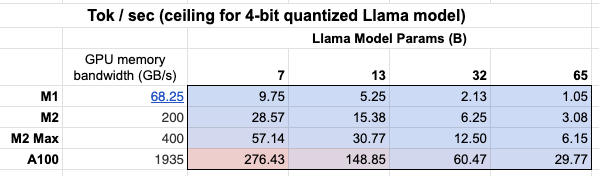
In particular, see [this excellent post](https://finbarr.ca/how-is-llama-cpp-possible/) on the importance of quantization.



With less precision, we radically decrease the memory needed to store the LLM in memory.

In addition, we can see the importance of GPU memory bandwidth [sheet](https://docs.google.com/spreadsheets/d/1OehfHHNSn66BP2h3Bxp2NJTVX97icU0GmCXF6pK23H8/edit" \l "gid=0)!

A Mac M2 Max is 5-6x faster than a M1 for inference due to the larger GPU memory bandwidth.



### Formatting prompts

Some providers have [chat model](https://github.com/langchain-ai/langchain/blob/5ecbb5f2772a216c1d6604793fa50348c85d1c52/docs/docs/how_to/docs/concepts/chat_models) wrappers that takes care of formatting your input prompt for the specific local model you're using. However, if you are prompting local models with a [text-in/text-out LLM](https://github.com/langchain-ai/langchain/blob/5ecbb5f2772a216c1d6604793fa50348c85d1c52/docs/docs/how_to/docs/concepts/text_llms) wrapper, you may need to use a prompt tailored for your specific model.

This can [require the inclusion of special tokens](https://huggingface.co/blog/llama2" \l "how-to-prompt-llama-2). [Here's an example for LLaMA 2](https://smith.langchain.com/hub/rlm/rag-prompt-llama).

## Quickstart

[Ollama](https://ollama.ai/) is one way to easily run inference on macOS.

The instructions [here](https://github.com/jmorganca/ollama?tab=readme-ov-file" \l "ollama) provide details, which we summarize:

* [Download and run](https://ollama.ai/download) the app
* From command line, fetch a model from this [list of options](https://github.com/jmorganca/ollama): e.g., ollama pull llama3.1:8b
* When the app is running, all models are automatically served on localhost:11434

**%pip** install -qU langchain\_ollama

**from** langchain\_ollama **import** OllamaLLM

llm **=** OllamaLLM(model**=**"llama3.1:8b")

llm**.**invoke("The first man on the moon was ...")

'...Neil Armstrong!\n\nOn July 20, 1969, Neil Armstrong became the first person to set foot on the lunar surface, famously declaring "That\'s one small step for man, one giant leap for mankind" as he stepped off the lunar module Eagle onto the Moon\'s surface.\n\nWould you like to know more about the Apollo 11 mission or Neil Armstrong\'s achievements?'

Stream tokens as they are being generated:

**for** chunk **in** llm**.**stream("The first man on the moon was ..."):

print(chunk, end**=**"|", flush**=True**)

...|

Neil| Armstrong|,| an| American| astronaut|.| He| stepped| out| of| the| lunar| module| Eagle| and| onto| the| surface| of| the| Moon| on| July| |20|,| |196|9|,| famously| declaring|:| "|That|'s| one| small| step| for| man|,| one| giant| leap| for| mankind|."||

Ollama also includes a chat model wrapper that handles formatting conversation turns:

**from** langchain\_ollama **import** ChatOllama

chat\_model **=** ChatOllama(model**=**"llama3.1:8b")

chat\_model**.**invoke("Who was the first man on the moon?")

AIMessage(content='The answer is a historic one!\n\nThe first man to walk on the Moon was Neil Armstrong, an American astronaut and commander of the Apollo 11 mission. On July 20, 1969, Armstrong stepped out of the lunar module Eagle onto the surface of the Moon, famously declaring:\n\n"That\'s one small step for man, one giant leap for mankind."\n\nArmstrong was followed by fellow astronaut Edwin "Buzz" Aldrin, who also walked on the Moon during the mission. Michael Collins remained in orbit around the Moon in the command module Columbia.\n\nNeil Armstrong passed away on August 25, 2012, but his legacy as a pioneering astronaut and engineer continues to inspire people around the world!', response\_metadata={'model': 'llama3.1:8b', 'created\_at': '2024-08-01T00:38:29.176717Z', 'message': {'role': 'assistant', 'content': ''}, 'done\_reason': 'stop', 'done': True, 'total\_duration': 10681861417, 'load\_duration': 34270292, 'prompt\_eval\_count': 19, 'prompt\_eval\_duration': 6209448000, 'eval\_count': 141, 'eval\_duration': 4432022000}, id='run-7bed57c5-7f54-4092-912c-ae49073dcd48-0', usage\_metadata={'input\_tokens': 19, 'output\_tokens': 141, 'total\_tokens': 160})

## Environment

Inference speed is a challenge when running models locally (see above).

