Comprehensive Local LLM and Chatbot Development Project

Chong Chen

September 25, 2024

Abstract

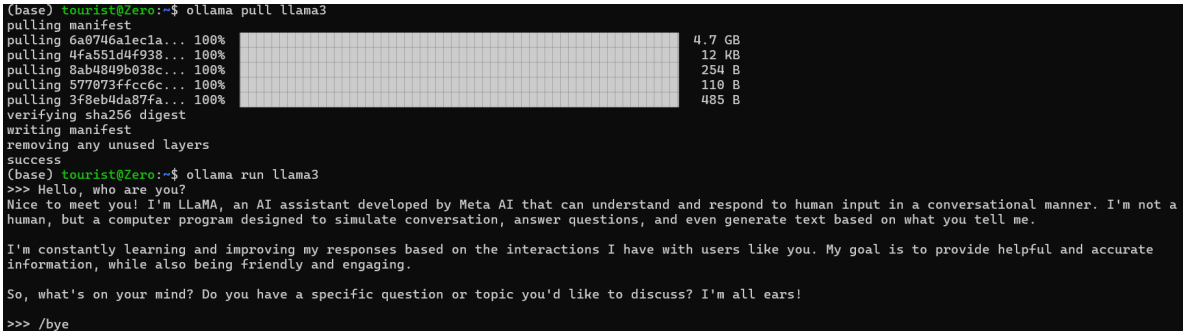
In this assignment, I set up and interacted with models on my local machine, and then developed a domain-specific chat-bot. This project improves my technical skills in model deployment, software development, and natural language processing. All codes and testing results could be seen on https://github.com/chongchen1999/LLM-Chatbot.

1 Local LLM Setup and Interaction

1.1 Set up Llama3

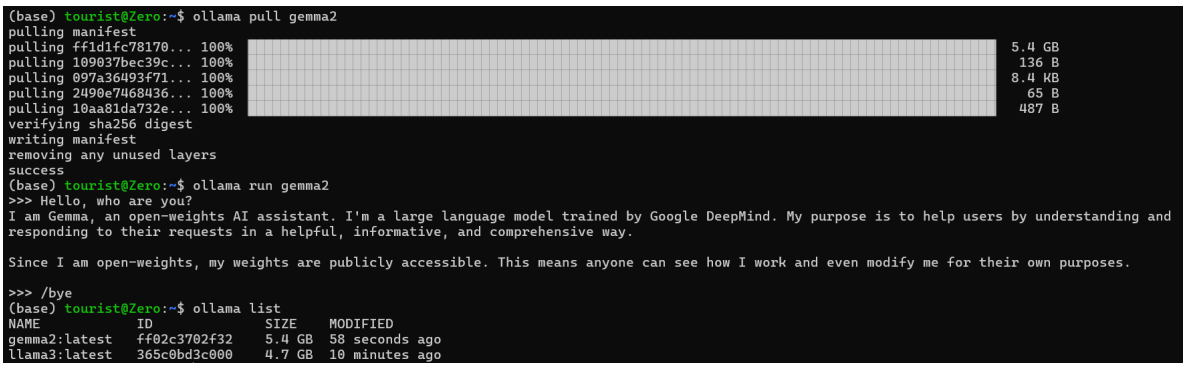
I pull and run llama3 by using Ollama, and then verify the installation by running a simple query using CLI.

```
ollama pull llama3
ollama run llama3
```



1.2 Choose and Set Up a Second LLM

For the second LLM model, I choose Google's Gemma2.



Through the command `ollama list`, all available models on local device will be shown.

1.3 Command-Line Interaction Using Curl

In this section, I developed a Python program that performs experiments by querying different language models through APIs. The program consists of two scripts: `query_sender.py` and `main.py`, each with specific functions to manage the querying process and analyze performance metrics.

The purpose of `query_sender.py` is to define the function `send_query`, which handles API requests and processes the model's responses. This script performs several key tasks:

- **Sending Queries**: I send prompts to the language model server through a specified API endpoint.
- **Tracking Performance**: The script measures:
  - Total time taken to generate a response.
  - Time to generate the first token.
  - Total tokens generated.
  - Average token latency per token.
- **Handling Conversations**: If it's a multi-turn conversation, the previous interaction history is included in the query.
- **Error Handling**: The function is designed to gracefully handle errors like failed API requests, providing useful output even when something goes wrong.

