LangChain Docs

ChatOllama

ChatOllama

Copy page

Get started using Ollama **chat models** in LangChain.

Ollama allows you to run open-source Large Language Models (LLMs), such as `gpt-oss`, locally.

Ollama bundles model weights, configuration, and data into a single package, defined by a Modelfile. It optimizes setup and configuration details, including GPU usage.

For a complete list of supported models and model variants, see the **Ollama model library**.

> **API Reference**
>
> For detailed documentation of all features and configuration options, head to the `ChatOllama` API reference.

Overview

Integration details

Class	Package	Serializable	JS support	Downl
`ChatOllama`	`langchain-ollama`	❌	✅	1.7M/month

Model features

LangChain Docs

ChatOllama

Setup

First, follow **these instructions** to set up and run a local Ollama instance:

- **Download** and install Ollama onto the available supported platforms (including Windows Subsystem for Linux aka WSL, macOS, and Linux)
  - macOS users can install via Homebrew with `brew install ollama` and start with `brew services start ollama`
- Fetch available LLM model via `ollama pull <name-of-model>`
  - View a list of available models via the **model library**
  - e.g., `ollama pull gpt-oss:20b`
- This will download the default tagged version of the model. Typically, the default points to the latest, smallest sized-parameter model.

> On Mac, the models will be download to `~/.ollama/models` On Linux (or WSL), the models will be stored at `/usr/share/ollama/.ollama/models`

- Specify the exact version of the model of interest as such `ollama pull gpt-oss:20b` (View the **various tags for the `Vicuna`** model in this instance)
- To view all pulled models, use `ollama list`
- To chat directly with a model from the command line, use `ollama run <name-of-model>`
- View the **Ollama documentation** for more commands. You can run `ollama help` in the terminal to see available commands.

To enable automated tracing of your model calls, set your **LangSmith** API key:

LangChain Docs

ChatOllama

LangChain Docs

ChatOllama

LangChain Docs

ChatOllama

LangChain Docs

ChatOllama

LangChain Docs

ChatOllama

LangChain Docs

ChatOllama

```
os.environ["LANGSMITH_TRACING"] = "true"
os.environ["LANGSMITH_API_KEY"] = getpass.getpass("Enter your LangSmith API
```

```
pip install -qU langchain-ollama
```

⚠ Make sure you're using the latest Ollama version!

Update by running:

```
pip install -U ollama
```

Instantiation

Now we can instantiate our model object and generate chat completions:

```
from langchain_ollama import ChatOllama

llm = ChatOllama(
    model="llama3.1",
    temperature=0,
    # other params...
)
```

Invocation

```
    ),
    ("human", "I love programming."),
]
ai_msg = llm.invoke(messages)
ai_msg
```

```
AIMessage(content='The translation of "I love programming" in Fre
```

```
print(ai_msg.content)
```

```
The translation of "I love programming" in French is:

"J'adore le programmation."
```

Tool calling

Ollama tool calling uses the OpenAI compatible web server specification, and can be used with the default `BaseChatModel.bind_tools()` methods as described here.

Make sure to select an ollama model that supports tool calling.

We can use tool calling with an LLM that has been fine-tuned for tool use such as `gpt-oss`:

```
ollama pull gpt-oss:20b
```

Details on creating custom tools are available in this guide. Below, we demonstrate how to create a tool using the `@tool` decorator on a normal python function.

LangChain Docs

ChatOllama

```python
from langchain_ollama import ChatOllama


@tool
def validate_user(user_id: int, addresses: List[str]) -> bool:
    """Validate user using historical addresses.

    Args:
        user_id (int): the user ID.
        addresses (List[str]): Previous addresses as a list of strings.
    """
    return True


llm = ChatOllama(
    model="gpt-oss:20b",
    validate_model_on_init=True,
    temperature=0,
).bind_tools([validate_user])

result = llm.invoke(
    "Could you validate user 123? They previously lived at "
    "123 Fake St in Boston MA and 234 Pretend Boulevard in "
    "Houston TX."
)

if isinstance(result, AIMessage) and result.tool_calls:
    print(result.tool_calls)
```

```
[{'name': 'validate_user', 'args': {'addresses': ['123 Fake St, E
```

Multi-modal

ıa has limited support for multi-modal LLMs, such as gemma3

LangChain Docs

ChatOllama

LangChain Docs

ChatOllama

```
from PIL import Image


def convert_to_base64(pil_image):
    """
    Convert PIL images to Base64 encoded strings

    :param pil_image: PIL image
    :return: Re-sized Base64 string
    """

    buffered = BytesIO()
    pil_image.save(buffered, format="JPEG")  # You can change the format if r
    img_str = base64.b64encode(buffered.getvalue()).decode("utf-8")
    return img_str


def plt_img_base64(img_base64):
    """
    Disply base64 encoded string as image

    :param img_base64:  Base64 string
    """
    # Create an HTML img tag with the base64 string as the source
    image_html = f'<img src="{img_base64}" />'
    # Display the image by rendering the HTML
    display(HTML(image_html))


file_path = "../../../static/img/ollama_example_img.jpg"
pil_image = Image.open(file_path)

image_b64 = convert_to_base64(pil_image)
plt_img_base64(image_b64)
```

```
ng src="
```

LangChain Docs

ChatOllama

```
llm = ChatOllama(model="bakllava", temperature=0)


def prompt_func(data):
    text = data["text"]
    image = data["image"]

    image_part = {
        "type": "image_url",
        "image_url": f"{image}",
    }

    content_parts = []

    text_part = {"type": "text", "text": text}

    content_parts.append(image_part)
    content_parts.append(text_part)

    return [HumanMessage(content=content_parts)]


from langchain_core.output_parsers import StrOutputParser

chain = prompt_func | llm | StrOutputParser()

query_chain = chain.invoke(
    {"text": "What is the Dollar-based gross retention rate?", "image": image
)

print(query_chain)
```

90%

Reasoning models and custom message roles

content set to `"thinking"`. Because `"control"` is a non-standard message role, we can use a **ChatMessage** object to implement it:

```python
from langchain.messages import HumanMessage
from langchain_core.messages import ChatMessage
from langchain_ollama import ChatOllama

llm = ChatOllama(model="granite3.2:8b")

messages = [
    ChatMessage(role="control", content="thinking"),
    HumanMessage("What is 3^3?"),
]

response = llm.invoke(messages)
print(response.content)
```

```
Here is my thought process:
The user is asking for the value of 3 raised to the power of 3, which is a ba

Here is my response:

3^3 (read as "3 to the power of 3") equals 27.

This calculation is performed by multiplying 3 by itself three times: 3*3*3 =
```

Note that the model exposes its thought process in addition to its final response.

API reference

For detailed documentation of all `ChatOllama` features and configurations head to the

LangChain Docs

ChatOllama

Edit this page on GitHub or file an issue.

Connect these docs to Claude, VSCode, and more via MCP for real-time answers.

Was this page helpful? Yes No

LangChain Docs

Resources

Forum

Changelog

LangChain Academy

Trust Center

Company

About

Careers

Blog

Powered by mintlify