To minimize latency, it is desirable to run models locally on GPU, which ships with many consumer laptops [e.g., Apple devices](https://www.apple.com/newsroom/2022/06/apple-unveils-m2-with-breakthrough-performance-and-capabilities/).

And even with GPU, the available GPU memory bandwidth (as noted above) is important.

### Running Apple silicon GPU

Ollama and [llamafile](https://github.com/Mozilla-Ocho/llamafile?tab=readme-ov-file" \l "gpu-support) will automatically utilize the GPU on Apple devices.

Other frameworks require the user to set up the environment to utilize the Apple GPU.

For example, llama.cpp python bindings can be configured to use the GPU via [Metal](https://developer.apple.com/metal/).

Metal is a graphics and compute API created by Apple providing near-direct access to the GPU.

See the [llama.cpp](https://github.com/langchain-ai/langchain/blob/5ecbb5f2772a216c1d6604793fa50348c85d1c52/docs/docs/how_to/docs/integrations/llms/llamacpp) setup [here](https://github.com/abetlen/llama-cpp-python/blob/main/docs/install/macos.md) to enable this.

In particular, ensure that conda is using the correct virtual environment that you created (miniforge3).

E.g., for me:

conda activate /Users/rlm/miniforge3/envs/llama

With the above confirmed, then:

CMAKE\_ARGS="-DLLAMA\_METAL=on" FORCE\_CMAKE=1 pip install -U llama-cpp-python --no-cache-dir

## LLMs

There are various ways to gain access to quantized model weights.

1. [HuggingFace](https://huggingface.co/TheBloke) - Many quantized model are available for download and can be run with framework such as [llama.cpp](https://github.com/ggerganov/llama.cpp). You can also download models in [llamafile format](https://huggingface.co/models?other=llamafile) from HuggingFace.
2. [gpt4all](https://gpt4all.io/index.html) - The model explorer offers a leaderboard of metrics and associated quantized models available for download
3. [Ollama](https://github.com/jmorganca/ollama) - Several models can be accessed directly via pull

### Ollama

With [Ollama](https://github.com/jmorganca/ollama), fetch a model via ollama pull <model family>:<tag>:

* E.g., for Llama 2 7b: ollama pull llama2 will download the most basic version of the model (e.g., smallest # parameters and 4 bit quantization)
* We can also specify a particular version from the [model list](https://github.com/jmorganca/ollama?tab=readme-ov-file" \l "model-library), e.g., ollama pull llama2:13b
* See the full set of parameters on the [API reference page](https://python.langchain.com/api_reference/community/llms/langchain_community.llms.ollama.Ollama.html)

llm **=** OllamaLLM(model**=**"llama2:13b")

llm**.**invoke("The first man on the moon was ... think step by step")

' Sure! Here\'s the answer, broken down step by step:\n\nThe first man on the moon was... Neil Armstrong.\n\nHere\'s how I arrived at that answer:\n\n1. The first manned mission to land on the moon was Apollo 11.\n2. The mission included three astronauts: Neil Armstrong, Edwin "Buzz" Aldrin, and Michael Collins.\n3. Neil Armstrong was the mission commander and the first person to set foot on the moon.\n4. On July 20, 1969, Armstrong stepped out of the lunar module Eagle and onto the moon\'s surface, famously declaring "That\'s one small step for man, one giant leap for mankind."\n\nSo, the first man on the moon was Neil Armstrong!'

### Llama.cpp

Llama.cpp is compatible with a [broad set of models](https://github.com/ggerganov/llama.cpp).

For example, below we run inference on llama2-13b with 4 bit quantization downloaded from [HuggingFace](https://huggingface.co/TheBloke/Llama-2-13B-GGML/tree/main).

As noted above, see the [API reference](https://python.langchain.com/api_reference/langchain/llms/langchain.llms.llamacpp.LlamaCpp.html?highlight=llamacpp" \l "langchain.llms.llamacpp.LlamaCpp) for the full set of parameters.

