Journal of Systemics, Cybernetics and Informatics (2024) 22(5), 96-100
https://doi.org/10.54808/JSCI.22.05.96

Improving performance of Local Chatbot with Caching

John JENQ
Montclair State University
Montclair, NJ 07043, USA

ABSTRACT[1]

Chatbots and the technology behind them are widely used in many places and in various ways. Retrieval Augmented Generation AI framework has gained its popularity by its linking of large language model with private dataset. It enables one to run AI locally and privately with the most updated information and knowledge. In this report, we aim to improve the local private chatbot response time by using a cache. From our experimental results, the majority of time spent during the query process is in the generation of the response. The response time can be significantly improved when there is a hit on the cache system which enables us to return the response to the user immediately without going through the generation step. In this report, we focus our efforts on improving the turnaround time of the generation step. The cache is organized into categories which can be used for efficient searching. User's query information such as query string, embedding information, and its response are recorded and stored in the cache. Experiment results are presented and the issues of speed up of request response turnaround time is addressed.

Keywords: Chatbot, Cache, Embeddings, Similarity Search, LLM, RAG

1. INTRODUCTION

Artificial Intelligence (AI) has evolved rapidly due to advancements in machine learning and natural language processing (NLP). Devlin et al. revolutionized the field of NLP by demonstrating how pre-trained transformers can achieve great performance in various tasks like answering questions and conversational AI [1]. While chatbots have been around for some time, they have recently gained recognition after ChatGPT became viral in 2022. Brown et al. presented a large-scale transformer model GPT-3 that displayed the potential capability for creating conversational agents and QA systems [2]. Chatbots and the technology behind them are widely used in many places and in various ways. For example, in the hospitality industry, Athikkal and Jenq implemented a hospitality voice chatbot to answer various questions related to a hotel application [3]. Wang et al., proposes an experiment of counseling hospitality employees using conversational AI chatbots [4].

Large language models (LLM), which are used in Generative AI applications, has become increasingly important in many areas as it provides natural language processing capabilities, prediction analysis, and help in the decision-making process. LangChain is an AI framework that integrates with external tools to form an ecosystem by collecting all the required components for creating private chatbots quicker. For how LangChain can be used to simplify the integration of LLM and applications, see [5].

Ma et al., discussed how LLMs is used in food science application in [6]. In the management field, Aguinis et al., showed how AI can help people quickly finish tasks, like human resources management [7]. In contract management, Wong et al., discussed how to incorporate construction contract domain knowledge to enhance language models which help identify construction contract risks in order to avoid loss [8].

In the medical fields, Olszewski et al., compared five chatbots (Gemini, Microsoft Copilot, PiAI, ChatGPT, ChatSpot) from the internet to study the quality of these chatbots in the area of cardiovascular health and concluded that chatbots vary in length, quality, and readability [9]. In [10], Alkhalaf et al., experiment to extract malnutrition information by using the efficacy of zero-shot prompt engineering and RAG to summarize both structured and unstructured data. Hart et al., investigated the use of LLMs in the areas of clinical and anatomic pathology [11].

In this paper, we use caching to improve the performance of local private AI chatbots which can be on run on a server. The idea of caching has been used for many years to improve the performance of computer systems and internet proxy servers. Cache memory was first developed in the computer hardware design of memory hierarchy. The CPU can access the cache more quickly compared to accessing the main memory. The concept was used on the paging system design to allow fast retrieval of memory page. It is a deterministic mapping process. To access a target block of memory such as a page in the main memory, one first checks if the target is in the cache memory. The result is either a hit or a miss. A similar idea can be extended to proxy server design, which stores the web page item and its content on the server's local storage. When there is a request from user's agent, the browser, to a particular web page or item, such as an image, the item stored on the proxy server can then be returned to the request computer immediately without request to the server that owns that item. This significantly reduces the network traffic and speeds up the turnaround time.