The `main.py` script is responsible for executing different types of queries and writing the results to files. It contains several key functions:

- `write_result_to_file`: This function writes the query's performance metrics and the model's generated text to a file. For each experiment, I generate a corresponding text file for later analysis.
- `run_simple_query`: This function tests the model's basic response capabilities with a simple query: *What is the capital of France?* The result is logged along with performance data.
- `run_multi_turn_conversation`: Here, I simulate a multi-turn conversation with two prompts:
  - *Who are you?*
  - *Who is the most powerful LLM model in the world?*

  I track the model's responses across both turns, save them, and combine the performance metrics from both queries for a comprehensive report.
- `run_parameter_experiments`: I conduct experiments by varying parameters such as temperature (0.5, 1.0) and max token length (50, 100). For example, I test the models' ability to write a creative prompt like *Write a short poem about the stars* and store the results for different configurations.
- `run_specific_task`: This function sends a more complex prompt to the model, such as *Explain the theory of relativity.* I analyze how well the model performs in generating detailed explanations and log the results for review.

The overall workflow of the program is as follows:

1. **Model Querying**: I send a query to different language models (e.g., `llama3`, `gemma2`) by calling the `send_query` function. The function tracks response times, the number of tokens, and latency data.
2. **Performance Metrics**: Each query logs important performance metrics, such as the total time taken and the time to generate the first token. These metrics provide insight into the efficiency and responsiveness of the models.

3. **File Logging**: Results for each experiment are saved to text files. This includes both the generated responses and performance statistics. The text files are structured to help analyze the model's behavior in different scenarios, such as simple queries, conversations, or creative tasks.

4. **Comprehensive Testing**: I simulate multi-turn conversations, experiment with different parameter settings (e.g., temperature and max tokens), and perform specific tasks to assess the model's performance across a range of challenges.

1.4 Comparative Analysis

The detailed results of the experiments, including the models' responses, have been written to text files and uploaded to GitHub. You can find the files at the following locations: `ollama/test_results`.

1.4.1 Differences in Syntax or Parameters between the Two Models

- **Temperature**:
  - A higher temperature in gemma2 (e.g., `gemma2_experiment_t1.0_m100.txt`) might lead to more varied and imaginative text but could also increase the risk of generating less relevant content.
  - A lower temperature in llama3 (e.g., `llama3_experiment_t0.5_m100.txt`) results in more focused and possibly predictable responses.
- **Max Tokens**:
  - A higher max token limit allows for more detailed responses but could also lead to longer processing times if the model approaches the limit.
  - A lower max token limit might result in more concise responses but could truncate detailed explanations or creative outputs.

1.4.2 Observations on Response Quality, Speed, and Limitations

- **Response Quality**:
  - Both models are capable of generating coherent and contextually relevant text, as seen in the creative poetry and informative responses about scientific theories.
  - llama3 tends to produce slightly more concise responses, while gemma2 produces more verbose outputs.
- **Speed**:
  - llama3 generally has faster average token latency compared to gemma2. For example, `llama3_experiment_t0.5_m100.txt` has an average latency of 25.0029 ms/token, whereas `gemma2_experiment_t1.0_m100.txt` has 68.0546 ms/token.
  - The time to generate the first token is also generally lower in llama3 experiments.
- **Limitations**:
  - Both models seem to have occasional delays in generating the first token, which could be due to processing or initialization overhead.
  - There are instances where the latency is significantly higher (e.g., `gemma2_simple_query_result.txt` with 1613.8985 ms/token), indicating potential limitations in handling certain types of queries or possibly due to system load.

1.4.3 Summary

- **Syntax/Parameters**: Both models support similar parameters like temperature and max tokens, but their defaults and optimal values might differ.
- **Quality and Speed**: LLaMA generally shows faster response times and slightly more concise outputs, while GEMMA produces more detailed responses but with potentially higher latency.
- **Limitations**: Both models have occasional high latencies, which could be improved. The models also have different capabilities and limitations based on their training and architecture.