From the [llama.cpp API reference docs](https://python.langchain.com/api_reference/community/llms/langchain_community.llms.llamacpp.LlamaCpp.html), a few are worth commenting on:

n\_gpu\_layers: number of layers to be loaded into GPU memory

* Value: 1
* Meaning: Only one layer of the model will be loaded into GPU memory (1 is often sufficient).

n\_batch: number of tokens the model should process in parallel

* Value: n\_batch
* Meaning: It's recommended to choose a value between 1 and n\_ctx (which in this case is set to 2048)

n\_ctx: Token context window

* Value: 2048
* Meaning: The model will consider a window of 2048 tokens at a time

f16\_kv: whether the model should use half-precision for the key/value cache

* Value: True
* Meaning: The model will use half-precision, which can be more memory efficient; Metal only supports True.

**%env** CMAKE\_ARGS="-DLLAMA\_METAL=on"

**%env** FORCE\_CMAKE=1

**%pip** install --upgrade --quiet llama-cpp-python --no-cache-dir

**from** langchain\_community.llms **import** LlamaCpp

**from** langchain\_core.callbacks **import** CallbackManager, StreamingStdOutCallbackHandler

llm **=** LlamaCpp(

model\_path**=**"/Users/rlm/Desktop/Code/llama.cpp/models/openorca-platypus2-13b.gguf.q4\_0.bin",

n\_gpu\_layers**=**1,

n\_batch**=**512,

n\_ctx**=**2048,

f16\_kv**=True**,

callback\_manager**=**CallbackManager([StreamingStdOutCallbackHandler()]),

verbose**=True**,

)

The console log will show the below to indicate Metal was enabled properly from steps above:

ggml\_metal\_init: allocating

ggml\_metal\_init: using MPS

llm**.**invoke("The first man on the moon was ... Let's think step by step")

Llama.generate: prefix-match hit

and use logical reasoning to figure out who the first man on the moon was.

Here are some clues:

1. The first man on the moon was an American.

2. He was part of the Apollo 11 mission.

3. He stepped out of the lunar module and became the first person to set foot on the moon's surface.

4. His last name is Armstrong.

Now, let's use our reasoning skills to figure out who the first man on the moon was. Based on clue #1, we know that the first man on the moon was an American. Clue #2 tells us that he was part of the Apollo 11 mission. Clue #3 reveals that he was the first person to set foot on the moon's surface. And finally, clue #4 gives us his last name: Armstrong.

Therefore, the first man on the moon was Neil Armstrong!

llama\_print\_timings: load time = 9623.21 ms

llama\_print\_timings: sample time = 143.77 ms / 203 runs ( 0.71 ms per token, 1412.01 tokens per second)

llama\_print\_timings: prompt eval time = 485.94 ms / 7 tokens ( 69.42 ms per token, 14.40 tokens per second)

llama\_print\_timings: eval time = 6385.16 ms / 202 runs ( 31.61 ms per token, 31.64 tokens per second)

llama\_print\_timings: total time = 7279.28 ms

" and use logical reasoning to figure out who the first man on the moon was.\n\nHere are some clues:\n\n1. The first man on the moon was an American.\n2. He was part of the Apollo 11 mission.\n3. He stepped out of the lunar module and became the first person to set foot on the moon's surface.\n4. His last name is Armstrong.\n\nNow, let's use our reasoning skills to figure out who the first man on the moon was. Based on clue #1, we know that the first man on the moon was an American. Clue #2 tells us that he was part of the Apollo 11 mission. Clue #3 reveals that he was the first person to set foot on the moon's surface. And finally, clue #4 gives us his last name: Armstrong.\nTherefore, the first man on the moon was Neil Armstrong!"

### GPT4All

We can use model weights downloaded from [GPT4All](https://github.com/langchain-ai/langchain/blob/5ecbb5f2772a216c1d6604793fa50348c85d1c52/docs/docs/how_to/docs/integrations/llms/gpt4all) model explorer.

Similar to what is shown above, we can run inference and use [the API reference](https://python.langchain.com/api_reference/community/llms/langchain_community.llms.gpt4all.GPT4All.html) to set parameters of interest.