The main purpose for both examples, computer system memory management and proxy server web item management, is to shorten the total time of the request and response cycle. Most of today's LLMs use a probabilistic approach instead of a deterministic approach. Retrieval Augmented Generation (RAG), a term coined by Lewis et al. [12], is an AI framework which intends to link the generative AI with specific source of domain knowledge or information, such as the most current information about a company's new regulations, or a new school policy, or new medical research results, etc. This information may not be available on the Internet but can be embedded into

[1] I want to express my sincere gratitude to JSCI Editorial board for accepting and publish this paper.

the vector store in order to become available for retrieval processing. The system will then retrieve specific information and augment with the LLM to generate a response to the user. As stated by [13], "almost any business can turn its technical or policy manuals, videos or logs into resources called knowledge bases that can enhance LLMs."

These domain related data or documents are divided into chunks. Each chunk will then be tokenized. Each token is embedded as a vector. What is an embedding? An embedding is a mapping of a token to a vector of numbers. The vector contains information about this token. To simplify the concept, let's assume each word is a token. Then a word can be represented as a vector which is a collection of numerical values. Thus, the process of text embedding translates a word into meaningful numbers. The resulting collection of numbers (a vector) are deterministic and carry meaning, so the vector (the list of numbers) is also deterministic and carry meaning. It uses hundreds or thousands of numbers to represent a word like 'king'. The total number of these numbers in the vector which represent a token is called the *dimension* of the LLM. Different models use a different number of dimensions to represent the meaning of a token.

These document chunks are then stored into the vector store. Users can enter a query and the query is encoded and used to search the vector store to retrieve relevant document chunks. For example, if our domain is 'Python Programming Language' and if a user wants to know about a Python list, the query will be embedded and some type of similarity checking, like cosine similarity, will be performed to find the document chunks with the best fit. These document chunks will then be used along with the user query be input into the generator (transformer) to generate the output. A transformer is a deep learning architecture developed by Vaswani et. al. at Google. It is based on the multi-head attention mechanism, proposed by the famous and important paper "Attention Is All You Need"[14].

According to Bolliboina and Jenq in [15], the private local AI chatbots can solve the privacy and security concerns for some industries such as the banking industry. But it is slower in response time when compared with the public chatbots. In this report, we aim to improve the response time by introducing a caching component. Unlike most of today's programming languages which are unambiguous, natural languages are ambiguous. Often times, its meaning is context sensitive. Not only that, one can form various sentences which have the same or similar meanings. If we were to implement a caching component to speed up the process, we have to answer several questions: (a) How do we organize and categorize these queries together so we don't waste our storage while caching and trying to achieve the speed up? and (b) How do we map a query to its slot and retrieve the answer or possible answers? In this report, we try to answer both of these questions.

In section 2, we outline the system design and its implementation. Section 3 contains of the experimental results and concluding remarks are in section 4. Section 5 lists some of the possibilities for future work and improvements.

2. SYSTEM CONFIGURATION AND IMPLEMENTATION

In this report, we utilize Python's *ollama* and *chromadb* modules. The chroma module is used in our project to create the vector embeddings. The ollama module is used for generating responses to a user's query. **Figure 1** shows the process of the vector embedding. We used the RecursiveCharacterTextSplitter text splitter of LangChain to split the pdf files into document chunks. This procedure allows us to set chunk size and the overlap size. These document chunks are fed into the embedding process which will generate a vector embedding and be stored into the vector store. A smaller chunk size will increase the total number of documents generated, while a larger chunk size decreases the number of documents and therefore reduces the number of embeddings in our vector collection. Our program uses chromadb's persistent client to create a vector store which will store the vector on the local disk so we don't have to create the vector store repeatedly each time we start the program.