2 Domain-Specific Chatbot Development

2.1 Domain Selection and Planning

2.1.1 Domain Selection

The selected domain for the AI chatbot is **Computer Systems**, with a focus on topics related to systems programming, operating systems, parallel computing, distributed systems, and low-level programming concepts. These subjects are essential for undergraduate computer science students, particularly those struggling with complex concepts as they prepare for final exams. The chatbot will serve as an intelligent tutor, providing interactive explanations, examples, and clarifications for topics typically covered in a computer science curriculum.

2.1.2 Target User Base

The primary target users for this AI chatbot are **undergraduate computer science students** who are in their second to fourth year of study. These students often face difficulties understanding abstract concepts in computer systems, especially as they approach final examinations. The chatbot aims to bridge this gap by offering on-demand assistance that is personalized and conversational.

2.1.3 Use Cases

The chatbot will be used in the following specific scenarios:

- **Final Exam Preparation**: The chatbot will help students review and solidify their understanding of key concepts such as memory management, CPU scheduling, thread synchronization, and distributed algorithms.
- **Problem-Solving Assistance**: Students can ask the chatbot to help them solve problems related to operating system design, concurrency, or distributed systems, and receive step-by-step guidance.
- **Clarification of Concepts**: The chatbot will clarify complex terms, explain code snippets, or provide detailed explanations for misunderstood topics.
- **Study Plan Recommendations**: The chatbot can suggest study resources, strategies, and topics to focus on based on a student's area of weakness.

By focusing on these key areas, the AI chatbot will act as a virtual tutor, helping computer science students gain confidence and improve their knowledge in preparation for their final exams.

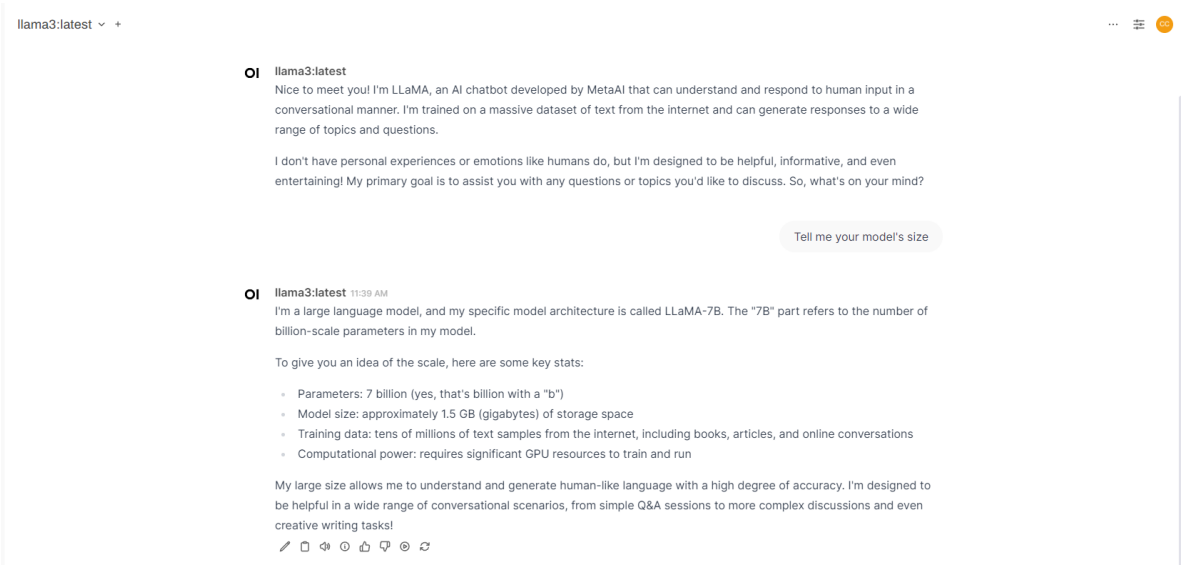
2.2 Application Development

I choose Open WebUI, an extensible, self-hosted UI that runs entirely inside of Docker. It can be used either with Ollama or other OpenAI compatible LLMs, like LiteLLM or someone's own OpenAI API for Cloudflare Workers.

