

Michael Hartmann, Nathaniel Jewel, Marina Rosenwald, Shaun Stangler
CSS 583: Knowledge Management Systems
Group Project Final Writeup
Due: 3/10/2024

Prototype Group Final Project Writeup

Prototype Group Final Project Writeup	1
Introduction	1
Project Status	2
What is working	2
What is not working	2
Methodology	3
Application Design	3
Frontend Design and Implementation	3
API Service Design and Implementation	6
LLM Service Design and Implementation	7
Confluence Application Design and Implementation	8
Integration	9
Hosting	10
Build Instructions	11
Conclusion	12
Team Contributions	12
Glossary	13
References	13

Introduction

Our project is a custom dictionary web-application with microservices and LLM integration that can be utilized by any organization. The application allows for a managed dictionary experience. The web-application has a React JS frontend that allows the user to search, add terms to the dictionary, and ask/receive definition suggestions from an LLM assistant. The custom UI is intricately linked to backend storage capabilities, supported by custom API calls, and utilizes a MongoDB database hosted on Azure. The LLM assistant is implemented using Python and is hosted in Azure. In addition, the web-application has some integration in Confluence.

This project was inspired by a mixture of Wiki and Urban Dictionary. Urban Dictionary is an online dictionary that utilizes crowdsourcing to define and explain different slang words or phrases [1]. This project uses the concept of crowdsourcing definitions from Urban Dictionary to help create a customizable dictionary an organization can access and add to at any time. The

product will replace custom in-house legacy company dictionary sites for organizations and create a dictionary knowledge management within the KMS system. Our product creates a knowledge management tool that helps increase knowledge sharing through a common vocabulary.

Project Status

What is working

The frontend runs with the start command and all elements are tied to corresponding API calls. The API service is capable of accepting API calls to create definitions, get all definitions, get a definition by id, get a definition by word, update a definition by id, and delete a definition by id in the MongoDB. It can also make calls to the LLM-server to execute prompts, create a definition, and get keywords from prompts. It creates MongoDB connections on startup and ends MongoDB connections on shutdown as well as having CORS support.

The LLM service is capable of running inference for any prompt as well as explicitly creating a definition for a given word and attempting to extract keyword from a given text. There is an API layer so that it can execute all three via API calls. The LLM-service can have the inference model easily swapped to any llama-cpp supported model with a single environment variable change.

For both the API service and LLM service there is API documentation and testing via swagger. Both services have key static variables as environment variables stored in a .env file allowing for easy changes of host, port, LLM model / location, routes, mongodb connections, and LLM parameters. Each service is a stand alone micro-service and can be run independently.

For Confluence WFE hosting, a second WFE version was built that enables the static WFE to load, however backend query functionality is not working due to security issues with Confluence/AWS. The full WFE functionality works properly outside of Confluence using either the Forge or native React version.

What is not working

There is no queue/semaphore/load-balancer so the LLM-service can't handle multiple incoming requests at the same time. If the LLM-service receives a new inference request while one is already running it will throw a segmentation fault because the llama-cpp-python library is executing the inference in C/C++.

As stated above, CRUD operations to the backend API server are not working within the Confluence host due to host side blocking outgoing cross-domain JS REST requests. Opening the application to cross domain requests within production Confluence environments would require significant development time and additional support requests to Confluence to open security controls.

Methodology

Application Design

The keyword definition application is a web service that uses microservices architecture. Each section of the application runs under its own process and is designed to be hosted in a separate cloud service/server. The web front end (WFE), API service, and LLM service all communicate through REST web calls using serialized JSON data. There are no backend components to the WFE microservice, which allows for modularized hosting options with Confluence as a Forge application. Figure 1 shows the service architecture of the application.

