Incorporate a Large Language Model to assist users

Mentors: Alex Richards, Mark Smith, Ulrik Egede

Contributor: Sidharth Sinhasane

Personal Information:

Name: Sidharth Sinhasane
Email: sidharth.sinhasane@gmail.com
Date of Birth: 26/08/2004
Github Id: sidharth-sinhasane
Contact number: +91 9322270517
Country: India
Time Zone: UTC+05:30 (Asia/Kolkata)
University: Savitribai Phule Pune University
LinkedIn : sidharth-sinhasane

About Me

I'm Sidharth Sinhasane, a 3rd-year Computer Science student with experience in the MERN stack, Django. I also worked on several projects involving LLMs, ML, and generative AI. I am excited to be a part of the community and work hard to make this opportunity a success.

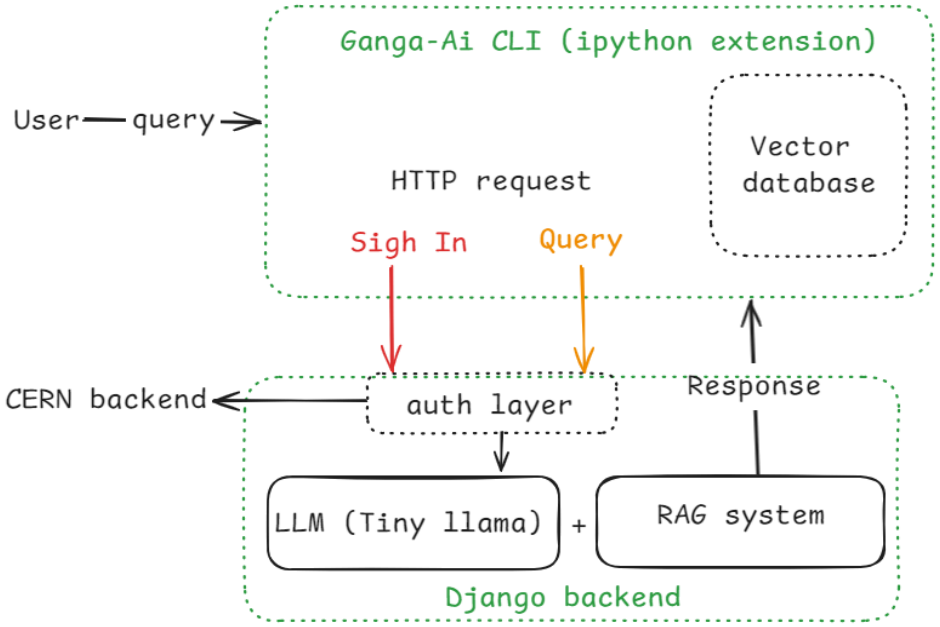
Abstract

This project integrates a Large Language Model (LLM) into Ganga's CLI to assist users with syntax and error handling through natural language input. It incorporates a Retrieval-Augmented Generation (RAG) system for context-aware responses and enhances usability with JWT-based authentication, chat context management, and LLM response streaming. By automating assistance and improving accessibility, this integration simplifies workflows for both new and experienced users.

Project Details

Overview:

To address users' syntax difficulties, this project aims to integrate a Large Language Model (LLM) into Ganga's command prompt. This integration would allow users to describe their goals in natural language and receive usable examples while also intercepting exceptions to provide explanations and solutions. The project, which already has an ollama-based interface for building a Retrieval Augmented Generation (RAG) system with Ganga-specific information, involves several tasks: integrating the LLM and RAG into Ganga, incorporating CLI context from past inputs and outputs, and developing continuous integration tests.



Goals:

1 Implement a JWT-based authentication layer within the Ganga-AI backend, integrating Single Sign-On (SSO) with CERN servers.

2 Develop a sign-in functionality at cli and sign-in endpoint at server side.

3 Write HTTP request handlers to enable CLI interactions with the backend.

4 Integrate a chat context management system in Ganga-AI to maintain ongoing chat history.

5 Rag data enhancement: currently, we are providing the whole documentation.

6 Develop continuous integration tests that ensure that LLM integration will keep working.

7 Create comprehensive project documentation.
Stretched goal: Enable streaming of LLM-generated responses.

Implementation details:

1 Authentication layer:

Sign-up mechanism (Inspired by this example link)

1. Device authorization will be initiated by sending a request to the /auth/device endpoint with my client_id, a state parameter for CSRF protection.
2. We will display the verification_uri and user_code to the user, with instructions to complete verification on another device/browser.
3. We will set up a polling mechanism that checks the /token endpoint every few seconds with user's client_id, device_code, and code_verifier until the user successfully logs in.
4. Once authentication is complete, the server will retrieve and return the JWT token that can be used for future API requests.