**%pip** install gpt4all

**from** langchain\_community.llms **import** GPT4All

llm **=** GPT4All(

model**=**"/Users/rlm/Desktop/Code/gpt4all/models/nous-hermes-13b.ggmlv3.q4\_0.bin"

)

llm**.**invoke("The first man on the moon was ... Let's think step by step")

".\n1) The United States decides to send a manned mission to the moon.2) They choose their best astronauts and train them for this specific mission.3) They build a spacecraft that can take humans to the moon, called the Lunar Module (LM).4) They also create a larger spacecraft, called the Saturn V rocket, which will launch both the LM and the Command Service Module (CSM), which will carry the astronauts into orbit.5) The mission is planned down to the smallest detail: from the trajectory of the rockets to the exact movements of the astronauts during their moon landing.6) On July 16, 1969, the Saturn V rocket launches from Kennedy Space Center in Florida, carrying the Apollo 11 mission crew into space.7) After one and a half orbits around the Earth, the LM separates from the CSM and begins its descent to the moon's surface.8) On July 20, 1969, at 2:56 pm EDT (GMT-4), Neil Armstrong becomes the first man on the moon. He speaks these"

### llamafile

One of the simplest ways to run an LLM locally is using a [llamafile](https://github.com/Mozilla-Ocho/llamafile). All you need to do is:

1. Download a llamafile from [HuggingFace](https://huggingface.co/models?other=llamafile)
2. Make the file executable
3. Run the file

llamafiles bundle model weights and a [specially-compiled](https://github.com/Mozilla-Ocho/llamafile?tab=readme-ov-file" \l "technical-details) version of [llama.cpp](https://github.com/ggerganov/llama.cpp) into a single file that can run on most computers without any additional dependencies. They also come with an embedded inference server that provides an [API](https://github.com/Mozilla-Ocho/llamafile/blob/main/llama.cpp/server/README.md" \l "api-endpoints) for interacting with your model.

Here's a simple bash script that shows all 3 setup steps:

*# Download a llamafile from HuggingFace*

wget https://huggingface.co/jartine/TinyLlama-1.1B-Chat-v1.0-GGUF/resolve/main/TinyLlama-1.1B-Chat-v1.0.Q5\_K\_M.llamafile

*# Make the file executable. On Windows, instead just rename the file to end in ".exe".*

chmod +x TinyLlama-1.1B-Chat-v1.0.Q5\_K\_M.llamafile

*# Start the model server. Listens at http://localhost:8080 by default.*

./TinyLlama-1.1B-Chat-v1.0.Q5\_K\_M.llamafile --server --nobrowser

After you run the above setup steps, you can use LangChain to interact with your model:

**from** langchain\_community.llms.llamafile **import** Llamafile

llm **=** Llamafile()

llm**.**invoke("The first man on the moon was ... Let's think step by step.")

"\nFirstly, let's imagine the scene where Neil Armstrong stepped onto the moon. This happened in 1969. The first man on the moon was Neil Armstrong. We already know that.\n2nd, let's take a step back. Neil Armstrong didn't have any special powers. He had to land his spacecraft safely on the moon without injuring anyone or causing any damage. If he failed to do this, he would have been killed along with all those people who were on board the spacecraft.\n3rd, let's imagine that Neil Armstrong successfully landed his spacecraft on the moon and made it back to Earth safely. The next step was for him to be hailed as a hero by his people back home. It took years before Neil Armstrong became an American hero.\n4th, let's take another step back. Let's imagine that Neil Armstrong wasn't hailed as a hero, and instead, he was just forgotten. This happened in the 1970s. Neil Armstrong wasn't recognized for his remarkable achievement on the moon until after he died.\n5th, let's take another step back. Let's imagine that Neil Armstrong didn't die in the 1970s and instead, lived to be a hundred years old. This happened in 2036. In the year 2036, Neil Armstrong would have been a centenarian.\nNow, let's think about the present. Neil Armstrong is still alive. He turned 95 years old on July 20th, 2018. If he were to die now, his achievement of becoming the first human being to set foot on the moon would remain an unforgettable moment in history.\nI hope this helps you understand the significance and importance of Neil Armstrong's achievement on the moon!"

## Prompts

Some LLMs will benefit from specific prompts.

For example, LLaMA will use [special tokens](https://twitter.com/RLanceMartin/status/1681879318493003776?s=20).

We can use ConditionalPromptSelector to set prompt based on the model type.

*# Set our LLM*

llm **=** LlamaCpp(

model\_path**=**"/Users/rlm/Desktop/Code/llama.cpp/models/openorca-platypus2-13b.gguf.q4\_0.bin",

n\_gpu\_layers**=**1,

n\_batch**=**512,

n\_ctx**=**2048,

f16\_kv**=True**,

callback\_manager**=**CallbackManager([StreamingStdOutCallbackHandler()]),

verbose**=True**,

)

Set the associated prompt based upon the model version.