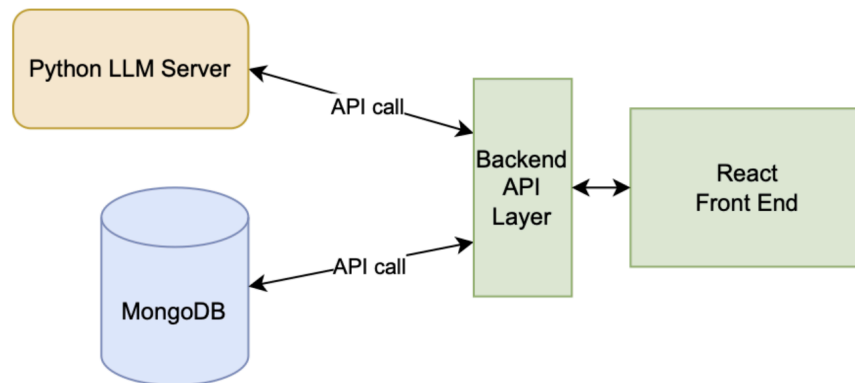


Figure 1 Component Architecture using Microservices

Frontend Design and Implementation

The frontend was implemented using JavaScript (JS) with the React library and Cascading Style Sheets (CSS). The design was created with a primary focus on optimizing user experience through enhanced ease of use and simplicity. It can be split up into three sections, the vocabulary list on the left hand panel, the search bar and results in the middle column and the add form with the large language model suggestion on the right side panel. Figure 2 below is an image of our working frontend demonstrating the three highlighted columns.

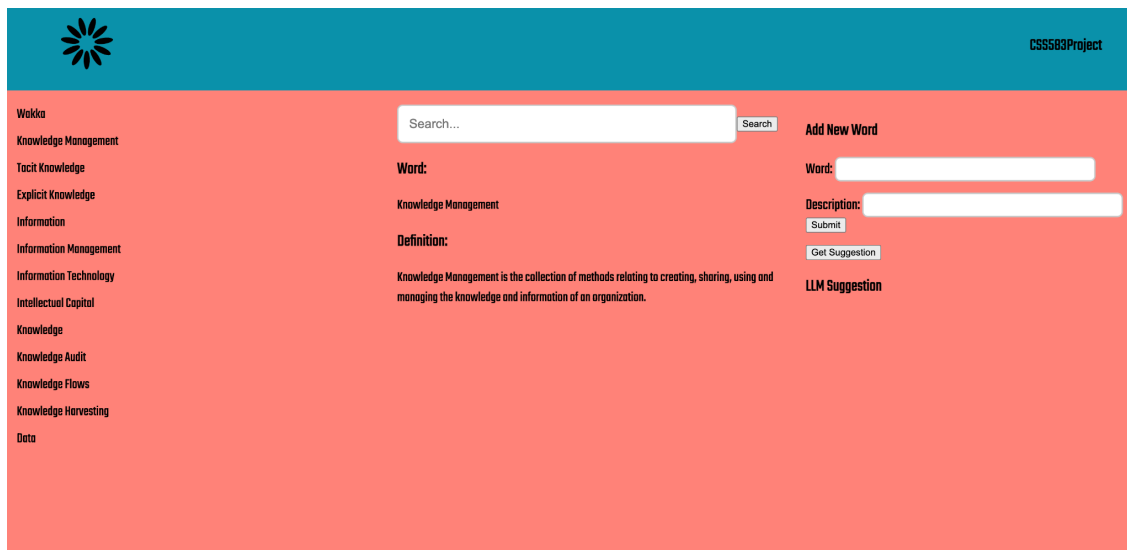


Figure 2 Working website frontend

The vocabulary list on the left hand side utilizes an API call to receive a list of all words and maps them to the stylized column seen in Figure 3. Each word is assigned with a click function that activates on selection, setting the middle panel to display the chosen word.

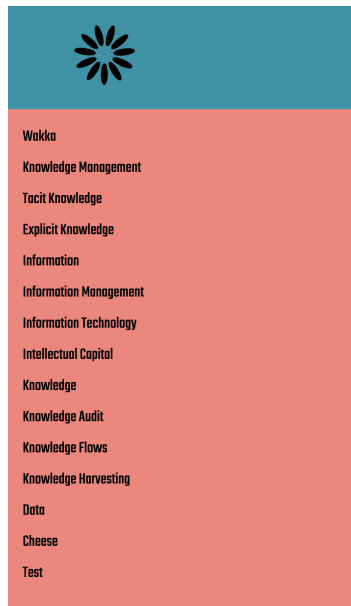
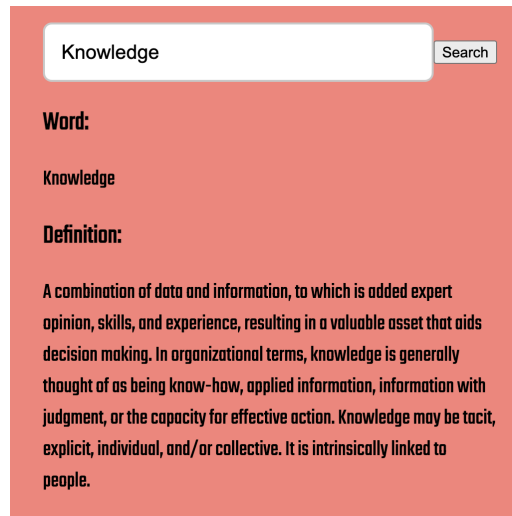