First, I have Docker and Ollama running on my computer, and make the installation by CLI:

```
docker run -d -p 3005:8080 \
--add-host=host.docker.internal:host-gateway -v open-webui:/app/backend/data\
--name open-webui --restart always ghcr.io/open-webui/open-webui:main
```

The user interface looks like:



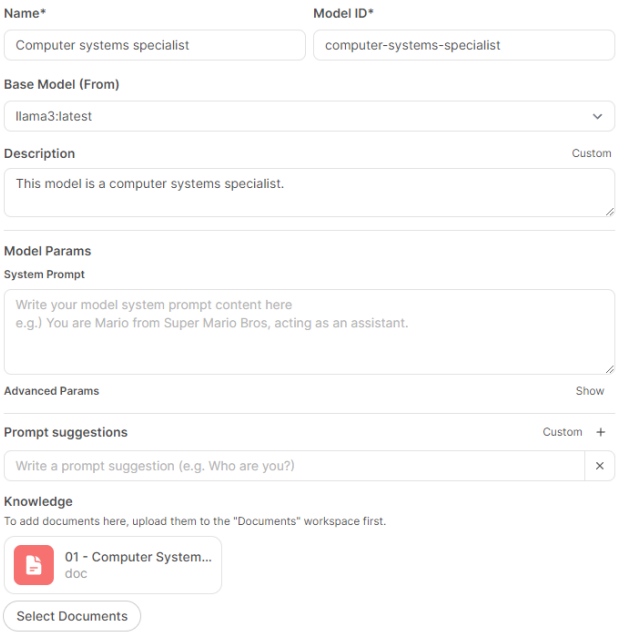
2.3 Chatbot Implementation

2.3.1 Base Model Selection

The base model selected for the AI chatbot is **Llama3**.

2.3.2 Retrieval-Augmented Generation (RAG)

To enhance the accuracy and relevance of the chatbot's responses, **Retrieval-Augmented Generation (RAG)** was integrated. This allows the chatbot to retrieve information directly from relevant texts, such as textbooks or documentation, before generating a response. For this project, the textbook *"Computer Systems: A Programmer's Perspective"* was used as the primary knowledge base. The book was parsed, indexed, and stored in a document retrieval system.