**from** langchain.chains.prompt\_selector **import** ConditionalPromptSelector

**from** langchain\_core.prompts **import** PromptTemplate

DEFAULT\_LLAMA\_SEARCH\_PROMPT **=** PromptTemplate(

input\_variables**=**["question"],

template**=**"""<<SYS>> \n You are an assistant tasked with improving Google search \

results. \n <</SYS>> \n\n [INST] Generate THREE Google search queries that \

are similar to this question. The output should be a numbered list of questions \

and each should have a question mark at the end: \n\n {question} [/INST]""",

)

DEFAULT\_SEARCH\_PROMPT **=** PromptTemplate(

input\_variables**=**["question"],

template**=**"""You are an assistant tasked with improving Google search \

results. Generate THREE Google search queries that are similar to \

this question. The output should be a numbered list of questions and each \

should have a question mark at the end: {question}""",

)

QUESTION\_PROMPT\_SELECTOR **=** ConditionalPromptSelector(

default\_prompt**=**DEFAULT\_SEARCH\_PROMPT,

conditionals**=**[(**lambda** llm: isinstance(llm, LlamaCpp), DEFAULT\_LLAMA\_SEARCH\_PROMPT)],

)

prompt **=** QUESTION\_PROMPT\_SELECTOR**.**get\_prompt(llm)

prompt

PromptTemplate(input\_variables=['question'], output\_parser=None, partial\_variables={}, template='<<SYS>> \n You are an assistant tasked with improving Google search results. \n <</SYS>> \n\n [INST] Generate THREE Google search queries that are similar to this question. The output should be a numbered list of questions and each should have a question mark at the end: \n\n {question} [/INST]', template\_format='f-string', validate\_template=True)

*# Chain*

chain **=** prompt **|** llm

question **=** "What NFL team won the Super Bowl in the year that Justin Bieber was born?"

chain**.**invoke({"question": question})

Sure! Here are three similar search queries with a question mark at the end:

1. Which NBA team did LeBron James lead to a championship in the year he was drafted?

2. Who won the Grammy Awards for Best New Artist and Best Female Pop Vocal Performance in the same year that Lady Gaga was born?

3. What MLB team did Babe Ruth play for when he hit 60 home runs in a single season?

llama\_print\_timings: load time = 14943.19 ms

llama\_print\_timings: sample time = 72.93 ms / 101 runs ( 0.72 ms per token, 1384.87 tokens per second)

llama\_print\_timings: prompt eval time = 14942.95 ms / 93 tokens ( 160.68 ms per token, 6.22 tokens per second)

llama\_print\_timings: eval time = 3430.85 ms / 100 runs ( 34.31 ms per token, 29.15 tokens per second)

llama\_print\_timings: total time = 18578.26 ms

' Sure! Here are three similar search queries with a question mark at the end:\n\n1. Which NBA team did LeBron James lead to a championship in the year he was drafted?\n2. Who won the Grammy Awards for Best New Artist and Best Female Pop Vocal Performance in the same year that Lady Gaga was born?\n3. What MLB team did Babe Ruth play for when he hit 60 home runs in a single season?'

We also can use the LangChain Prompt Hub to fetch and / or store prompts that are model specific.

This will work with your [LangSmith API key](https://docs.smith.langchain.com/).

For example, [here](https://smith.langchain.com/hub/rlm/rag-prompt-llama) is a prompt for RAG with LLaMA-specific tokens.

## Use cases

Given an llm created from one of the models above, you can use it for [many use cases](https://github.com/langchain-ai/langchain/blob/5ecbb5f2772a216c1d6604793fa50348c85d1c52/docs/docs/how_to/docs/how_to" \l "use-cases).

For example, you can implement a [RAG application](https://github.com/langchain-ai/langchain/blob/5ecbb5f2772a216c1d6604793fa50348c85d1c52/docs/docs/how_to/docs/tutorials/rag) using the chat models demonstrated here.

In general, use cases for local LLMs can be driven by at least two factors:

* **Privacy**: private data (e.g., journals, etc) that a user does not want to share
* **Cost**: text preprocessing (extraction/tagging), summarization, and agent simulations are token-use-intensive tasks

In addition, [here](https://blog.langchain.dev/using-langsmith-to-support-fine-tuning-of-open-source-llms/) is an overview on fine-tuning, which can utilize open-source LLMs.