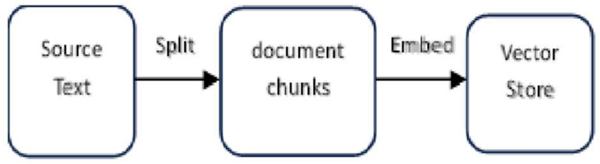


Figure 1. Vector Embedding

An embedding is a numerical representation of a piece of information. For example, an embedding can be used to represent a text, a document, an image, or audio, etc. Thus, it is a translation process. Text embedding translates words into meaningful numbers and the resulting numbers (a list) are deterministic and it carries meaning, so the vector (the list of numbers) is also deterministic and therefore carry meaning. Similarly, image embedding translates a picture into a vector based on categories, like type of animal, type of flower, color, background etc. Once again, it is a list of numbers that represents the object under our consideration. Therefore, given a text embedding, we can determine what kind of image a sentence describes, and the same can be done with audio or video embeddings. Embeddings can be used in works such as clustering, searching, classification, recommendation, etc. The idea behind using embeddings to do the above-mentioned tasks is because the process of embedding enables us to find the k-nearest neighbors in a *n-dimensional* space using distance between embeddings.

Different models have different dimensions *n*. For example, ollama embeddings has 768 dimensions for nomic-embed-text model and 4096 dimensions for mistral model. We use the default embedding of Chroma. Citing from [16], "[b]y default, Chroma uses the Sentence Transformers all-MiniLM-L6-v2 model to create embeddings. This embedding model can create sentence and document embeddings that can be used for a wide variety of tasks. This embedding function runs locally on your machine, and may require you download the model files". The all-MiniLM-L6-v2 generates vector of 384 dimensions.

The distance between two embeddings represents the similarity between the two pieces of information. The most common distance function is cosine similarity, which uses the cosine value to determine the similarity. A smaller value means increased similarity between two vectors. There are several metrics used to measure the distance of two embeddings. For example, Chroma currently supports three measurements: cosine, Euclidean (L2) and Inner Product. The default distance function is L2. Both cosine and L2 are good for text similarity, but because L2 is more sensitive to noise, we choose cosine in this report. For ChromaDB distance functions, see [17].

Figure 2 shows the query request and response cycle. The user is prompted to enter a query. This query will be used to call the embedding function to embed into its vector format. This query embedding is then used to retrieve similar documents based on the similarity search function. The number of documents to return in this stage can be pre-determined in our program. In our experiments, we set 10 as the number of documents to return so that we can do other further experiments. After this, we prepare data for the generator. We experiment with varying number of documents and fine-tune to see which is the best. After the data is fed into the generator, the generator uses the user's query and the meaning embedded inside the embeddings to make an inference and generate its output to the user.

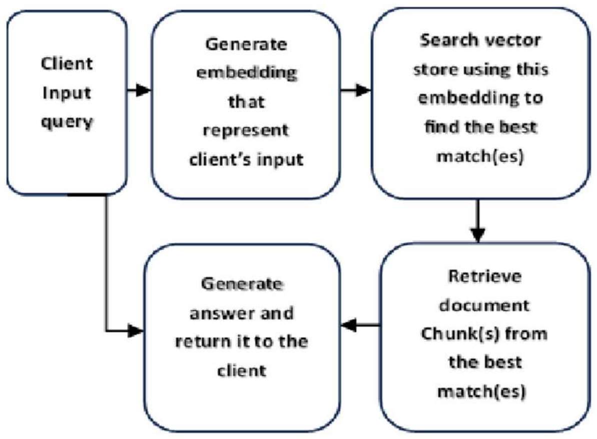

Client Input query
Generate embedding that represent client's input
Search vector store using this embedding to find the best match(es)
Retrieve document Chunk(s) from the best match(es)
Generate answer and return it to the client

Figure 2. Query Request and Response Process

In order for us to generate a response based on the documents found, we used ollama as our inference engine to generate the response. We import ollama module to create an instance of the ollama model to generate query response based on the data from the retrieval process described previously. The pre-trained model we used in this report is llama2 and its total size is 3.8G when downloading its latest version from the Internet. In our experiment, the generator takes the longest time to generate the results in the whole request and response process. Most of the queries required a few tens of seconds on the local computer to generate the response, which was returned to the user without using cache. In order to speed up the whole process, we can improve the first step to load all embeddings to disk storage rather than doing it each time we start the system. The second way to improve the performance is to reduce the number of documents which are stored to the system. But because reducing the number of chunks means increasing the chunk size, that may result in the inclusion of too many texts in one chunk and therefore introduce noise. The other way to improve the performance is to search the similarity faster. This depends on the dimensions of the embedding, which depends on the language model we are using. In this report we are using mini language model, sometimes referred to as small language model (SLM) for embedding.