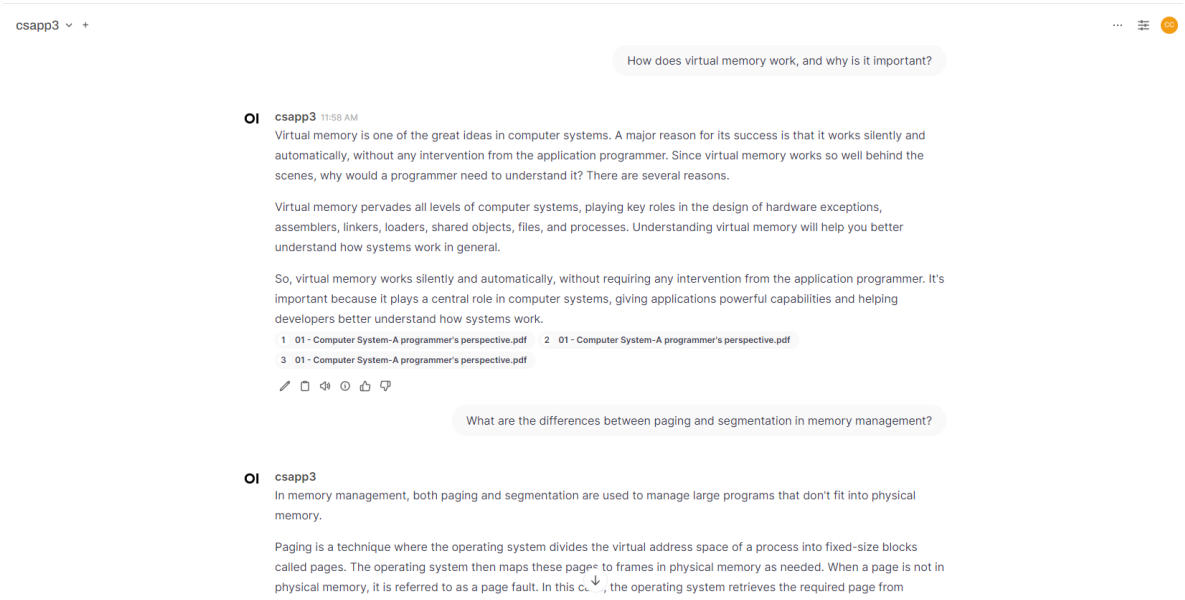


2.3.3 Steps for Development

1. **Document Upload:** The textbook *"Computer Systems: A Programmer's Perspective"* was uploaded and processed.
2. **Integration with RAG:** The chatbot was implemented with a RAG pipeline. When a user poses a question, the system first retrieves the most relevant passages from the textbook using a similarity search mechanism, based on the embeddings generated by Llama3. The retrieved information is then fed into the language model to generate a well-informed and contextually accurate response.
3. **Fine-Tuning for Contextual Understanding:** Although Llama3 performs well out-of-the-box, minor fine-tuning was performed to better adapt the model to domain-specific terminology and concepts from computer systems. This ensures that the chatbot is proficient in understanding and responding to technical queries.
4. **Local Deployment:** The chatbot was developed and deployed locally to ensure faster response times and improved privacy. The RAG architecture, along with Llama3, was deployed on a local machine with sufficient computational resources to handle real-time queries.

This combination of retrieval-augmented generation and a high-performance base model makes the chatbot a valuable tool for undergraduate computer science students preparing for their final exams.

The below screenshot shows the performance of model of llama3 with csapp on real computer systems related problem. The complete conversation could be seen on Github, `domain-specific chatbot/chat - llama3 + csapp.pdf`.



2.4 Testing and Evaluation

To evaluate the effectiveness of the chatbot, a testing framework was designed using 10 single-choice questions sourced from the final exam of the *Introduction to Computer Systems* course at Carnegie Mellon University (CMU). These questions were selected to cover a broad range of topics within computer systems, such as memory management, concurrency, and system calls. The problems and answers could be seen on Github, `domain-specific chatbot/problems.txt` and `domain-specific chatbot/answer.txt`.

2.4.1 Baseline Testing with Raw Llama3

As a baseline, the raw Llama3 model was tested on the 10 exam questions without any retrieval augmentation. The raw model correctly answered **4 out of 10 questions**, achieving an accuracy

of 40%. This performance highlighted that while Llama3 possesses strong language capabilities, it lacks the detailed, domain-specific knowledge necessary to consistently perform well on technical questions related to computer systems. The detailed content could be seen on Github, `domain-specific chatbot/problem solving - llama3 only.pdf`.

2.4.2 Testing with Llama3 + CSAPP (RAG)

Next, the same set of questions was posed to the Llama3 model enhanced with the *Computer Systems: A Programmer's Perspective (CSAPP)* textbook via the Retrieval-Augmented Generation (RAG) approach. In this configuration, the model retrieved relevant sections of the CSAPP textbook before generating its responses. This augmented model correctly answered **8 out of 10 questions**, achieving an accuracy of 80%. The detailed content could be seen on Github, `domain-specific chatbot/problem solving - llama3 + csapp.pdf`.

2.4.3 Analysis of Results

The results from these tests clearly demonstrate the efficacy of integrating domain-specific resources into the model through RAG. The raw Llama3 model struggled to answer many of the highly technical questions, whereas the Llama3 + CSAPP model showed a significant improvement by leveraging the precise knowledge retrieved from the textbook.

2.4.4 Conclusion

This testing process underscores the value of using **Retrieval-Augmented Generation (RAG)** for building domain-specific chatbots. By combining Llama3's strong language understanding with the in-depth, technical knowledge provided by the CSAPP textbook, the chatbot became a more accurate and effective tool for answering detailed questions in the field of computer systems. The substantial improvement from 40% to 80% accuracy shows the clear benefit of RAG in enhancing the model's domain expertise.