Figure 3 Expanded left side panel

The middle panel consists of two main components: the search bar and the selected display. The search bar facilitates user input, taking in text and assigning it to a search variable. Adjacent to the search bar, a button labeled “Search” triggers a function on click. This function initiates an API call, querying the database for the search variable. If the search word is found, the selected display element of the main panel will show the word and its corresponding information.

The selected display component of the middle panel will display the word and definition of a selected variable via a reference to the corresponding database elements. The selection of a word on the left-hand panel assigns the selected variable to the word’s corresponding database object. Similarly, the search button assigns the selected variable to the database object assigned to the user-inputted word, given it exists in the database. This dynamic interaction enhances the user’s ability to utilize the database and find words making knowledge sharing easier within the company. Figure 4 shows the middle panel after a successful search was completed.



The screenshot shows a web application interface with a red background. At the top, there is a search bar containing the text "Knowledge" and a "Search" button to its right. Below the search bar, the word "Knowledge" is displayed under the heading "Word:". Under the heading "Definition:", there is a paragraph of text: "A combination of data and information, to which is added expert opinion, skills, and experience, resulting in a valuable asset that aids decision making. In organizational terms, knowledge is generally thought of as being know-how, applied information, information with judgment, or the capacity for effective action. Knowledge may be tacit, explicit, individual, and/or collective. It is intrinsically linked to people."

Figure 4 Middle panel with search results shown

The right side panel can be broken down into two main components, both highlighted below in Figure 5: the “Add New Word” component and the “LLM Suggestion” component. These components allow for the expansion of the database thus contributing to knowledge growth within the organization through the product.

The “Add New Word” component consists of two prompts and response boxes for user text input. The first prompt asks the user to input a word they would like to define and assigns the user’s input to a word variable, while the following prompt gather’s the corresponding description from the user and assigns it to a description variable. Below the two prompts is a button labeled “Submit”, when selected the button will call a function that utilizes an API Post call sending the word variable and the definition variable to the database. The function will also utilize an API call to update the left side panel adding the submitted word.

The “LLM Suggestion” component serves as a tool for the user to reference a Large Language Model (LLM) for generating the description of their entered words. After the user enters their word into the corresponding word prompt and response box highlighted in the above paragraph, clicking the “Get Suggestion” button triggers an API call requesting the LLM to provide a description for the word variable. The response from the LLM is then stored in a response variable. The text section below the button displays the text response received from the LLM.

Add New Word

Word:

Description:

LLM Suggestion

Suite A: The term "test suite" refers to a set of tests that are designed to be run in sequence to ensure that a product, system or process meets its intended specifications. It typically includes multiple test cases with different inputs and scenarios designed to cover all possible use cases and edge cases. The goal of a test suite is to identify any failures or errors in the system under testing so they can be addressed and corrected before deployment.

Figure 5 Right side panel with the “Add New Word and “LLM Suggestion” components

API Service Design and Implementation

Core Libraries: pydantic, fastAPI, uvicorn, httpx, and pymongo

Core Files: main.py, models.py, dict_routes.py, llm_routes.py.

The API service was written in python. The API service is started by running the main.py file. On startup the main.py will fetch key sensitive environment variables from the .env file. Main.py uses host, port, routing paths, and mongodb connection information. Main.py uses fastAPI, a modern web framework for building RESTful APIs in python. FastAPI is robust and allows for asynchronous capabilities, high performance, and data validation. In main.py, fastAPI is used to asynchronously make our mongodb connection and connect to the mongodb data. The mongodb connection is down with pymongo, a mongodb driver for python, and we automatically close the connection on shutdown. Next in the main.py we set our CORS access and security as well as define our RESTful API routes. Finally, we use Uvicorn, an asynchronous server gateway interface (ASGI), to start our web service.