Authentication middleware:

Sample code :

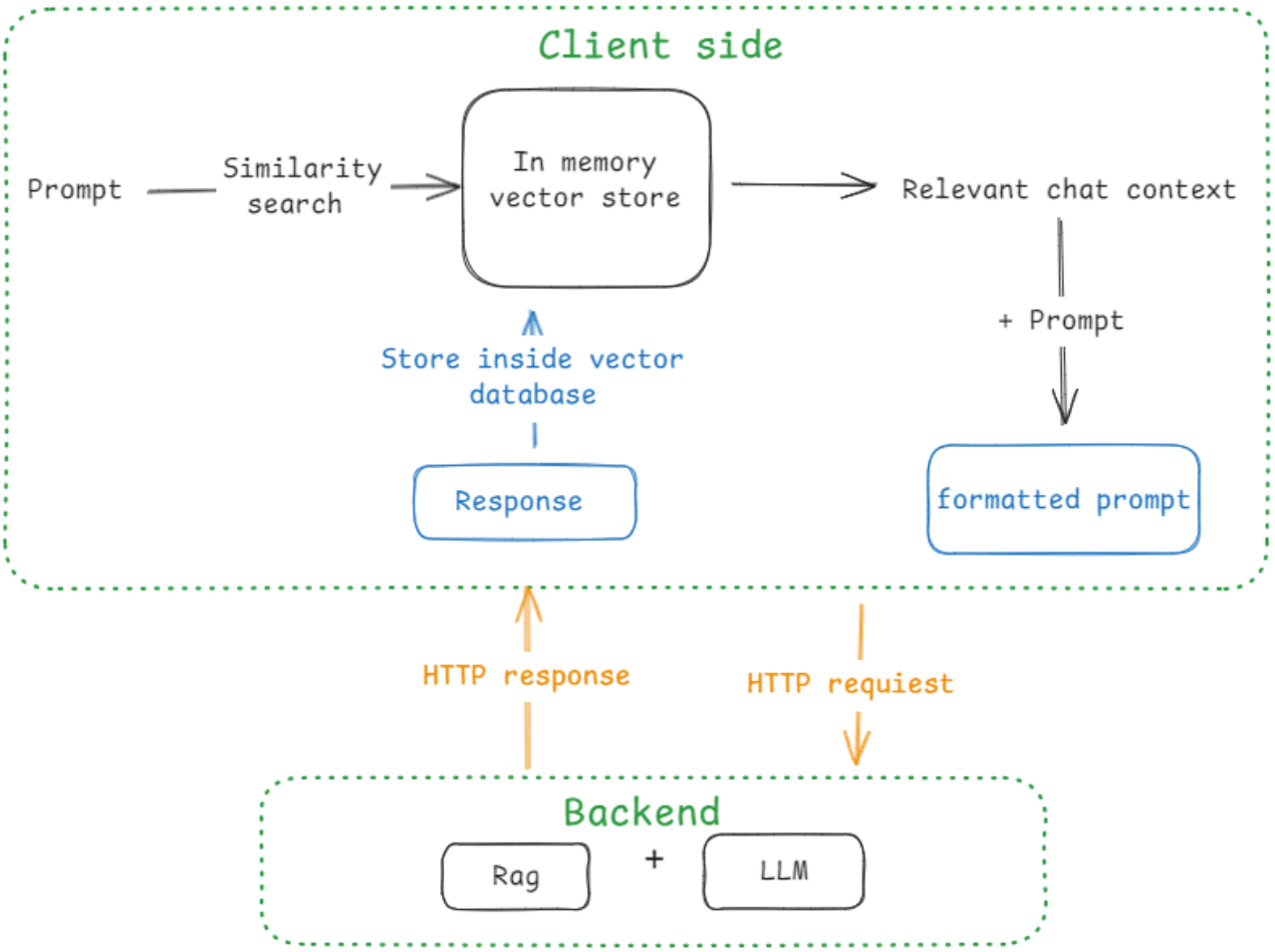
```
class JWTAuthenticationMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        auth_header = request.headers.get('Authorization')

        try:
            token = auth_header.split(' ')[1]
            decoded=jwt.decode(token,setings.JWT_SECRET,algorithm=["HS256"])
            request.user = decoded
        except (IndexError, jwt.ExpiredSignatureError, jwt.InvalidTokenError):
            return JsonResponse({'message': 'Invalid or expired token.'},
status=401)

        response = self.get_response(request)
        return response
```

2 Chat context management system



- A vector database is created in memory for each user on the client side using **langchain ollama embeddings**, and a similarity search is performed over the database to retrieve only the relevant chat context based on the query.

