# Build Your Spending Insights Advisor with LLM

**Estimated time needed:** 1.5 hours

## Introduction

Imagine engaging in a casual conversation with your transaction records, effortlessly inquiring about your spending habits and receiving immediate, insightful responses. This scenario is not a futuristic fantasy but a present reality made possible by our innovative project. In this tutorial, we leverage the capabilities of LangChain's SQLChain to seamlessly connect a credit card transaction database with Large Language Models (LLMs). Our goal is to enable intuitive, natural language querying, offering you instant and profound insights into your financial behaviour. Welcome to a new era of financial management where complex queries are transformed into simple, conversational interactions.

🌟 Check out the demo of what you'll build:

Try The Demo App

## What Will You Learn?

Here's what you will learn to:

- Integrate LLMs with SQL databases to handle data queries, enhancing your skills in database management and working with LLMs.
- Enhance the capabilities of LLMs by using the techniques of prompt engineering.
- Transform any database data into understandable and actionable insights.

## What is LangChain?

LangChain is a powerful and open-source Python library designed for natural language processing tasks. It facilitates easy access to a range of potent language models from providers such as OpenAI, Cohere, Huggingface Hub, and IBM watsonx, among others. Beyond mere model access, LangChain offers tools for interactive dialogues with these models, allowing for seamless human-machine conversations. We'll use LangChain to:

- Connect with an LLM for processing and understanding natural language queries.
- Interface with a SQL database, enabling the LLM to retrieve and interpret data from the database in response to user queries.