In the models.py file we use pydantic, a python library for data validation and schema creation. We created three schemas, Definition for the data structure of a definition in our mongodb, DefinitionUpdate, for structure to update a definition, and LLM for the structure of data expected by the LLM service.

Lastly are the dict_routes.py and llm_routes.py that are held in our routes folder. These files are for executing the API calls. The dict_routes.py handles the definitions routes and can create definitions, get all definitions, get a definition by id, get a definition by word, update a definition by id, and delete a definition by id. The llm_routes.py handles the llm routes, which make http calls, using the httpx library, to the LLM service in order to run prompts. The httpx library has a very low default timeout on its API calls, which is why for all API calls to the LLM server we set “timeout=None”, since the LLM can sometimes take a long time for inference. There are routes to make an default llm call, make an llm call to define a word, and make an llm call to get the keywords from text. Each API route has code to handle and return the correct status code, like 200, 201, 404, etc.

REST API Methods supported:

Definitions

- GET: /definition - List all words
- POST: /definition - Create word
- GET: /definition/{id} - Find word by ID
- PUT /definition/{id} - Update word
- DELETE: /definition/{id} - Delete word
- GET: /definition/word/{word} - Get word by name

LLM

- POST: /llm - LLM freeform text query
- POST: /llm/keywords - LLM query to define keywords from post
- POST: /llm/definition - LLM query to define a word

LLM Service Design and Implementation

Core Libraries: llama_cpp_python, pydantic, fastapi, uvicorn

Core Files: main.py, llama_main.py, models.py, routes.py.

Similar to the API service, fastapi is used as the web framework for creating our RESTful APIs, Pydantic is used for schema creation and data validation, and uvicorn is used for the ASGI server. In main.py we use fastAPI to declare our routes and start the ASGI server. In models.py we only need a single pydantic schema, llm, which is how the service is expecting to receive data. In llama_main.py we have our singleton class that creates our LLM_Llama object. This object allows us to make default prompt calls, calls to define a word, and calls to get the keywords from a body of text. The llama_cpp_python library is used in order to run inference on mainly Meta's LLaMa models in C/C++. This library was chosen because it's dependency free, has minimal configuration requirements, allows hybrid GPU/CPU support, and has quantization support.

The last file is the routes.py that is held in our routes folder. This file creates the LLM_Llama object that is used for inference and executes three different API calls for default prompt inference, extracting keywords from a body of text, and creating a definition from a given word. The LLM_Llama object will use the downloaded llama_cpp supported model that is found by the "MODEL_PATH" environment variable. This path is the path to where the downloaded model is. In our final implementation we used a four bit quantized Orca-mini-3b model. Orca-mini-3b is a three billion parameter model for text generation. Quantized models execute their operations on tensors with reduced bit width precision instead of full floating point values. We also did testing with a four bit quantized Llama2-7b-chat model, however we swapped to the smaller orca model for fast inference times with our limited hardware. You can point the "MODEL_PATH" environment variable to any model supported by llama_cpp in order to change LLM model with no additional changes.

There is a folder llm-model containing only a .gitkeep file. This file is for putting your downloaded LLM models, but isn't required.

Rest API Methods Supported:

LLM

- POST: /llm-server - LLM freeform text query
- POST: /llm-server/keywords - LLM query to define keywords from post
- POST: /llm-server/defineterm - LLM query to define a word

Confluence Application Design and Implementation

To host the WFE under Confluence, the WFE microservice would need to be ported over as an Atlassian Forge application. For that, the application needed to be wrapped in a separate node.js application using the Forge schema:

CSS583GroupWFE

- Src
 - Index.js - Contains the Forge application resolver.
- Static
 - CSS583WFE2 - Our keyword definition application
 - Public
 - Index.html - Application index page
 - Src
 - Api - Application API backend function calls
 - Components
 - Pages
 - CSS583Project.js - Core WFE React code
 - app.js
 - Routes for CSS583Project.js
 - index.js
 - Package.json - Application packages
 - Manifest.yml - Forge specific properties for Confluence web page hosting
 - Package.json - Forge specific packages

This application wrapping allows the Forge framework to install and deploy the application to Confluence using simple Forge CLI calls. The resulting architecture changes can be seen in the following figure.

