LiteRAGBot

Cameron O’Dell
School of Science and Engineering
University of Missouri
Kansas City, MO, USA
ccomfb@umkc.edu

Daniel Sponsler
School of Science and Engineering
University of Missouri
Kansas City, MO, USA
dpsfgp@umkc.edu

Reed White
School of Science and Engineering
University of Missouri
Kansas City, MO, USA
wrwzq8@umkc.edu

Introduction

Modern chatbot systems are relatively simple to develop and implement. Being almost exclusively powered by large-language models (LLM), they are more powerful than they’ve ever been. However, training these LLMs to meet specific business needs is a highly-technical, expensive, and slow process. So, many individuals and businesses have turned to Retrieval-Augmented Generation (RAG) systems, which utilize corpuses of provided information to provide an LLM with additional context during inference. These RAGs allow entities to modify the behavior of an LLM without the need to retrain them. However, even RAGs are oftentimes beyond the technical abilities of the average business or individual, which this project seeks to remedy.

Objectives

This project aimed to develop a simple, portable, customizable proof-of-concept chat system using retrieval augmented generation (RAG). This system was to be kept as light as possible while still being able to run on a variety of user hardware with a variety of large-language models (LLM), both local and remote. To meet these requirements, LiteRAGBot was designed with the following objectives (in no particular order):

1. It uses RAG to generate text content, in the form of a chatbot.
2. It includes a webapp user interface (UI) to interact with the RAG system.
3. It allows users to define and utilize their own LLMs
4. It allows users to install and run it on a variety of Windows or Linux servers with minimal effort
5. It allows users to provide a corpus to the RAG system in a variety of formats.
6. It achieves a reasonable level of inference performance.
7. It uses as few resources as possible to achieve this performance.
8. It is as inexpensive as possible.

Unfortunately, when discussing generative LLMs, a “reasonable level of inference performance” is mostly subjective. It is nearly impossible to automatically score the accuracy and precision of a generative LLM. We attempted to at least standardize this measure, shown in the next section.

Methodology

Before we could begin any form of testing, we first had to develop a working RAG chatbot. We quickly created a simple RAG system using the following publicly-available components (which is available in the referenced GitHub repository) [1]:

- Python (for the RAG) [2]
- Ollama (for the LLM) [3]
- Streamlit (for the webapp UI) [4]
- Chroma (for the corpus datastore) [5]

Which interact according to the following architecture:

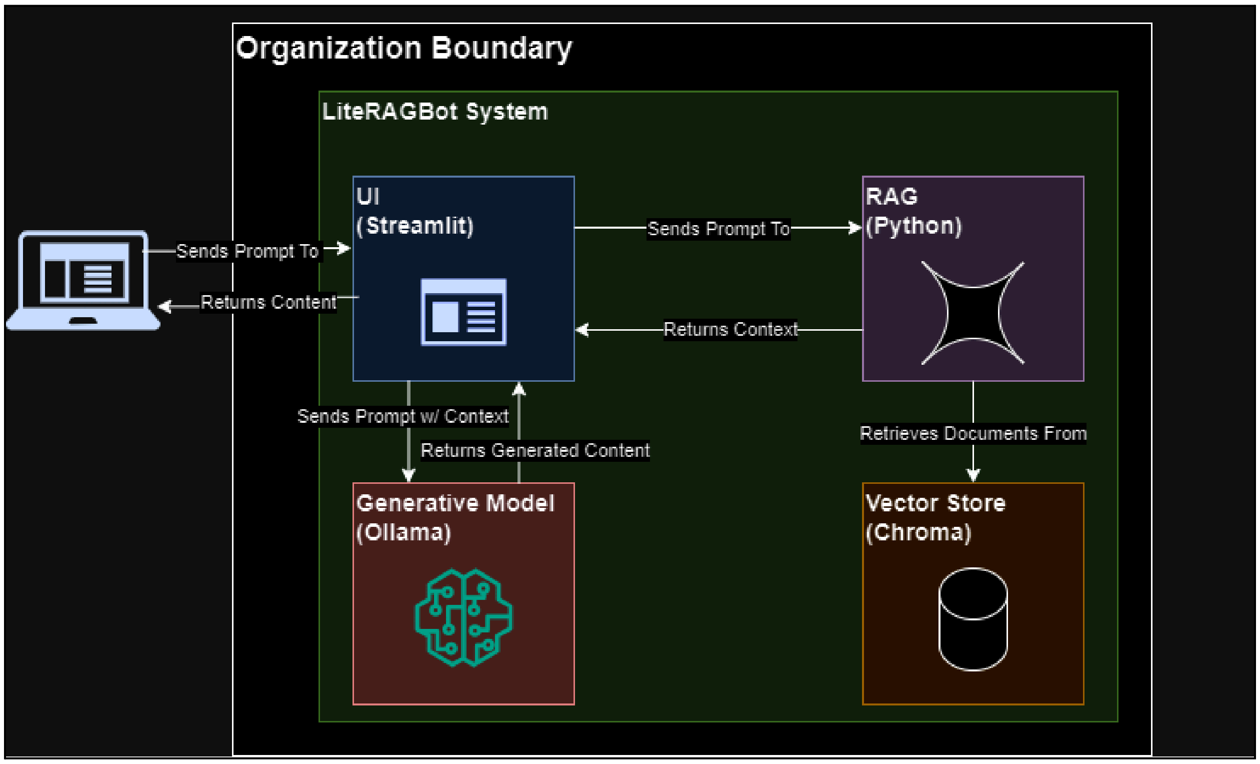

FIGURE I. LiteRAGBot High-Level Architecture

Each component of the above architecture performs the following function(s):

- End User: a customer/client of the small business or individual that installed this system. Interacts with the UI, usually in the form of a question expecting a response.
- UI: provides the webapp UI. Receives end user prompts, sends prompts and retrieves content from the RAG system, sends prompts/content to and receives content from the LLM, and displays results to the end user.

- RAG: provides the RAG system. Receives prompts from the UI, queries the vector store for similar results, and sends retrieved content back to the UI.
- Vector Store: provides the vector store. Stores the corpus, receives queries from the RAG system, and sends similar results in the corpus back to the RAG system.
- Generative Model: provides the LLM. Receives prompts and retrieved content from the UI, generates a reply based on those details, and sends the generated response back to the UI.

Once we had the RAG system, we needed a corpus to test it with. Our project, originally called “GrocerBot”, was initially conceived as being a chatbot used exclusively by grocery stores, so we started our search with a structured grocery dataset. We found a feasible dataset from Walmart on Kaggle, so we began building a Walmart corpus [6]. Since the first document was a structured table in a CSV, we also took some unstructured articles from Wikipedia: a general overview of Walmart, saved as a DOCX file; the history of Walmart, saved as a TXT file; and a list of assets that Walmart owns, saved as a PDF [7-9].

Now that we had a corpus and a RAG, our first real challenge was integrating the two. Vector datastores, such as Chroma, work by taking unstructured documents, “chunking” (splitting) them up by some standardized yet arbitrary measure (such as by number of words), “embedding” them by converting all of the tokens (words) in the split chunks into numbers, and then storing them as separate embeddings. These embeddings are then retrieved via plaintext queries by mathematically calculating how similar the tokens in the query are to the tokens stored in the embeddings. This allows RAG systems to quickly retrieve “context” for any given query from documents stored in the vector store.

Chunking and storing the unstructured documents (the TXT, PDF, and DOCX files) was relatively simple and straightforward, but chunking and storing the original CSV dataset was significantly more difficult. Vector datastores are not necessarily the “best” place to store such documents, as their structure makes them more suitable for traditional, relational databases, such as SQL databases. However, RAG systems are currently unable to easily integrate with these relational databases: while it is possible to create a RAG that can query a SQL database, doing so requires significantly more resources and knowledge than the average user has, violating our fourth objective. So, we took a much simpler approach: we “chunked” the CSV table based on column names, such that each unique combination of values in the given columns (in our case, “CATEGORY” and “BRAND”) was stored as a separate embedding. For example, all rows in the table that had “Great Value” products that fell under the category of “Hummus, Dips, & Salsa” were collected and stored as one embedding. This strategy allowed the RAG system to retrieve and provide the LLM product context relatively well, however it did have a downside: since the number of rows for each chunk varied greatly, some embeddings were too large to “fit” into the LLM. Splitting the table up by both “CATEGORY” and “BRAND” alleviated most of these issues, but it’s very likely that any large table will see this issue.

With a working RAG system and a stored corpus (fulfilling objective five), we moved on to testing. At first, we wanted to see if other vector datastores such as Pinecone or Weaviate would yield better results [10, 11]. While they did in some areas, they were quickly ruled out, as most other vector stores are either only available remotely (as Software as a Service, or "SaaS" offerings) or are significantly more complex, which violates our fourth, seventh, and eighth objectives. So, we primarily focused on testing different LLMs.

