Rubber Ball Playground

Deploying and Interacting with Large Language Models using Ollama

Roy Ratcliffe[1,*]

Abstract

This article explores the deployment and interaction of large language models (LLMs) using Ollama, an open-source platform designed for local execution. It demonstrates how to containerise Ollama with Docker, configure it for GPU or CPU usage, and interact with its RESTful API using SWI-Prolog. The discussion includes practical examples of both streamed and non-streamed JSON communication, as well as Prologue predicates for chat interactions and debugging techniques. The approach emphasises the flexibility and simplicity Ollama offers for running and managing LLMs locally, making it a powerful tool for developers seeking privacy, control, and performance in AI applications.

Keywords: Prolog, LLM

Apparently, "ollama" refers to an ancient meso-American rubber ball game player. "Olli" means rubber in the ancient Aztec language. Ollama [1] is an open-source platform for simplifying the deployment of large language models (LLMs) on one's own hardware.

1. Ollama in Docker

Launch Ollama in a Docker container, ideally on a host machine with a GPU. The container does not *require* a GPU-enabled host; it utilises the CPU if it fails to find a GPU, albeit slow to run without the additional cores.

The command line below launches and detaches[2] a Docker container named "ollama" using the `ollama/ollama` image. It gives the container access to all the available GPUs and exposes port 11434 on the host. The "start always" keeps the container running and restarts it automatically on boot up, *and* if it fails in some way, only manually stopping the container prevents the restart.

```
docker run -d --gpus=all -p 11434:11434 --restart always --name ollama ollama/ollama
```

You might see a pull-failure message about no matching manifest for `windows(10.0.19045)/amd64` if running on Windows. Right-click on the Docker icon and switch to Linux containers.

The container does not contain a default model; running `docker exec ollama ollama list` lists the currently running models, initially an empty list. Run DeepSeek-R1 with seven billion parameters (the 7B variant) distilled from its original 671 billion as follows. Adjust according to requirements.

```
docker exec ollama ollama run deepseek-r1:7b
```

*Corresponding author
Email address: roy@ratcliffe.me (Roy Ratcliffe)
[1] See more hackery at GitHub.
[2] runs in the background without an interactive terminal interface

This container exposes a JSON-based RESTful API at port 11434. That makes interacting with Ollama fairly straightforward. JSON delivers compact payloads for faster transmission over networks, is easy to read even for humans, and parsers prove ubiquitous in every language.

The simplest, perhaps the most naive, logic-based programming implementation for interaction with the Ollama chat agent appears in the listing below; the logical approach defines *rules* and *facts* rather than step-by-step instructions. The `ollama_chat1` predicate runs a standard HTTP POST request where the payload is a JSON object derived from a dictionary. This is the first iteration, but not the last.

```
:- ensure_loaded(library(http/http_json)).

:- setting(ollama_model, string, env('OLLAMA_MODEL', "deepseek-r1:7b"), '').

ollama_chat1(Content, Reply, Options) :-
    setting(ollama_model, OllamaModel),
    option(model(Model), Options, OllamaModel),
    http_post('http://localhost:11434/api/chat',
              json(_{stream:false,
                     model:Model,
                     messages:[_{role:user, content:Content}]}),
              Reply, [json_object(dict)]).
```

1.1. What to note?

The dictionary contains "`stream:false`", which instructs the Ollama agent **not** to stream. It streams by default, but the standard POST request over HTTP does not expect a chunked response.

The pair "`messages:ListOfMessages`" is a list of sub-dictionaries that carry the conversation between the user and the AI assistant. The `role` key identifies the source of the content.

The `Options` argument lets the caller override the model, defaulting to the `OLLAMA_MODEL` environment variable and defaulting to DeepSeek R1 as a final fallback.

The "ensure loaded" for the JSON module makes sure that the HTTP library finds the necessary JSON functionality. Atoms and strings become JSON strings. Hence, key-value dictionary pair "`role:user`" translates to JSON `"role": "user"` in the POST's payload.

2. Chunked JSON Streams

How to handle streaming responses from Ollama?

The API delivers chunked JSON by default. A language model's output appears progressively, word by word, or strictly speaking, token by token. Their design targets real-time chat applications where the model "talks" to a user interactively. Its tokens appear in replies as if someone or something were typing. The first token predicts the next, rinse, repeat.

The following refactors the Ollama chat predicate to allow for streamed and non-streamed interactions. This comes after a little bit of tweaking and fiddling.

```
%!  ollama_chat(+Messages:list(dict), -Message:dict, +Options:list) is
%!              nondet.

ollama_chat(Messages, Message, Options) :-
    option(stream(Stream), Options, true),
    setting(ollama_model, OllamaModel),
    option(model(Model), Options, OllamaModel),
```

```
    option(reply(Reply), Options, Reply),
    setting(ollama_url, URL),
    chat(URL, _{stream:Stream,
                model:Model,
                messages:Messages}, Reply, Options),
    Message = Reply.message.

chat(URL, Dict, Reply, Options) :-
    http_open(URL, In, [post(json(Dict)), headers(Headers)|Options]),
    call_cleanup(read_dict(In, Reply, Headers), close(In)).
```

To stream or not to stream? That becomes an option, specify either `stream(true)` or `stream(false)`, defaulting to streaming. This option selects the predicate's determinism. Predicate `ollama_chat/3` becomes non-deterministic *when* streaming, but falls back to deterministic when not.