# Setting Up and Understanding the Front-End

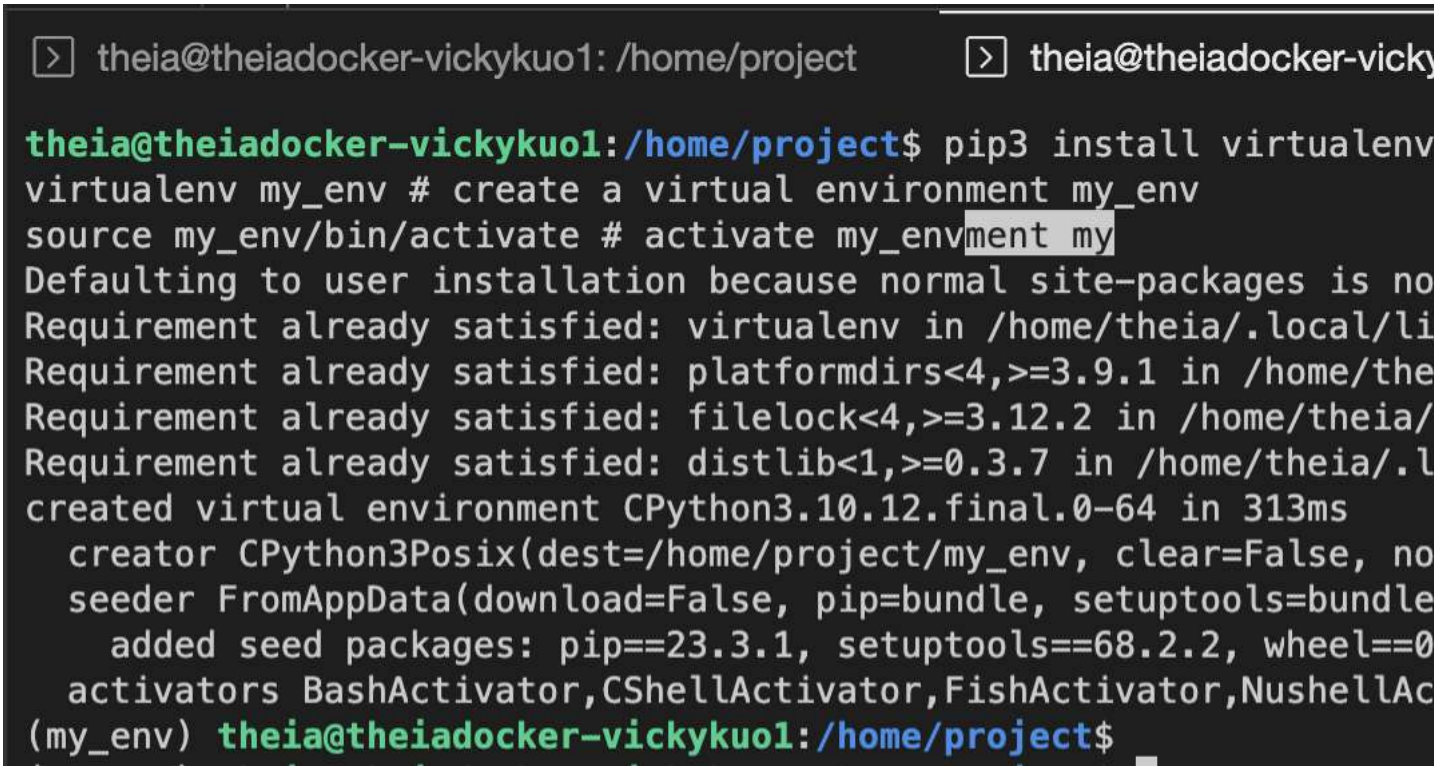
## Setting Up

First, we need to set up your computer for the project. Here's how:
1.To set up the environment Run these commands in the terminal:

```
1
2
3
```

```
pip3 install virtualenv
virtualenv my_env # create a virtual environment my_env
source my_env/bin/activate # activate my_env
```

Copied! Executed!




2. Now, let's get the project files:

```
1
2
3
```

```
git clone https://github.com/ibm-developer-skills-network/TransactBot
mv TransactBot TransactBot
cd TransactBot
```

Copied! Executed!

**Important: The virtual environment has to be rerun every time when you open a new terminal.**

3. Install the project requirements:

```
1
```

```
pip3 install -r requirements.txt
```

Copied! Executed!

☕ Have a cup of coffee, it will take 5-10 minutes to install the requirements (You can continue this project while requirements are installing).

```
1
2
3
4
5
6
7
8
9
```

```
      )  (
     (   ) )
      ) ( (
    _______)_
 .-'---------|
( C|/\/\/\/\/|
 '-./\/\/\/\/|
   '_________'
    '-------'
```

Copied!

**Important: The requirements have to be re-run every time you have a new virtual environment.**

## Understanding the Front-End

Let's talk about how the interface of your app works:

- In the **templates** folder:
  - `index.html`: This file is like the skeleton of your website. It has all the parts like buttons and text boxes.
- In the **static** folder
  - `style.css`: This makes your website look good. It adds colors, styles, and animations.
  - `script.js`: This is where the magic happens. It makes your website interactive, like changing themes or sending messages.

Since delving into front-end development is beyond the scope of our learning objectives, let's directly jump into the backend of building your AI investment advisor app!

# Try LLMs for Free with Watsonx.ai

In this lab you'll interact with one of the foundation models available on watsonx.ai. As these models are part of the IBM Cloud service, accessing them externally requires specific credentials, namely an API key and a project ID. To facilitate your learning experience, we have already generated the necessary credentials for you to access the `meta-llama/llama-2-70b-chat` model through watsonx.ai. You can utilize these credentials directly to set up the LLM by **copying and pasting** the following code into your `model.py`

Open **model.py** in IDE

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
```

```
from ibm_watson_machine_learning.foundation_models import Model
from ibm_watson_machine_learning.foundation_models.extensions.langchain import WatsonxLLM
from ibm_watson_machine_learning.metanames import GenTextParamsMetaNames as GenParams

my_credentials = {
    "url"    : "https://us-south.ml.cloud.ibm.com",
    "token" : "skills-network" # <--- NOTE: specify "skills-network" as your token (NOT as your apikey)
}

params = {
        GenParams.MAX_NEW_TOKENS: 256, # The maximum number of tokens that the model can generate in a single run.
        GenParams.TEMPERATURE: 0.1,   # A parameter that controls the randomness of the token generation. A lower value makes the generation more d
    }