As previously stated, measuring the performance of LLMs is mostly subjective. We attempted to at least standardize our measurements by developing a list of twenty prompts, covering the four documents in our corpus. These prompts all involved asking questions with factual answers, which were defined in the corpus. Each LLM was loaded into the RAG system (with all configurations left to their defaults), which was given these prompts in the same order. Responses were then graded as simply as possible: if the LLM gave a result that contained the expected factual information from the corpus, it was given a single point. If the LLM gave a result that at least approached the factual information, or used some other logic to arrive at a similar conclusion, it was given half a point. These points were then summed and divided by twenty, to arrive at an "accuracy".

In addition to the above, other metrics were tracked as well, mainly the size of each LLM in terms of storage size (to approximate how much storage is needed) and number of parameters (to approximate how much processing power is needed). Larger LLMs require more storage and processing power, and our goal was to balance these requirements with accuracy (objectives six, seven, and eight). The results of these tests and comparisons are shown in the below section.

However, we still had two objectives left: three and four. Even though we had tested a suite of LLMs to arrive at a reasonable default model, we still wanted to grant users the ability to choose their own model. This ability went hand-in-hand with the ability to quickly and easily install the entire system. To that end, we developed a pair of simple scripts that users can run on their Windows or Ubuntu Linux machines (a PowerShell script for Windows, and a Bash script for Ubuntu) [1]. These scripts read from a simple configuration file, which allows users to define their own configurations and LLM, and will download, install, and run all necessary technologies in a virtual Python environment. No user modifications should be necessary to install and run our default, example version of LiteRAGBot: users need to simply download the GitHub repository and run the appropriate scripts. Instructions are included in that repository.

Results

Testing the various LLMs led to some interesting results. The initial version of LiteRAGBot was developed using llama3.1, and a great deal of prompt engineering was required to achieve reasonable performance [16]. This initial testing revealed some surprising things: the RAG system was shockingly good at retrieving and processing data retrieved from the structured CSV, missing only a single prompt. However, perhaps more shockingly, the RAG system was much more hit-or-miss on the unstructured data, potentially due to our efforts to tune the system to account for the structured data. The initial llama3.1 RAG system still managed to achieve a reasonable, estimated 58% accuracy, and all of the prompts it did miss still generated expected failure responses. It did not hallucinate or respond with any made-up information.

Once the initial llama3.1 version was tested, we applied the same methodology to four other LLM families: Meta's llama3.2, Alibaba's qwen2.5, Mistral AI's mistral, and Microsoft's phi3.5 [17-20]. For llama3.2 we tested both the 2B and 1B parameters versions, for qwen2.5 we tested both the 7B and 0.5B parameters versions, and for gemma we tested the 7B and 2B parameter versions. In total, we tested nine different LLMs, which gave us the following results:

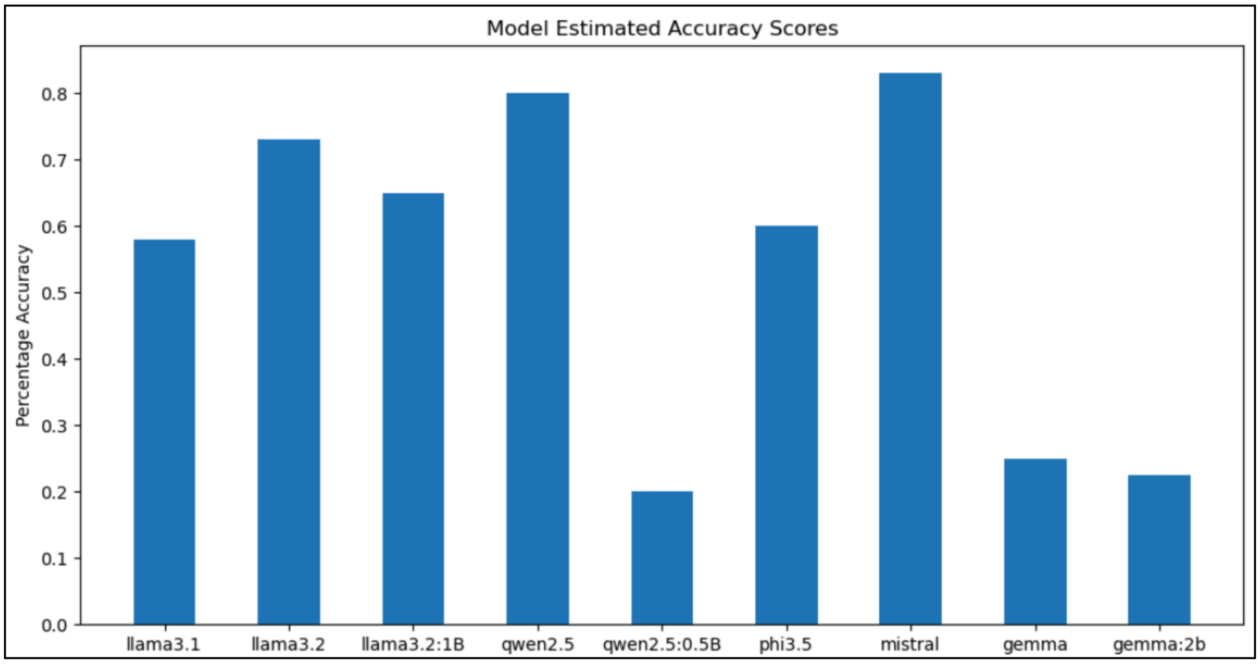

FIGURE II. Model Estimated Accuracy Scores

The mistral and qwen2.5 (7B) models generated the most correct results, with an estimated accuracy of 83% and 80% respectively. The llama3.2 (2B) model achieved similar results, with an estimated accuracy of 73%, with llama3.2 (1B) falling slightly behind. The llama3.1 and phi3.5 models achieved similar results (around 60%), with there then being a noticeable gap in performance for the lowest scoring models, gemma (7B), gemma (2B), and qwen2.5 (0.5B), which all scored around 20%.

These results become far more interesting when comparing them with the sizes of the models:

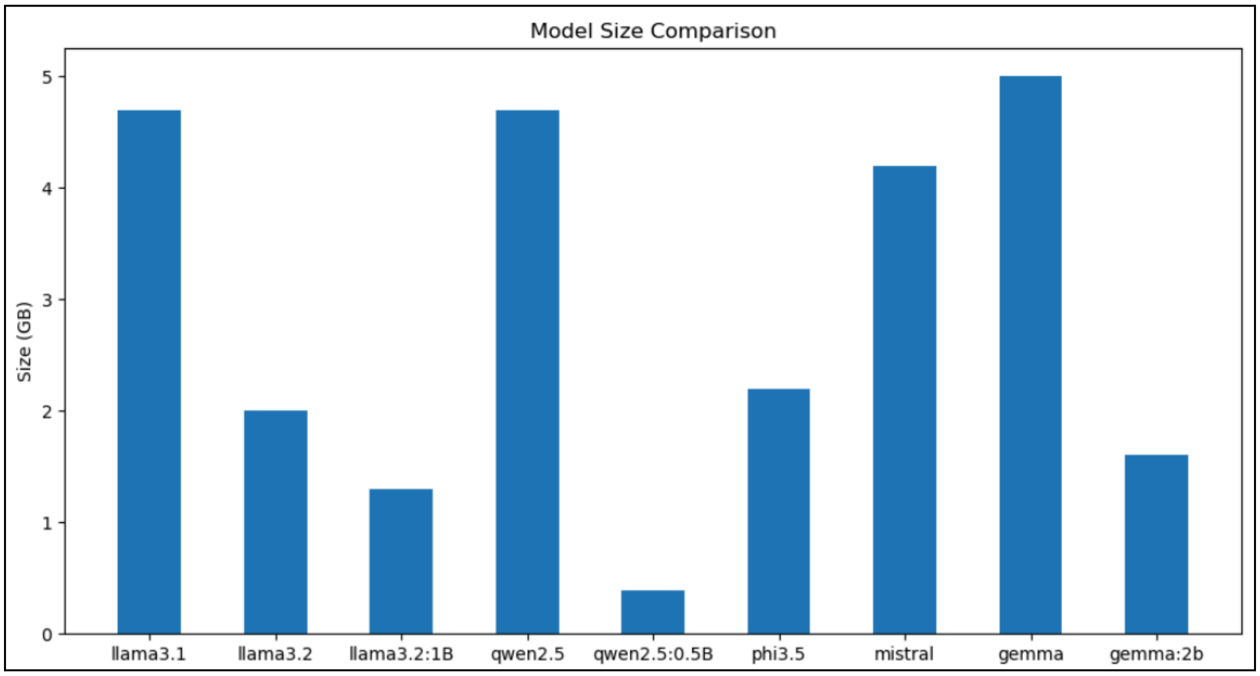

FIGURE III. Model Size Comparison

Of course, the size of each model in GB very closely correlates to the number of parameters the model has:

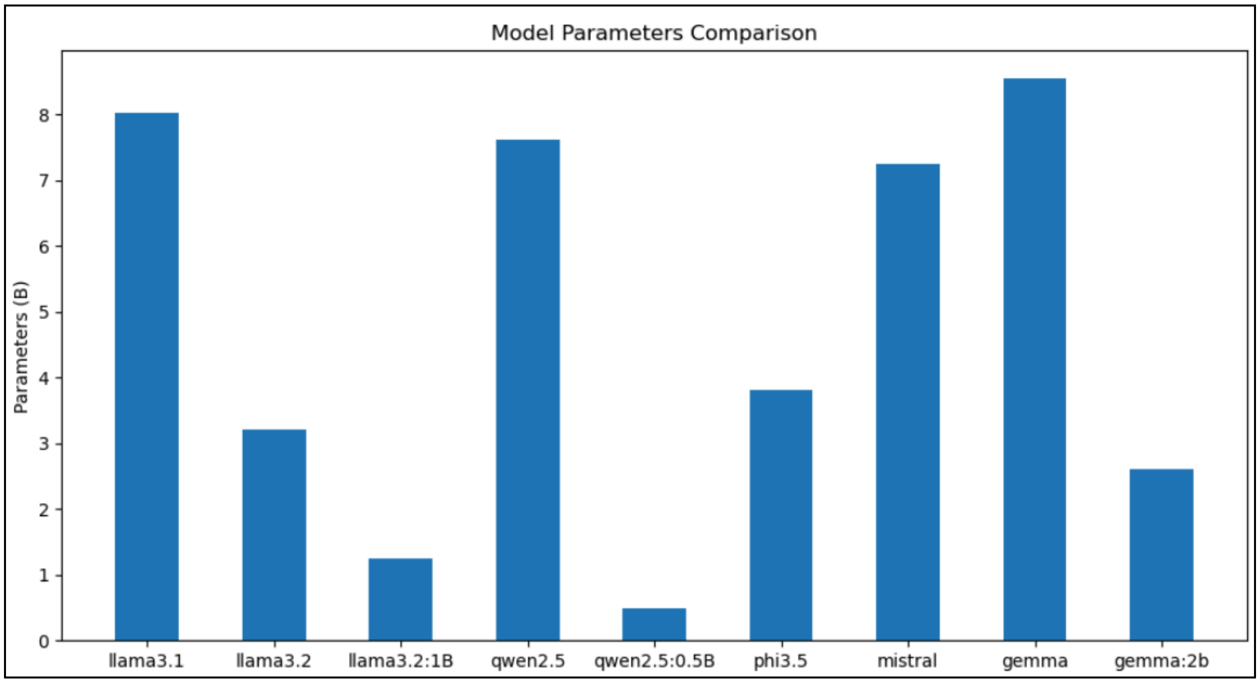

FIGURE III. Model Parameters Comparison

Generally speaking, larger, more complex models tend to perform better in most applications, which we see in our testing results. The top two models, mistral and qwen2.5 (7B), are the fourth and second largest models respectively (in terms of GB), while the worst two models, gemma (2B) and qwen2.5 (0.5B), are the third and first smallest models. But we do see some exceptions, some fairly severe ones. For example, gemma (7B), the largest model, performed the third worst. Most notably is the performance of the two llama3.2 models: llama3.2 (2B), the fourth smallest model, performed the third best, while llama3.2 (1B), the second smallest model, performed the fifth best.

As our objectives for this project prioritized both performance and resource usage, we determined that the llama3.2 (2B) model achieved the best balance of useful results and resource footprint. It has a roughly 10% drop in estimated accuracy over the best performing two models, but at one-half the size and nearly one-quarter the parameters of those larger models, llama3.2 (2B) puts up an extremely impressive show.

Discussion

The significance of LiteRAGBot may not be immediately apparent. There are many commercially-available SaaS chatbot systems, such as Google's DialogFlow, Amazon's Lex, or Botpress [12, 13, 14]. There are even other smaller, self-hosted options, such as Botkit [15]. But these systems are all expensive, risky, complicated, or some combination thereof. SaaS options, such as Lex, are relatively simple to implement but require an ongoing subscription. These also typically required the corpus to be stored in the cloud. Existing self-hosted options, like Botkit, require a great deal of effort to implement.