The last clause of the `ollama_chat/3` predicate pulls out the message from the Reply; it becomes the result of the chat interaction: messages in, one message out. Taking only the Message assumes that the other keys within the reply dictionary have less value. Callers can ignore them most of the time. The predicate unifies with `reply(Reply)` in the Options argument if the caller wants to see the dirty details.

2.1. Newline-delimited JSON

The Ollama API sends content type `application/x-ndjson`. Launch SWI-Prolog with the `-Dsource` command line option and enable HTTP debugging using the `debug(http(open))` query. Prolog will show the request's headers and status.

Tell Prolog about that type of JSON as follows. It does not recognise it by default.

```
:- multifile http_json:json_type/1.

http_json:json_type(application/'x-ndjson').
```

2.2. Reading the chunked JSON stream

The goal: read a JSON-encoded dictionary from an HTTP stream, whether a chunked stream or not. Ask Prolog for a `dict` object. Succeed with a backtracking choice point if the dictionary has *done* = *false* or "there is more", in other words. Cut the choice point if done. There is no need to continue when Ollama completes, even though the predicate has not yet seen the end-of-stream marker. The stream will deliver no more messages.

```
%!  read_dict(+In, -Dict, +Options) is nondet.
%
%   Reads a dictionary from a JSON object, either deterministically for
%   non-chunked stream responses or non-deterministically for chunked
%   JSON responses. Continues with backtracking when the dictionary has
%   not done, i.e. done is false. Cuts before the end-of-stream if done
%   is true.

read_dict(In, Dict, Options) :-
    read_data(In, Dict, [json_object(dict)|Options]),
    (   get_dict(done, Dict, false)
    ->  true
    ;   !
    ).

read_data(In, NotEndOfFileData, Options) :-
```

```
    select_option(end_of_file(EndOfFile), Options, Options_, end_of_file),
    repeat,
    (   http_read_data([input(In)], Data, [end_of_file(EndOfFile)|Options_]),
        Data \== EndOfFile
    ->  NotEndOfFileData = Data
    ;   !,
        fail
    ).
```

2.3. *Transfer encoding bug*

SWI-Prolog includes a transfer-encoding bug. Version 9.3.24 closes the stream before chunking. It's a bug. Jan, the author, recently fixed it. Use a later version or the daily build to avoid the bug.

3. Usage

How to use and abuse the interface? Take some examples.

The following queries run with HTTP debugging enabled. Notice the headers. Debugging enabled.

For streaming:

```
?- ollama_chat([_{role:user, content:"Hello"}], Message, [stream(true)]).
% http_open: Connecting to localhost:11434 ...
%       ok <stream>(000001a4454d2630) ---> <stream>(000001a4454d2740)
% HTTP/1.1 200 OK
% Content-Type: application/x-ndjson
% Date: Sat, 31 May 2025 10:48:49 GMT
% Connection: close
% Transfer-Encoding: chunked
Message = _{content:" Hello", role:"assistant"} ;
Message = _{content:"!", role:"assistant"} ;
Message = _{content:" How", role:"assistant"} ;
Message = _{content:" can", role:"assistant"} ;
Message = _{content:" I", role:"assistant"} ;
Message = _{content:" assist", role:"assistant"} ;
Message = _{content:" you", role:"assistant"} ;
Message = _{content:" today", role:"assistant"} ;
Message = _{content:"?", role:"assistant"} ;
Message = _{content:"", role:"assistant"}.
```

The streaming content type is **not** "application/json" but rather newline-delimited JSON. This is correct. Our addition to the JSON type multifile predicate catches this.

For non-streaming:

```
?- ollama_chat([_{role:user, content:"Hello"}], Message, [stream(false)]).
% http_open: Connecting to localhost:11434 ...
%       ok <stream>(000001a4454d3ea0) ---> <stream>(000001a4454d4e90)
% HTTP/1.1 200 OK
% Content-Type: application/json; charset=utf-8
% Date: Sat, 31 May 2025 10:50:04 GMT
% Content-Length: 347
% Connection: close
Message = _{content:" Sure, I'm here to help! How can I assist you today?", role:"assistant"}.
```

Perfect. Just what the ‘rubber ball player’ ordered.

4. Conclusions

> “Take it easy, take it easy. Don’t let the sound of your own wheels drive you crazy.”—Eagles.

Ollama makes life easier when deploying local large-language models, or even small language models. Ollama also makes running multiple models simultaneously very easy.

References

[1] Ollama Contributors, Ollama: Run large language models locally, https://github.com/ollama/ollama, accessed: 2025-05-31 (2025).