LLAMA2_model = Model(
        model_id= 'meta-llama/llama-2-70b-chat',
        credentials=my_credentials,
        params=params,
        project_id="skills-network",  # <--- NOTE: specify "skills-network" as your project_id
        )

llm = WatsonxLLM(LLAMA2_model)
```

Copied!

We'll be using the `LLAMA_2_70B_CHAT` model because it's a top-notch AI model that's going to power our AI agent.

> IBM watsonx utilizes various language models, including LLAMA2 by Meta, which is currently the strongest open-source language model published as of today (sep 2023).

## Alternative: Activate Your Own IBM Cloud Trial Account

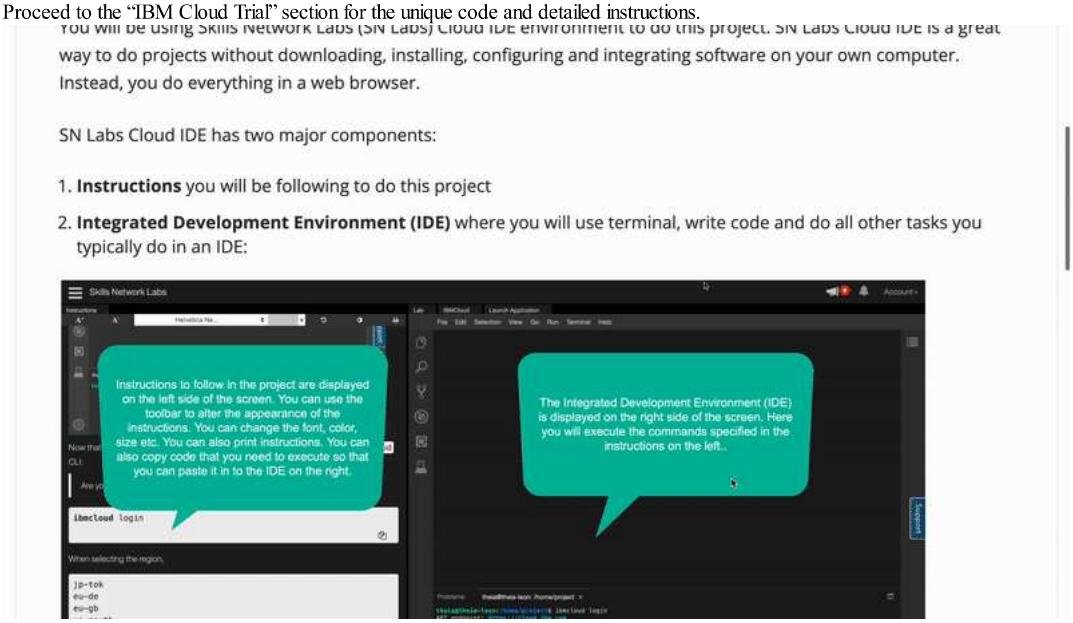