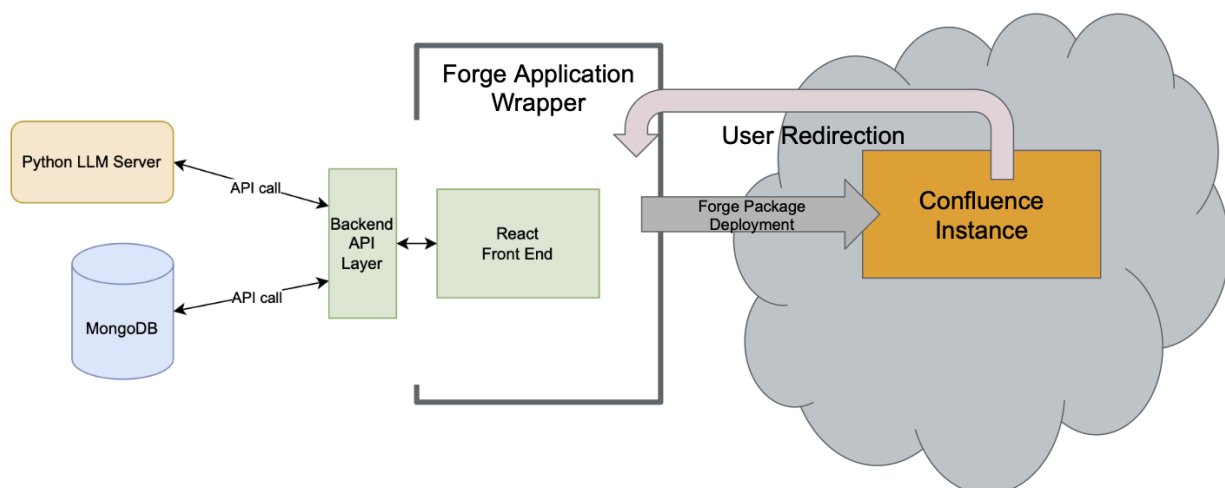


Figure 6 Confluence Integration Architecture

The Confluence hosting required additional security concerns and measures. A separate WFE needed to be used to work around WFE asset requests and relative/absolute linking issues within Confluence's AWS provider. This second WFE could be seen in Confluence within Figure 7.

Confluence

Home

Recent ▾

Spaces ▾

Teams ▾

Apps ▾

Templates

Create

Upgrade

Search

CSS583 Keyword Dictionary

Word List

Definition

Add New Word

Keyword:

Description:

Submit

Get Suggestion

LLM Suggestion

Search

Keyword:

Submit

Figure 7 Confluence Hosting With 2nd WFE

Due to the outbound outgoing calls, more permissions are required to allow for outgoing JavaScript REST calls using cross-domain techniques. These security measures put the full implementation of Confluence out of the scope for this project due to time constraints.

Integration

To integrate all of the microservices into a single application, local testing, debugging, and integration techniques were used before transferring the service components into their hosted environments. Integrating each environment uncovered several issues with build setup, missing requirements, missing functionality, and data schema mismatches.

Extensive testing was utilized with traditional IDE breakpoint debugging, Postman/Swagger request/response troubleshooting, and browser developer console debugging as seen in Figure 8.

