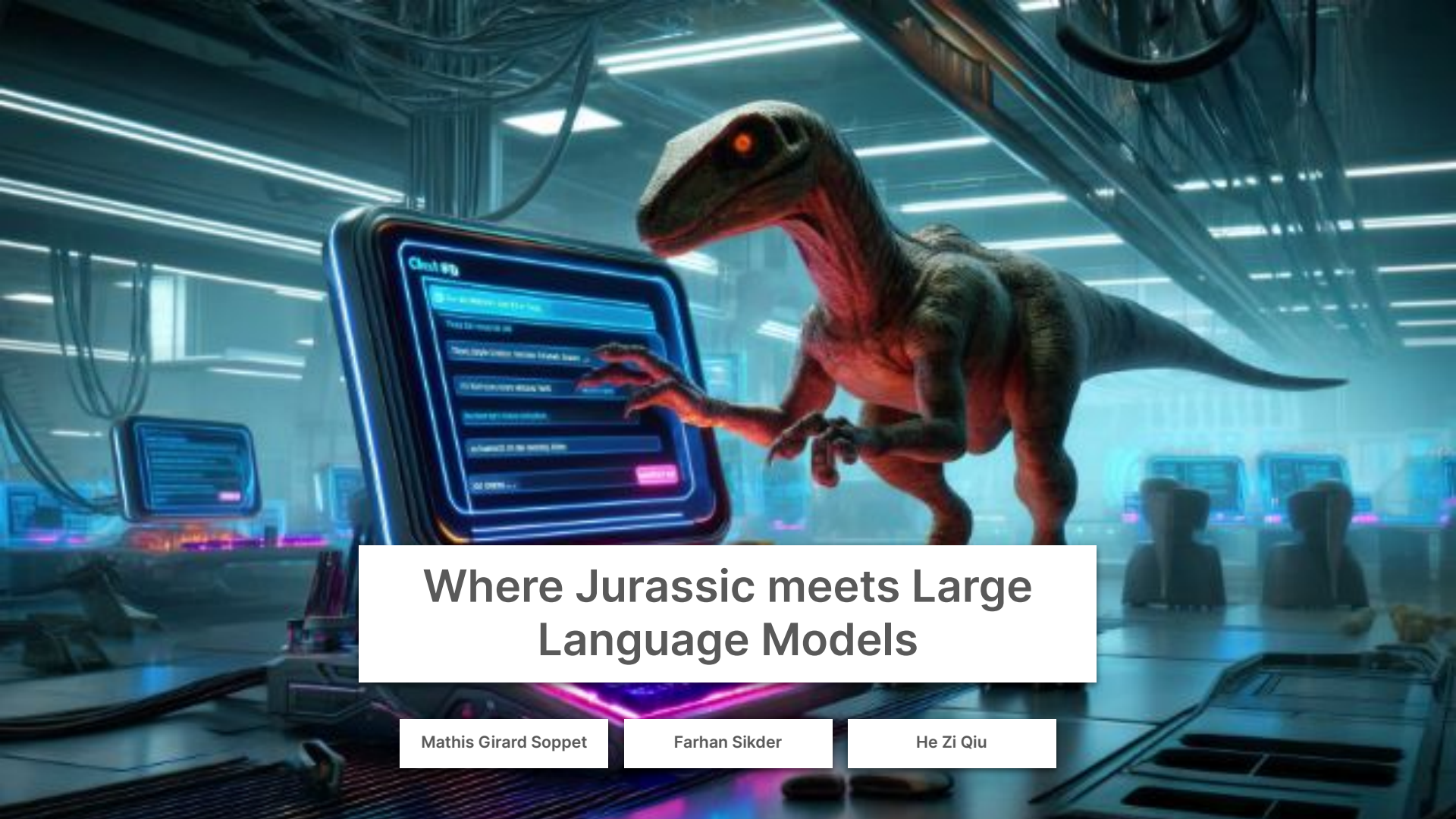


Where Jurassic meets Large Language Models

Mathis Girard Soppet

Farhan Sikder

He Zi Qiu

Overview

Our goal was to craft a GenAI chatbot, a digital assistant armed with a deep knowledge of dinosaurs, sourced directly from detailed PDFs we dug up like true paleontologists of data.