In its current form, LiteRAGBot is inexpensive, self-hosted, and simple to implement. It offers the core features of these existing systems to businesses and users who would otherwise be unable to afford or implement those systems. It is a RAG-powered chatbot, stripped to its most basic form so that it can be installed anywhere and streamlined so that it can be installed by anyone. Because it was developed using basic Bash, Powershell, and Python, it is also easily modifiable and extensible. Administrators and power users can easily go beyond the included configuration file and tweak the code directly to fit their needs.

However, LiteRAGBot does have its downsides. Because it is stripped down, it is missing many features that larger businesses require. It is not multi-modal, it does not support automatic scaling, and it has no enterprise security or management features. It is a proof-of-concept framework and as such it is not appropriate for enterprise usage in its current form.

Conclusion

This project did not aim to make any massive discoveries or to develop any ground-breaking technologies. It instead aimed to take those ground-breaking technologies and simplify them, so that they could be more widely-adopted by the general public. We believe that LiteRAGBot accomplishes this, using publicly-available, state-of-the-art RAG, LLM, and vector store components, along with a little bit of Bash, Powershell, and Python. Most modern LLM-based technologies focus on size, power, and complexity. LiteRAGBot bucks these trends, implementing a small, simple framework that is easily modifiable and implementable on most any hardware. LiteRAGBot is simple, and for most users, simplicity is key.

Future Work

There are many areas in which the LiteRAGBot system could be improved. As a proof-of-concept, it is extremely bare-bones and is lacking many of the required features of an enterprise product. Some areas in which this system would be improved include:

- Improved storage/retrieval of unstructured data.
- An installation GUI.
- A bespoke user interface, with improved options for security.
- A bespoke vector store.
- Development in a more secure language.
- Improved packaging, so that all dependencies are included.
- Improved webapp integration, such as a plugin that integrates with popular hosting services.

However, it is worth noting that the development and implementation of some of these features may detract from LiteRAGBot’s overall goal of being a simple, flexible, customizable, and lightweight RAG system.

References

[1] Cameron O’Dell, Daniel Sponsler, Reed White, “5542-0001 Team Project: LiteRAGBot”, Github. https://github.com/mist861/5542-Big-Data-Analytics (December 2024).
[2] Python Software Foundation, “Python”, Python. https://www.python.org (December 2024)
[3] Ollama, “Get up and running with large language models.”, Ollama. https://ollama.com (December 2024).
[4] Snowflake Inc, “A faster way to build and share data apps”, Streamlit. https://streamlit.io (December 2024).
[5] Chroma, “Chroma is the open-source AI application database. Batteries included.”, Chroma. https://www.trychroma.com (December 2024).
[6] BarkingData, “Walmart Grocery Product Dataset”, Kaggle. https://www.kaggle.com/datasets/polartech/walmart-grocery-product-dataset (December 2024).
[7] “Walmart”, Wikipedia. https://en.wikipedia.org/wiki/Walmart (December 2024).
[8] “History of Walmart”, Wikipedia. https://en.wikipedia.org/wiki/History_of_Walmart (December 2024).
[9] “List of assets owned by Walmart”, Wikipedia. https://en.wikipedia.org/wiki/List_of_assets_owned_by_Walmart (December 2024).

[10] Pinecone Systems, Inc., “Build knowledgeable AI”, Pinecone. https://www.pinecone.io (December 2024).
[11] Weaviate, “The AI-native database for a new generation of software”, Weaviate. https://weaviate.io (December, 2024).
[12] “Conversational Agents and Dialogflow”, Google. https://cloud.google.com/products/conversational-agents?hl=en (December 2024).
[13] “Artificial Intelligence (AI) Service - Amazon Lex”, Amazon. https://aws.amazon.com/pm/lex/ (December 2024).
[14] “The Complete AI Agent Platform”, Botpress. https://botpress.com (December 2024).
[15] “Botkit”, Github. https://github.com/howdyai/botkit?tab=readme-ov-file (December 2024).
[16] Meta, “llama3.1”, Ollama. https://ollama.com/library/llama3.1 (December 2024).
[17] Meta, “llama3.2”, Ollama. https://ollama.com/library/llama3.2 (December 2024).
[18] Alibaba Cloud, “qwen2.5”, Ollama. https://ollama.com/library/qwen2.5 (December 2024).
[19] Mistral AI, “mistral”, Ollama. https://ollama.com/library/mistral (December 2024).
[20] Microsoft, “phi3.5”, Ollama. https://ollama.com/library/phi3.5 (December 2024).