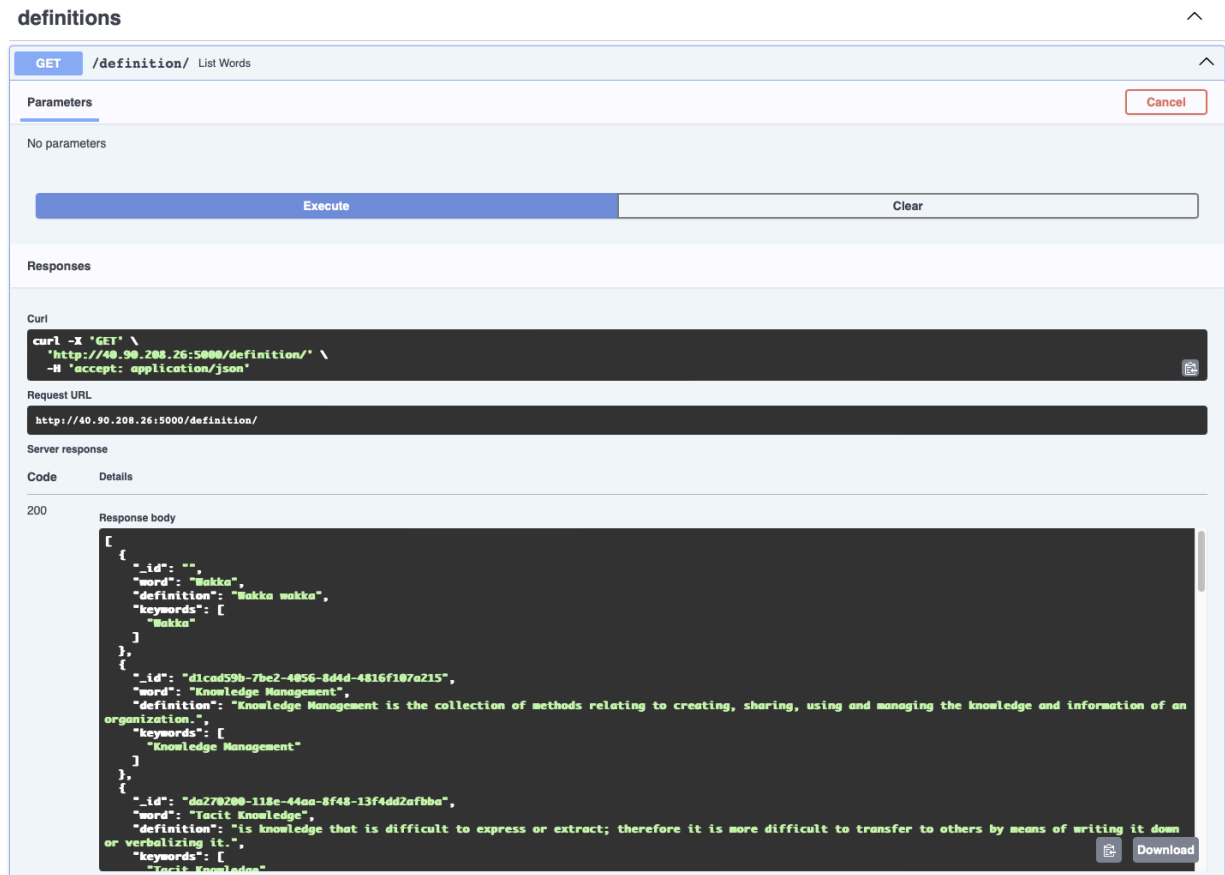


Figure 8 Swagger API Debugging

The following work was performed for remediation items to fix integration issues:

- WFE application rewrite to create a 2nd WFE for integration reference.
- DefineTerm LLM query wrapper to allow for keyword definition suggestions.
- Fixed miscellaneous data format bugs.
- Move to Orca-mini-3b model due to LLM performance issues.
- Removal of keyword suggestion functionality in WFE to account for chaotic LLM responses from Orca-mini-3b.
- Package.json updates to fix missing/broken requirements packages.

Hosting

Once local integration work was completed, the project services were moved to hosted environments. The MongoDB cluster, LLM service, and API backend service were hosted in Azure. Linux VMs were utilized for the LLM service and API backend service to keep deployment simple. Hosting the microservices in a cloud environment not only allowed for a

seamless prototype demonstration, but greatly improved the development experience when troubleshooting the React WFE, as we could see real calls and data packages within the system. Each service was secured and built with separate network security rules to only allow traffic from each of the service's respective upstream endpoints.

However, we also hit a few issues with hosting the LLM service, as all GPU enabled VMs were prohibitively expensive. Simple instances using NVidia GPUs cost up to \$2000 per month. This put utilizing performant VMs out of the scope as we had to rely on 4 core CPU VMs for LLM work. This led to us switching our LLM model to a simpler offering to fit within our overall performance requirements.