Task 1

Making a GenAI Chatbot. This is where we take off!

Task 2

Adding a knowledge base to our Chatbot

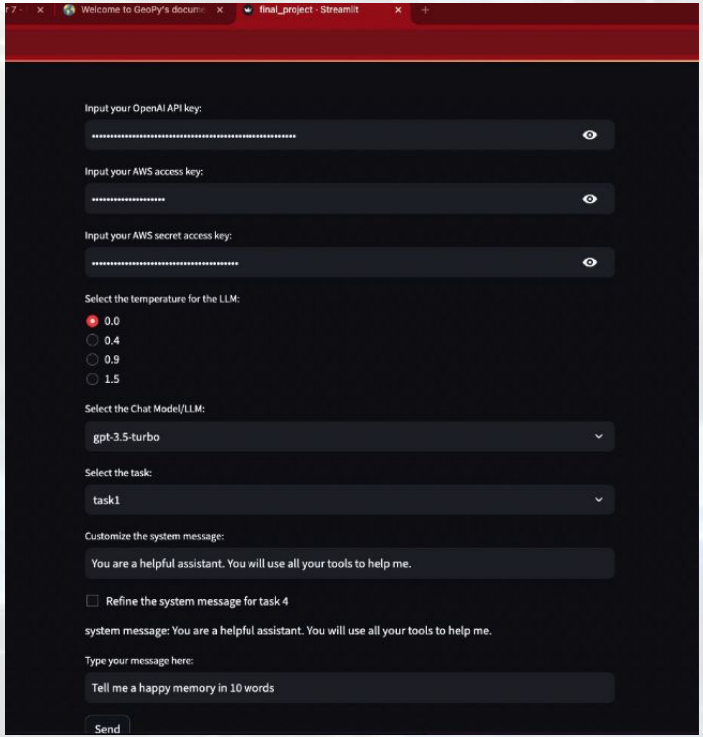
Task 3

Connecting AWS to the chatbot and using query, SMS services

Task 4

Adding it all up! An end-to-end workflow

Task 1 : Create a Streamlit Chatbot



Challenges:

Installation of Libraries: Ensuring all required libraries were correctly installed was a key challenge, especially since the development environment can affect the installation process.

Example: When importing libraries in the terminal, some libraries failed to work in the IDE. It was determined that libraries had to be installed directly within the IDE's terminal, not just the system terminal.

API Key Management: Safeguarding API keys was crucial to maintaining security and integrity within the project.

Example: Implementing environment variables to store API keys helped prevent hard-coding sensitive information into the public codebase.

Observations:

Importance of Documentation: The project underscored the importance of consulting official documentation to understand library functionalities and best practices, rather than solely relying on external tools like ChatGPT for quick answers.

Task 2 : Create A Vector Datastore and use it to retrieve information

Prepare vector store

```
from langchain_community.document_loaders import TextLoader
loader = TextLoader('./all_text.txt')
documents = loader.load()

from langchain_community.vectorstores import FAISS
from langchain_openai import OpenAIEmbeddings
# from langchain.embeddings import HuggingFaceEmbeddings # just use a free model from hugging face, should be enough
from langchain_text_splitters import CharacterTextSplitter

text_splitter = CharacterTextSplitter(
    separator="\n",
    chunk_size=1000,
    chunk_overlap=500,
    length_function=len,
)
texts = text_splitter.split_documents(documents) # print this out to show the text being split
print('number of chunks:', len(texts))

# embeddings = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")
model_for_creating_embeddings = OpenAIEmbeddings(api_key=os.environ.get('OPENAI_API_KEY'))

# initialize a vector database db by applyign the embedding model to the text chuncks
db = FAISS.from_documents(texts, model_for_creating_embeddings)

# create a retriever from the db
retriever = db.as_retriever()

from langchain.tools.retriever import create_retriever_tool

retriever_tool = create_retriever_tool(
    retriever,
    name="retriever",
    description="A retriever tool that to retrieve relevant documents about Dino: T-Rexs and Velociraptors",
)
tools = [retriever_tool]
```