- **Advantages of this approach:**
  There is no need to maintain a backend database solely for storing chat history. Additionally, session management is handled automatically (maintaining context for multiple users at a time) as context is sent through an http request.

- **Challenge**:
  We need to take care of the token size of LLM.

- **To tackle this** :
  We will be selecting top n results from similarity search based on token limit.

- **Dependencies required**:
  Langchain MemoryVectorStore
  Langchain OllamaEmbeddings
  Langchain ChatPromptTemplate

Sample code:

Initialise vector store:

```
def initialiseVectorStore():
    embeddings = OllamaEmbeddings(
            base_url='http://localhost:11434',
            model='nomic-embed-text' # general purpose embedding model
    )
    vector_store = MemoryVectorStore.from_documents([], embeddings)
    return vector_store
```

Store Conversations:

```
def store_conversation(self, user_prompt, llm_response):
        prompt_doc = Document(
            page_content=user_prompt,
            metadata={
                "type": "prompt",
                "timestamp": datetime.utcnow().isoformat()
            }
        )


        response_doc = Document(
            page_content=llm_response,
            metadata={
                "type": "response",
                "timestamp": datetime.utcnow().isoformat()
            }
        )
        self.vector_store.add_documents([prompt_doc, response_doc])
```

Query vector database:

```
def query_vector(self, query, top_k):
        if not self.vector_store:
            return []

        results = self.vector_store.similarity_search(query, top_k)
        return results
```

3 LLM response streaming:

- Streaming of response means getting LLM output in chunks. This feature will definitely improve the user experience of our project.
- For that, I need to make changes to the query_vector_store function at the backend.
- After properly configuring both the LLM and the query engine and making the **streaming flag True**, calling the query will return a StreamingResponse object. Reference
- Client-side handling of streamed response:
  Using the rich Python library, I will rewrite this function _display_formatted_output currently uses the console.print function of rich; I will be using Live class.

Sample code:

```
from rich.live import Live
from rich.console import Console
Console = Console()

def Display_formated_response(StreamedResponse):
    with Live(console=console, refresh_per_second=4) as live:
        for data in StreamedResponse:
        live.update(data)
```

Note: Ipython is not allowing buffer updation, Need to figure out the way to surpass it.

4 Integration:

Server Side

Endpoint	Input	Delivers	status	Description
rag/query	user prompt + chat context	LLM response	ReWrite needed Current implementation	Rag system build, reWrite needed to stream data
rag/signup	Client id	JWT	To be implemented	Need to implement Polling mechanism, Middleware, interaction with CERN backend

Client side:

- Inside magicfunction.py handle_input function will be invoked for user's input.
- The Handle_input function needs to be implemented inside the terminal class, and this function will send a request to our backend.
- So, two HTTP requests for sign-up and query endpoints should be implemented.

Sample code:

1 RequestBackend:

```
import requests
import json

def RequestLLM(query):
        response = request.post(
            url=url,
            json={"query": query},
            headers={"authentication": JWT_TOKEN}
        )
        return response.json()
```

2 Handle Input Functions:

```
from rich.console import Console
from IPython.core.getipython import get_ipython

class Terminal:
    def __init__(self, prompt):
        self.console = Console()
        self.ipython_install = get_ipython()

    def handle_input(self, user_input):
        if user_input.strip().lower() == "exit":
            print("Exiting terminal...")
            return

        self.current_input = user_input
        response = RequestLLM(current_input)
        if response.status_code != 200:
            print("Error:", response.text)
            return
        display_formated_response(response)
```

3 magicfunction.py

```
def assist(line,cell):
      user_input = senitize(line,cell)
      termina.handle_input(user_input)
```

5 RAG data enhancements:

- Here, I am planning to build a system that can update the current data inside RAG's Vector database.
- This provides a way to test which samples are most useful for adding to the RAG.
- Aiming to also include question-answer discussions from the mattermost channel. Reference: [script to extract messages from mattermost channel]
- Also, we need to clean the messages before adding them to RAG.
- Format of message extracted into a JSON file

```
{
   "id": 1,
   "created": "2025-04-07T13:00:00Z",
   "username": "alice",
   "message": "```python\nprint('Hello')\n```"
 }
```

- We need to run a general-purpose LLM locally. Engineering a good prompt to get data that can be added to RAG is going to be a crucial step here.