**📄 Páginas e guias escritos**

**Tutoriais com código e explicações**

* **Build a Local AI Agent with Python, Ollama, LangChain and SingleStore** — tutorial técnico com Docker, embeddings, LangChain e SingleStore como vetorial DB. [YouTube+4YouTube+4YouTube+4](https://www.youtube.com/watch?v=Y6sn-yGhRmI&utm_source=chatgpt.com)
* **Building a Local RAG Agent with LLaMA3 and LangChain** — tutorial do Medium mostrando como integrar LLaMA 3, RAG, autoproteção para evitar alucinações e rotas adaptativas. [Medium](https://medium.com/%40pankaj_pandey/building-a-local-rag-agent-with-llama3-and-langchain-6f041655eb83?utm_source=chatgpt.com)
* **From Zero to Hero: Building Your First LangChain Agent with RAG** — guia para iniciantes com integração de UI, ferramentas (calculator, search) e RAG. [DEV Community](https://dev.to/vaib/from-zero-to-hero-building-your-first-langchain-agent-with-rag-1c8h?utm_source=chatgpt.com)
* **How to Build a Local AI Assistant with Llama 3.2, RAG and Search using Ollama and MCP** — workflow por n8n com Llama 3.2 e servidor MCP passo a passo. [YouTube+15YouTube+15YouTube+15](https://www.youtube.com/watch?v=7qrruzYC0b4&utm_source=chatgpt.com)

**Recursos gerais e frameworks open‑source**

* **Awesome AI Agents: Tools, Resources, and Projects** — repositório no GitHub com curadoria completa de ferramentas, frameworks, modelos, agentes com código aberto. Excelente para descobrir frameworks como Cursor, Flowise, LangChain, Agent-UniRAG, etc. [arXiv](https://arxiv.org/abs/2408.05933?utm_source=chatgpt.com)
* **Build an Agent – LangChain docs** — tutorial oficial da LangChain que mostra construção de agente com integração a buscador e lógica de ferramentas. [python.langchain.com](https://python.langchain.com/docs/tutorials/agents/?utm_source=chatgpt.com)
* **Run models locally – LangChain docs** — explicação sobre frameworks como llama.cpp, Ollama, GPT4All, quantização de modelos para rodar localmente. [python.langchain.com](https://python.langchain.com/docs/how_to/local_llms/?utm_source=chatgpt.com)
* **n8n AI Agents tutorials – Medium** — apresenta seis workflows reais com agentes: RAG, análise de dados, newsletters automatizadas, resumos de PDF e criação de MCP Server dentro n8n. [Medium](https://medium.com/data-science-in-your-pocket/n8n-ai-agents-tutorials-b83c15da5018?utm_source=chatgpt.com)

**📚 Pesquisas acadêmicas relevantes**

* **Agent‑UniRAG: trainable open‑source agent framework** — framework unificado para RAG com LLMs menores como Llama‑3‑8B, interpretável e open‑source. [arXiv](https://arxiv.org/abs/2505.22571?utm_source=chatgpt.com)
* **LAMBDA: Large Model Based Data Agent** — sistema open‑source para agentes data‑driven que geram e depuram código automaticamente. Útil pro seu propósito de fazer um agente que programe. [arXiv](https://arxiv.org/abs/2407.17535?utm_source=chatgpt.com)
* **Agents: an open‑source framework for autonomous language agents** — biblioteca para criar agentes com memória, ferramentas, multi‑agente, sem necessidade de muito código. [arXiv](https://arxiv.org/abs/2309.07870?utm_source=chatgpt.com)

**🧭 Como organizar para o seu projeto**

Sugiro agrupar os links nos seguintes tópicos e incluir anotações breves:

1. **Configuração básica local** (LLM local, quantização, frameworks como llama.cpp, Ollama, GPT4All)
2. **Código com Python + LangChain + RAG** (tutoriales step-by-step)
3. **No-code / visual workflows com n8n + MCP** (auto geração de workflows com prompts naturais e uso de embeddings)
4. **Frameworks open‑source e projetos adaptáveis** (repositórios como Awesome AI Agents, Agent‑UniRAG, LAMBDA, Agents library)
5. **Casos de uso relevantes** (programação automatizada de código via agente, agentes de assistência com memória, auto‑aprendizado, multi‑agente)