[31]

```
number of chunks: 24
```

Agent

```
from langchain import hub

# prompt = hub.pull("hwchase17/openai-tools-agent")
# prompt.messages

# let's create a chat prompt that allows for system message
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder

# prompt format
system_message = "You are a helpful assistant"
prompt = ChatPromptTemplate.from_messages(
    [
        ("system", system_message),
        MessagesPlaceholder("chat_history", optional=True),
        ("human", "{input}"),
        MessagesPlaceholder("agent_scratchpad"),
    ]
)

from langchain_openai import ChatOpenAI
# create an open ai llm
model_name = 'gpt-3.5-turbo' # 'gpt-4'
temperature = 0 # 0.5, 0.7, 1.5

llm = ChatOpenAI(temperature=temperature,
                 model = model_name,
                 api_key=os.environ.get('OPENAI_API_KEY'))

from langchain.agents import AgentExecutor, create_openai_tools_agent

agent = create_openai_tools_agent(llm, tools, prompt)
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)
```

Task 3 : Create Tools for an AI Agent to execute tasks

AWS Resource Initialization

Description: This part demonstrates the initialization of AWS resources using the Boto3 library. It sets up a connection to Amazon DynamoDB and Amazon Simple Notification Service (SNS), specifying the required AWS region and using environment variables for secure API key management. This setup is crucial for data storage and communication tasks in the application.

DynamoDB Table Management

Description: This part includes functions for managing a DynamoDB table: it can delete, create, and insert data into a table named 'TransportData'. The table creation function sets 'Date' as the primary key and configures the table's read and write capacity. Data insertion uses a batch writer for efficiency, ideal for handling multiple records efficiently.

CSV Data Upload to DynamoDB

Description: This part handles the conversion of CSV data into a list of dictionaries and attempts to create a DynamoDB table. If the table already exists, it captures the exception and proceeds to insert the CSV data into the existing table, demonstrating error handling and data insertion in DynamoDB.

Task 3 : Create Tools for an AI Agent to execute tasks

SQL Database Integration and SMS Notification

Description: This part demonstrates several functionalities:
1. It creates a pandas DataFrame from dictionary data, which is then written to an SQLite database using SQLAlchemy, effectively creating a lookup table for Dino IDs and Names.
2. It utilizes a SQLDatabaseToolkit integrated with a ChatOpenAI model to facilitate querying this SQL database via an LLM.
3. Additionally, it showcases how to use AWS SNS to send a text message, providing the functionality needed for alerts or notifications.

AWS SNS Text Messaging Function

Description: This function defines a method for sending SMS messages using AWS SNS. It initializes an SNS resource, obtains a specific SNS platform application object, and sends a message to a designated phone number with transactional attributes, handling potential errors in the process. This is ideal for ensuring timely communication in applications requiring alerts or notifications.

DynamoDB Query and Weather Data Retrieval Integration

Description: This part includes a function that queries a DynamoDB table to retrieve city and DinoID information based on a specified date. Additionally, it incorporates the OpenMeteo API using the openmeteopy library to fetch historical or current weather data for the retrieved city, integrating geographical and meteorological data retrieval into the application's functionality.

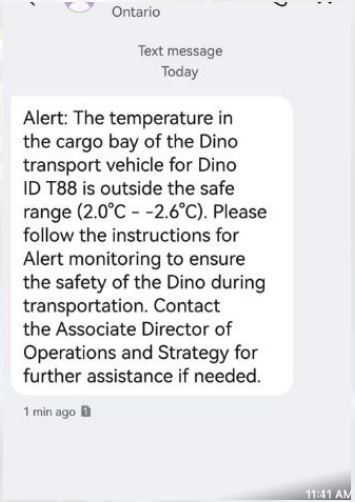
Task 3 : Create Tools for an AI Agent to execute tasks

Historical Weather Data Retrieval Function

Description: This part retrieves the maximum and minimum temperatures for a given city on a specific date. It utilizes geocoding to determine the city's coordinates and then fetches historical weather data using the OpenMeteo API. The function is designed to return the temperature range, ensuring accurate weather data retrieval for any required date.