Sample code:

```
def add_text_data(self, text_directory):
    # Read documents
    new_documents = SimpleDirectoryReader(
        input_dir=text_directory,
        file_extname=".txt"
    ).load_data()
    # add documents to vector store
    for doc in new_documents:
         self.vector_store.insert(doc)

self.vector_store.storage_context.persist(persist_dir=self.vector_store_path)
```

- This function takes a path to the director of text files and adds it to the vector store.
- I need to implement it inside vectore_store class functions.

Timeline

Period	Objective
Pre Work Program	1 Work on issues in Ganga. 2 Do analysis of MCP based Ganga AI approach. 3 Community Bonding Period. 4 Project setup.
Week 1 [2 June - 8 June]	1 Get access from CERN to their database. 2 Start working on the sign-up endpoint at the backend. 3 Remove Ollama installations from the front end.
Week 2 [9 June - 15 June]	1 Complete sign up endpoint. 2 Document the schema. 3 Start working on signup HTTP requests from CLI.
midterm evaluations 1st milestone	
Week 3 [16 June - 22 June]	1 Complete sign up request . 2 Test backend authentication. 3 Start working on auth middleware for generating endpoints.
Week 4 [23 June - 29 June]	1 Complete middleware. 2 Complete and test authentication functionality .
Week 5 [30 June - 6 July]	1 Start context management work. 2 implement a vector store database.
Week 6 [7 July - 13 July]	1 Implement history context retrieval. 2 Write HTTP request to access backend-generated endpoint. 3 Write a text formatter to format the prompt.
Week 7 [14 July - 20 July]	1 Test context management system. 2 Handle the token length limit problem.
Week 8 [21i July - 27 July]	1 Explore CERN docs and host our backend at the CERN server. 2 Develop continuous integration tests.
2nd milstore **(Auth + context management completion)**	
Week 9 [28 July - 3 August]	1 Data collection for RAG to get better results. 2 Work with tokenizers and retrieval chains for better results.
Week 10 [4 August - 10 August]	1 Working on streaming LLM results. 2 Modify the handler at front end to display streamed results.
Week 11 [11 August - 17 August]	1 Add comments and improve code readability 2 Continue working on any pending tasks.

	3 Write detailed documentation.
Week 12 [18 August - 24 August]	1 Work on the final GSOC report. 2 Continue working on any pending tasks. 3 Thoroughly test all features for bugs.
25 August - 1 September Submission week	

Experience:

Backend Developer Intern at I2i Specialist pvt limited. Jan 2025 - March 2025

- Leveraged the MERN stack and TypeScript to build robust backend solutions.
- Developed and optimized REST APIs to enhance system performance and scalability.
- Integrated WebSocket functionalities into the backend, enabling efficient real-time communication.

Related Work:

Supplier Compliance Monitor and Insights Dashboard

live link GitHub

- Developed a full-stack monitoring system to track compliance with contract terms (e.g., delivery times, quality standards). And supplier compliance record as input.
- Integrated the Llama 3 (8B) model locally with CUDA for pattern detection inside the data and provided improvement suggestions.
- Preprocessing of data in the csv file is done and leveraged as a RAG system.
- Built a FastAPI backend with PostgreSQL to store supplier-related data and historical compliance records.
- Developed a React.js front-end using Tailwind CSS, offering an intuitive dashboard to manage multiple suppliers, their reports and view AI-generated insights.
- Tech stack: FastAPI, PostgreSQL, React.js, Tailwind CSS, Llama 3 (8B), CUDA

Evaluation Task

Github

1 Ganga Initial task:

- Worked with merges and splitters in ganga.Got familiar with the usage of ganga. And successfully completed this task.

2 Interfacing Ganga:

- Created a Django backend that hosts an LLM.

- The backend responds with Ganga code when a query is asked.

Contributions

PR	Description	Status
#2411	Solving the issue: Job outputfiles object can not be extended	Open

Other commitments during GSoC

My university exams are scheduled for mid-May. Apart from that, I have no other commitments. Even after my summer break, I will still be able to dedicate 30 hours per week to the project.

Post GSoC:

I am eager to continue contributing to this project and remain an active part of the community. Additionally, I want to further explore Ganga and deepen my understanding of its capabilities.

Thank you!