We've also provided you with a free IBM Cloud account, including a **USD 200** credit, so you can explore and use our services freely with your own credentials. To sign up and start using your account, please follow the steps outlined below:

### Step 1: Obtain Your Watsonx API Key and Project ID

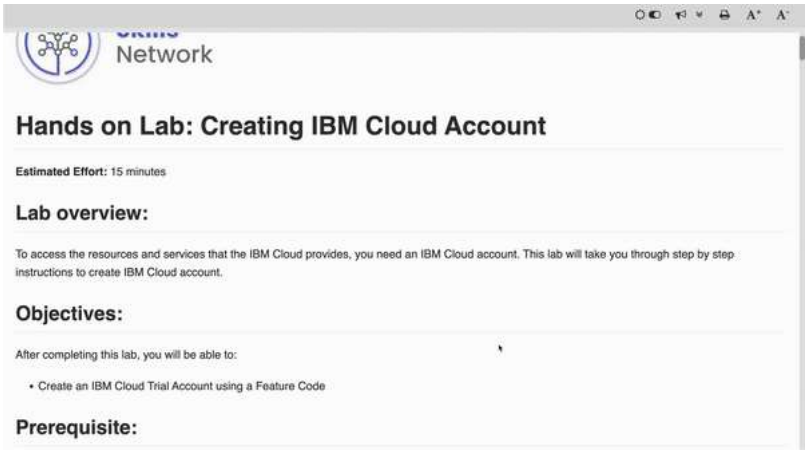
To access the credentials, we've provided a special code for you to claim a USD 200 credit on IBM Cloud at no charge! Here's how to get started:

1. **IBM Cloud Trial Access:**

- Proceed to the “IBM Cloud Trial” section for the unique code and detailed instructions.

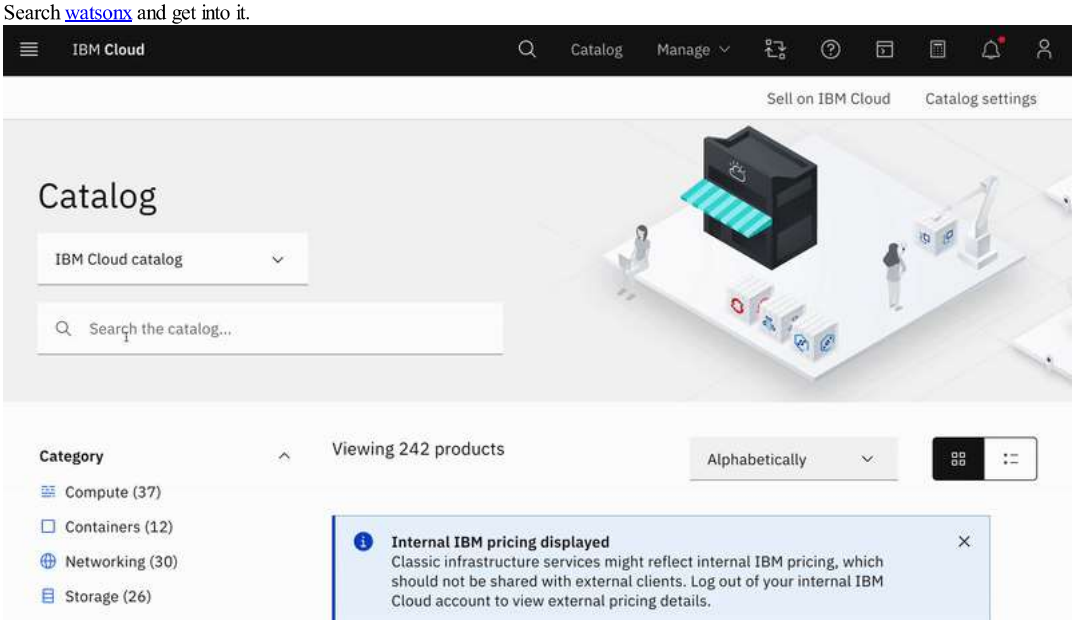



- Once you’ve accessed “IBM Cloud Trial”, follow the instructions on the page to sign up for a non-expiring Lite account at IBM Cloud, which includes the USD 200 credit.