As for the cache, the main purpose of using it is to store some objects on a faster device (hardware cache) or data structure (software cache) so they can be retrieved quickly when we need them. Here, we introduce a software cache data structure to improve the performance in the generation stage. The common way to organize a cache is to store the most frequently needed objects in the cache store so that each time we need them, we can always find them. We create different levels in the cache to hold various words of users' query. One possibility is to hold the most used term in the top level to search, as it is the term that we will most likely encounter. This method would use a term-based approach to categorize the terms to include in each level based on how frequently the terms appear on all documents in our application, i.e., the terms to be included will be determined by the word counts in our source documents (pdf file, csv, files, etc.).

Instead of using this approach, we implement a different approach in our project. In the current project, we build three levels of cache. The first level consists of key terms such as *dict*, *class*, *tuple*, *string*, *list*, etc. The second level includes terms such as *import*, *def*, etc. The third level consists of Python built-in function names. The argument behind this arrangement is based on our knowledge of organizing a collection of topics, sub-topics, etc. Let's consider how most books or documents or web sites are organized. For example, a book is divided into several chapters. Each chapter represents a particular concept and is itself a sub domain of the book's domain, the book name. One can continue this process and generate a tree structure as the book's table of content. A website has similar organization. A website map usually is a kind of tree structure. Similarly, a company or an organization has the similar hierarchical structure. In order for us to find information from a book, we use keywords to narrow down and find the page number(s) which are related to our question. Hopefully, these page numbers that correspond to our document chunks can give us good matches.

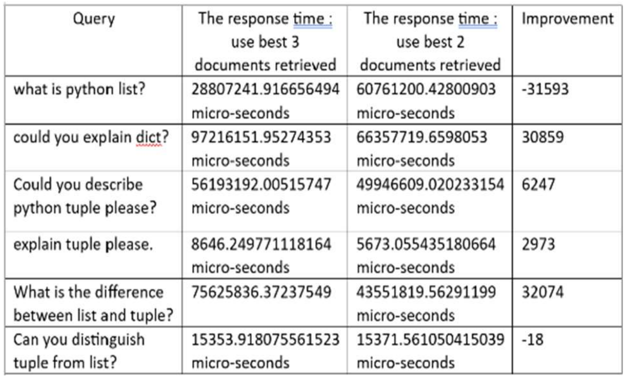
3. EXPERIMENTAL RESULTS

Figure 3 shows comparisons of sample query running times. The first column shows the user queries. The second column shows the response time when the three best documents returned from the embedding similarity search procedure were used. The use of three documents is acceptable since our implementation returns more than three documents, along with their distances and embeddings, from the retrieval process. The third column indicates the response time when two best documents are used as data input to the generator.

Since cache are used, when same or similar queries are presented to the system, and when there is cache hit, the speed up is significant. For example, when questions "What is the difference between list and tuple?" and "Can you distinguish tuple from list?", are asked, it saved 75,610 microseconds. Because our system determined that they are similar questions, it generates a cache hit and the response is readily available to return to the user. According to our experiment experiences, the retrieval time which required similar search is in the millisecond time range, while the generative of answers are in the range of a few tens of seconds. Most of the running time spent in the process from request to response is the generation time. By using cache, we try to avoid that generation process and can significantly save time in our particular application.

By using persistent client of chroma to save the embeddings to local disk storage, as many researchers have also done to save time, it allows users to use the system immediately.