Build Instructions

GitHub: https://github.com/marinarosenwald/CSS583_KMS_Project.git

API Service:

Requirements: pip3, python3.9 (other python3 version may work but aren't tested)

Recommended: create a virtual environment using pip3 install venv

Setup:

- Create a .env File for key static variables. An example .env file is in the README

- Run "pip3 install -r requirement.txt" in order to install dependencies

Start: python3 main.py

Testing: "http://<host>:<port>/docs#/" for API documentation and testing

LLM Service:

Requirements: pip3, python3.9

- Your torch, torchaudio, torchvision libraries must have specific versions to match your python version, so if you use a different python3 version, make the respective changes to these three libraries in the requirement.txt file. If using python3.9 no dependency changes are required.

Recommended: create a virtual environment using pip3 install venv

Setup:

- Create a .env File for key static variables. An example .env file is in the README

- Run "pip3 install -r requirement.txt" in order to install dependencies

- go to

`https://huggingface.co/TheBloke/orca_mini_3B-GGML/tree/main`

For orca or

`https://huggingface.co/TheBloke/Llama-2-7B-Chat-GGML/tree/main`

For Llama

Click files and version, and download the model (we used the .ggmlv3.q4_1.bin versions) then set "MODEL_PATH" = "<path>" to point to the location of your downloaded model. If you wish to use a .gguf model then set your llama_cpp_python

version to 0.1.79 or higher. Otherwise if using a .ggml version, make no additional changes.

Start: python3 main.py

Testing: "http://<host>:<port>/docs#/" for API documentation and testing

Frontend:

Requirements: react

Terminal commands following the cloned repo linked at the top of this section:

```
cd Frontend/css583-project-frontend
```

```
npm install
```

```
npm start
```

Note: once the frontend is running, make sure you click on the "CSS583Project" link in the navigation bar to get to the working page for this project

Conclusion

Lessons Learned:

- Most LLM implementation libraries don't support quantized models. However, when working with limited hardware resources, quantized models will help significantly.
- The M1 chip in the Macbook is not supported and will not get full utilization by most LLM libraries.
- Without high-end supported GPUs you won't get anywhere close to advertised inference speeds. It won't be seconds slower, it will be many minutes slower.
- The Llama-cpp-python library is a C/C++ implementation under the hood and trying to run back to back inference without queuing/semaphore will cause a segmentation fault
- You need your API calls to accommodate potentially long inference time when using LLMs
- LLMs are not a good solution to every NLP problem. For example, they can extract keywords from text, but there are better and more simple solutions that can already solve this problem.
- While system and instruction prompts for LLMs can produce responses closer to expected results, they still lack consistency.
- The order of operations in creating a product is extremely important. When creating this application we went in the order of frontend, backend, database, then put everything together. This was not optimal and caused a lot of problems at the end. There were many compatibility issues and functions we believed would work with dummy data turned out to be nonfunctional with the database. Through this process we learned that we should have gone in an order of dependencies, so database then backend, then frontend putting everything together as we went along.

Team Contributions

The team roles were as follows:

- Michael Hartmann - Backend developer (LLM focus)
- Nathaniel Jewel - Backend developer (API focus, LLM, front-end API portion)
- Marina Rosenwald - Frontend developer
- Shaun Stangler - Database administrator, confluence development, additional WFE development work, integration development work, and hosting.

Everyone on the team followed the roles and responsibilities assigned to them (equal contribution). Michael and Nathaniel created the backend, Marina created the frontend, Shaun created the database, put everything together (debugged) and worked with Confluence.

Glossary

- Quantized Models: Quantized models execute their operations on tensors with reduced bit-width precision instead of full floating point values, this makes the model significantly smaller at the cost of some amount of accuracy.
- Urban Dictionary: an online dictionary that utilizes crowdsourcing to define and explain different slang words or phrases [1].
- React: React is a free and open-source front-end JavaScript library for building user interfaces based on components. It is maintained by Meta and a community of individual developers and companies [2].

References

[1] Wikimedia Foundation. (2024, February 3). *Urban dictionary*. Wikipedia.
https://en.wikipedia.org/wiki/Urban_Dictionary

[2] Wikimedia Foundation. (2024, March 7). *React (software)*. Wikipedia.
[https://en.wikipedia.org/wiki/React_\(software\)](https://en.wikipedia.org/wiki/React_(software))