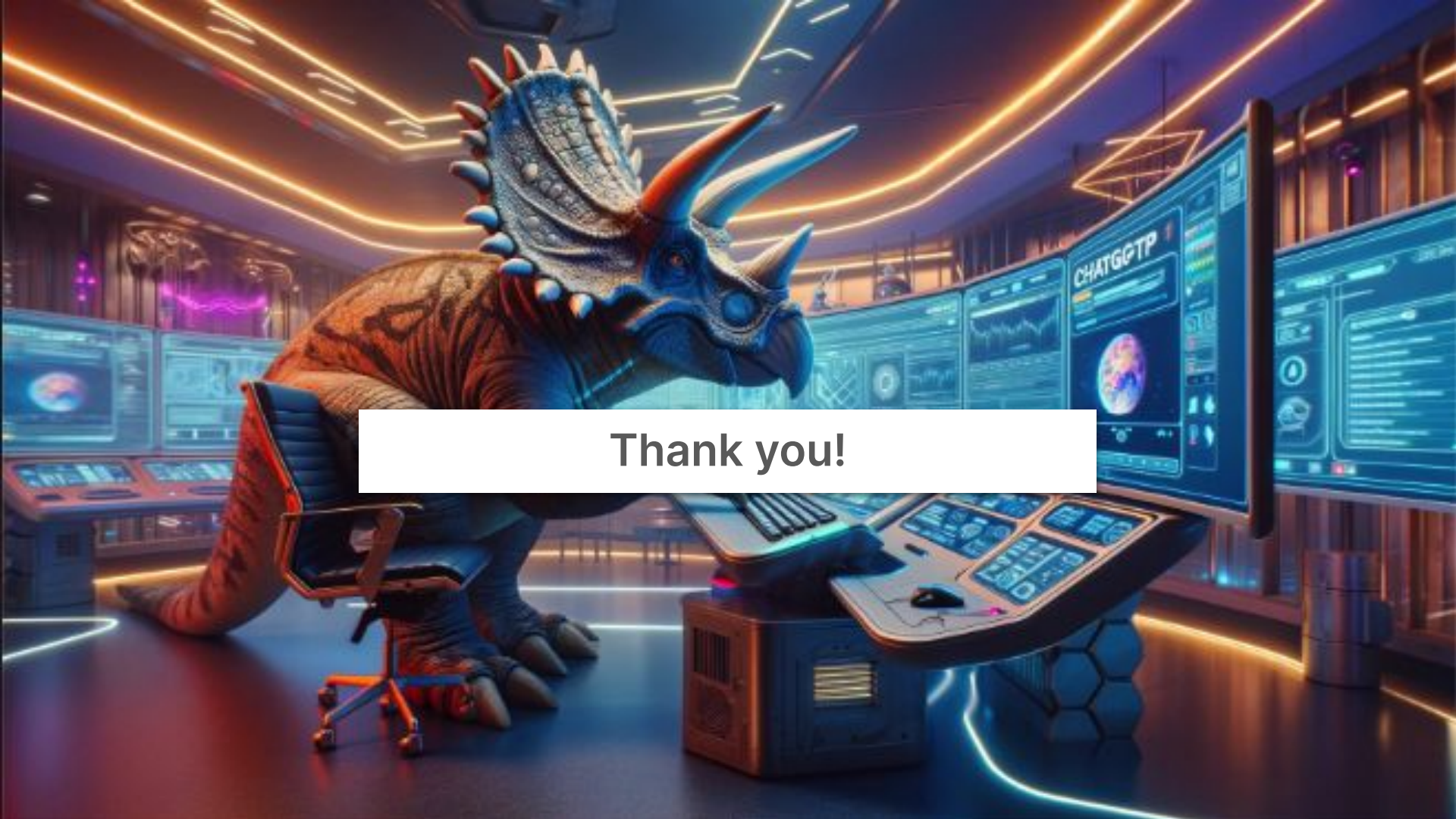
Function Integration and Testing (Dyno ID, city, weather)

Description: This example demonstrates the successful integration and testing of the SMS sending function and data retrieval functions. It showcases sending an SMS via AWS SNS and retrieving data by date from DynamoDB, followed by fetching weather data for a specific city and date. The output confirms the functions' efficacy with details of the SMS sent, dinosaur transport data, and temperature information for Anchorage.



Task 4 : Create a program that executes the end-to-end workflow

OUR APPLICATION AT A GLANCE!



Thank you!