The chunk size will affect the performance in terms of running time and quality of responses. When chunk size is too big, some information may be truncated by our embedding function, since the all-MiniLM-L6-v2 model truncates all input to a maximum of 256 tokens. The chunk size and chunk overlap size require fine tuning for different use cases.



Query	The response time : use best 3 documents retrieved	The response time : use best 2 documents retrieved	Improvement
what is python list?	28807241.916656494 micro-seconds	60761200.42800903 micro-seconds	-31593
could you explain dict?	97216151.95274353 micro-seconds	66357719.6598053 micro-seconds	30859
Could you describe python tuple please?	56193192.00515747 micro-seconds	49946609.020233154 micro-seconds	6247
explain tuple please.	8646.249771118164 micro-seconds	5673.055435180664 micro-seconds	2973
What is the difference between list and tuple?	75625836.37237549	43551819.56291199 micro-seconds	32074
Can you distinguish tuple from list?	15353.918075561523 micro-seconds	15371.561050415039 micro-seconds	-18

Figure 3. Comparison of Response Times

4. CONCLUSION REMARKS

We aim to use a cache data structure to speed up the chat between human user and an AI chatbot on the local machine. Cache was used to store the queries and their corresponding responses. If the system believes there is a good match between the new query and any of the existing queries in the system cache, the system will claim a cache hit and the response is immediately sent back to the user by fetching the response from the cache. If there is a miss, then the normal procedure will be followed by starting with embedding the user query. It will be followed by finding the best matches from the system embedding to retrieve the related documents. The last stage is to use the predetermined number of matched documents and the user query to feed into the generator of language model to generate the response.

Although it is unlikely that a user will repeatedly ask the same question and thus justify using a cache to speed up the response process, it is beneficial if the cache is deployed into an AI proxy server of a private company doesn't want their employees to use public AI, but still wants its employees to gain the advantage of using AI.

5. FUTURE WORK AND IMPROVEMENT

There are various areas where additional investigation and experimentation can be done. How we categorize terms, keywords, and features in our particular application into various categories levels and sub levels to ensure the quality and speed up of responses is one interesting topic worth further pursuit.

There is much more work needed on improving the quality of the responses which are generated by the current system. It depends on the quality of the original pdf file, text files, or csv files, etc. are used, i.e., the sources of information are very important. For example, if we use false information sources, then no matter how good our model is, the wrong information will still be output to users. Assuming all the source files are truthful, we still may get odd or unexpected answers. Finding ways to ensure that the system always generates correct and useful responses is another challenging research topic worth considering.

Another aspect of improving response quality is the chunk size of our document splitting. As mentioned, bigger chunk sizes generate noise and sometimes increasing the size as much as possible may even generate wrong information. So, the question is: for different application how do we quickly fine-tune the chunk size and overlap size to guarantee the best performance in terms of time and quality?

As we know, LLMs are neural networks. Usually, one doesn't want to train the network to be overfit or underfit. The main reason is because we want to ensure that our machine is able to handle unknown input. For example, we can use it to predict the future of the stock market when an unforeseen stock market scenario occurs. If our query to the system is simply to get information such as 'when the homework 1 is due?', or 'what are the new regulations for customer privacy?' etc., the answer would be very straight forward. In Chroma DB, one can use the meta tag to tag the documents in the vector store and when we provide the query, we can use a filter to retrieve the information we need. These kinds of queries, which are related to facts, can be handled by using the cache efficiently. If we can somehow fine-tune the machine in a way that enhances them to remember (like overfitting), then the information in the cache can most likely be used again without any issues.

The other matter we need to resolve is detecting when queries have the same meaning, i.e., the research question is: given two queries, output true if they represent the same question and false otherwise. If the two queries are equivalent, then the answer stored in the cache can be returned to the user immediately. This can also help improve the system's performance.

Lastly, one of the most challenging aspects of is the *validation* of a response. If a user asks the AI system to verify and validate something, can the system do it? For example, if we ask "Is ['a', 123, {'45':789}] a Python list?", the system might answer "no". It will analyze and correctly identify 'a' as a valid string, 123 as an integer, and {'45':789} as a valid dictionary, but incorrectly categorize the whole thing as not a list because it is not mentioned in the source files or the chunk that was selected among the k-nearest neighbors. In an online system, the same query returns a "yes", but modifying the query to "Is ['a', 123, {'45':789] a Python list?" will also result in the system answering "yes", even though we can easily observe the syntax error. Thus, it is worth considering how to build a validation AI agent.

6. REFERENCES

[1] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding" arXiv preprint arXiv:1810.04805.

[2] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J.,

Winter, C., Radford, A. (2020). "Language Models are Few-Shot Learners". arXiv preprint arXiv:2005.14165.

[3] Sagina Athikkal and John Jenq, "An Implementation of Voice Assistant for Hospitality", Signal & Image Processing: An International Journal (SIPIJ) Vol.13, No.2/3/4, August 2022

[4] Yao-Chin Wang, Oscar Hengxuan Chi, Hiroaki Saito, Yue (Darcy) Lu, "Conversational AI chatbots as counselors for hospitality employees", International Journal of Hospitality Management 122 (2024) 103861

[5] Janakiram MSV, "A brief guide to LangChain for software developers", Aug. 28, 2023, https://www.infoworld.com/article/3705097/a-brief-guide-to-langchain-for-software-developers.html,

[6] Peihua Ma, Shawn Tsai, Yiyang He , Xiaoxue Jia , Dongyang Zhen, Ning Yu, Qin Wang, Jaspreet K.C. Ahuja, Cheng-I Wei, "Large language models in food science: Innovations, applications, and future", **Trends in Food Science & Technology**, Volume 148, June 2024 104488

[7] J Herman Aguinis, Jose R. Beltran, and Amando Cope, "How to use generative AI as a human resource management assistant", **Organizational Dynamics**, ORGDYN 53 (2024) 101029

[8] Saika Wong, Chunmo Zheng, Xing Su, Yinqiu Tang, "Construction contract risk identification based on knowledge augmented language models", **Computers in Industry** 157-158 (2024) 104082

[9] Robert Olszewski, Klaudia Watros, Małgorzata Manczak, Jakub Owoc, Krzysztof Jeziorski, Jakub Brzezinski, "Assessing the response quality and readability of chatbots in cardiovascular health, oncology, and psoriasis: A comparative study", International Journal of Medical Informatics, 190 (2024) 105562

[10] Mohammad Alkhalaf, Ping Yu, Mengyang Yin, Chao Deng, "Applying generative AI with retrieval augmented generation to summarize and extract key clinical information from electronic health records", Journal of Biomedical Informatics, 156 (2024) 104662

[11] Steven N. Hart, Noah G. Hoffman, Peter Gershkovich, Chancey Christenson, David S.M Clintock, Lauren J. Miller, Ronald Jackups, Vahid Azimi, Nicholas Spies, Victor Brodsky, "Organizational preparedness for the use of large language models in pathology informatics", Journal of Pathology Informatics, 14 (2023) 100338

[12] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, Douwe Kiela, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks", https://arxiv.org/pdf/2005.11401

[13] Nvidia, "What is retrieval augmented generation", https://blogs.nvidia.com/blog/what-is-retrieval-augmented-generation/#:~:text=Patrick%20Lewis,%20lead%20author

[14] A. Vaswani et al., “Attention is all you need,” arXiv.org, Jun. 12, 2017. https://arxiv.org/abs/1706.03762

[15] Pavan Sai Bolliboina and John Jenq, "Performance Comparisons of Private AI Chatbot and Public AI Chatbot", Proceedings World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4) July 2024

[16] Chroma Embeddings, retrieved Aug. 7, 2024, https://docs.trychroma.com/guides/embeddings

[17] ChromaDB distance functions, retrieved Aug. 8, 2024, https://cookbook.chromadb.dev/core/concepts/#embedding-function