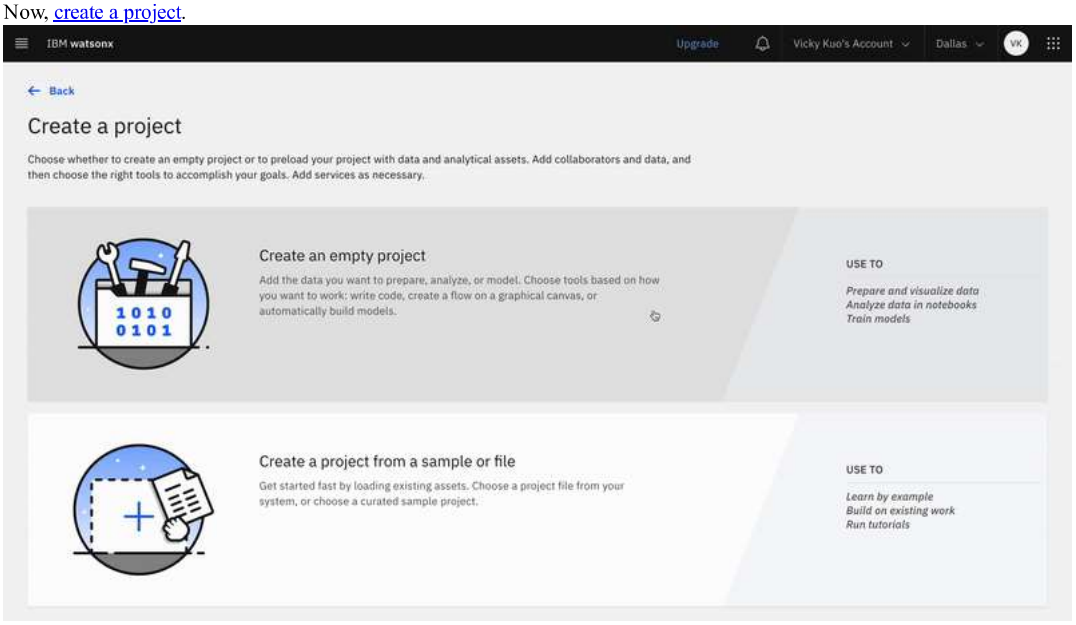



2. **Create Your Watsonx Project:**

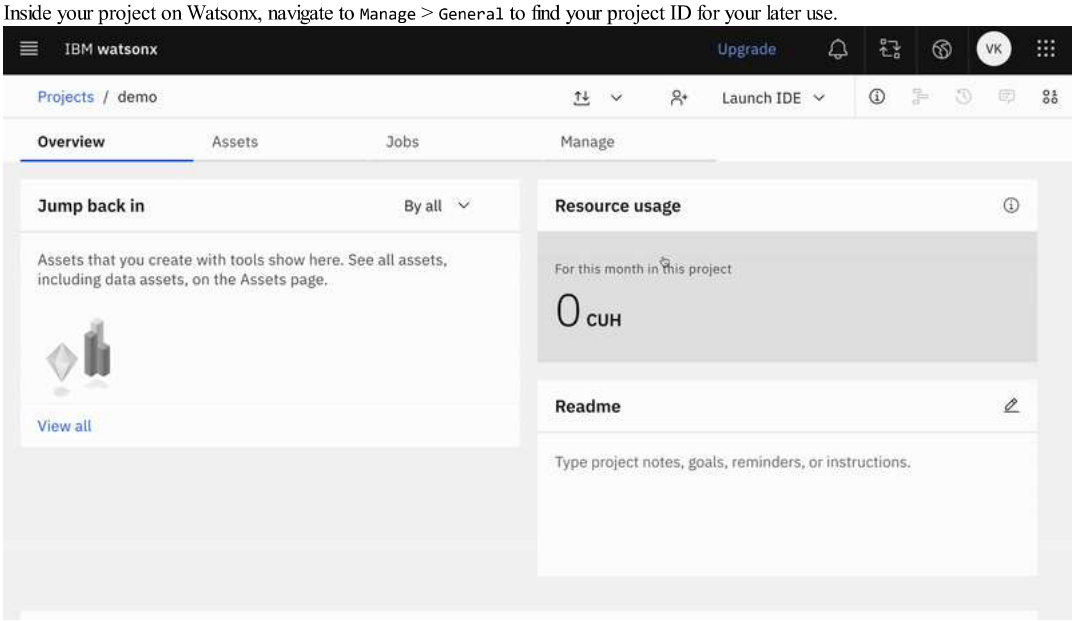
- Search watsonx and get into it.



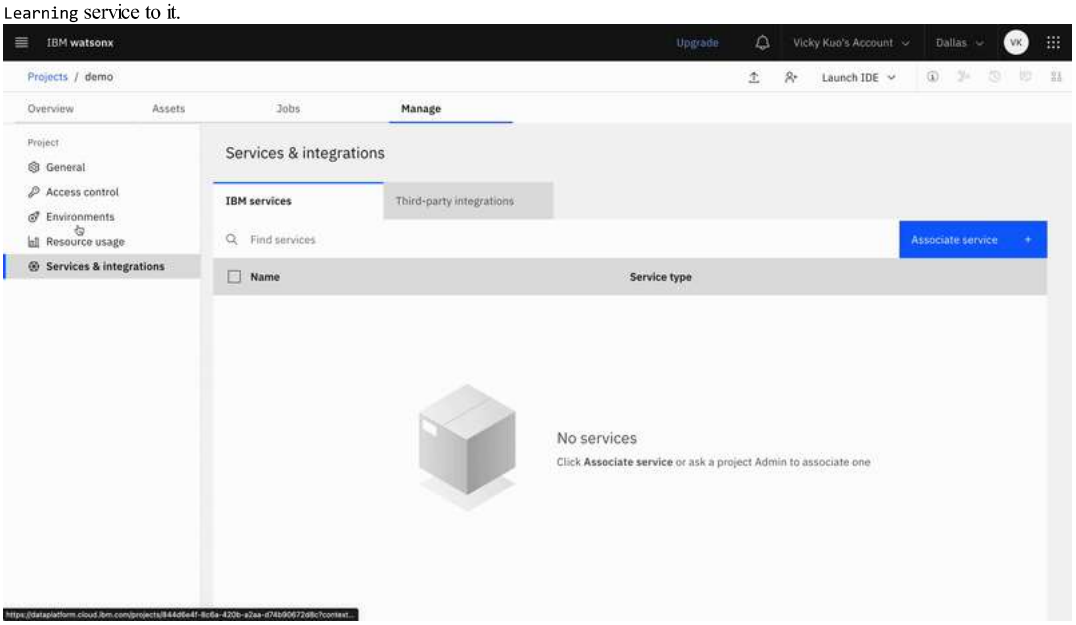
- Now, create a project.



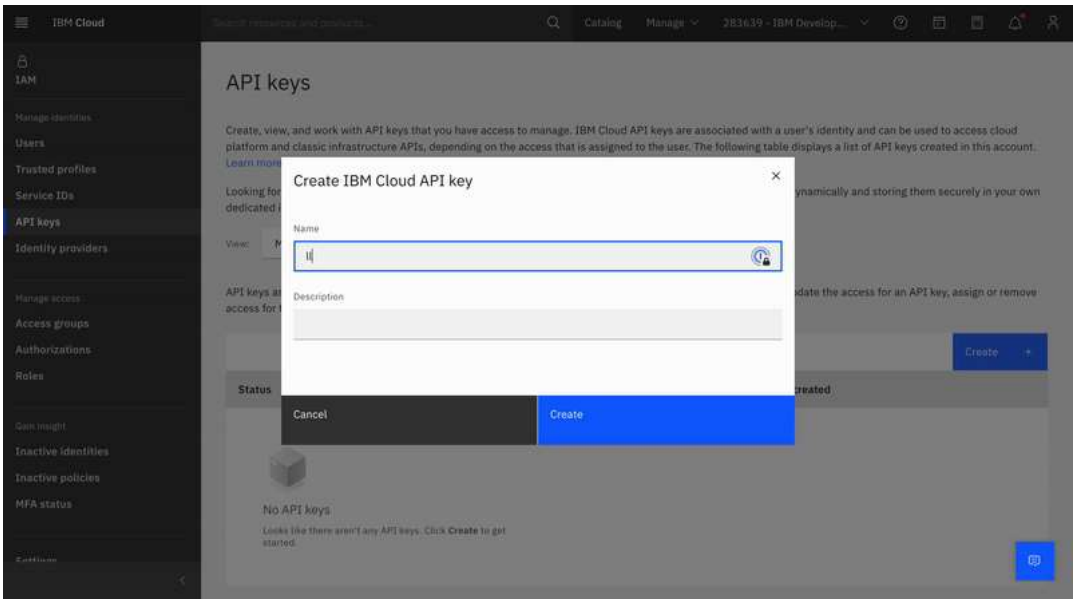

- Inside your project on Watsonx, navigate to `Manage` > `General` to find your project ID for your later use.




3. **Associate Watson Machine Learning Service:**
   After you create a project, you can go to the project's `Manage` tab > select the `Services and integrations` page > click `Associate Service` > add the `Watson Machine Learning` service to it.




4. **Generate IBM Cloud User API Key:**
   Lastly, you can follow the below demonstration to create/get your IBM Cloud user API key. Be sure to write your API key down somewhere right after you create it, because you won't be able to see it again!




### Step 2: Update model.py

Replace the placeholders (`YOUR_API_API` and `YOUR_PROJECT_ID`) with your actual API key and Project ID.

Open **model.py** in IDE

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
```

```
from ibm_watson_machine_learning.foundation_models import Model
from ibm_watson_machine_learning.foundation_models.extensions.langchain import WatsonxLLM
from ibm_watson_machine_learning.metanames import GenTextParamsMetaNames as GenParams

my_credentials = {
    "url"    : "https://us-south.ml.cloud.ibm.com",
    "apikey" : "YOUR_API_API" # <--- NOTE: replace "YOUR_WATSONX_API" with your actual API key)
}

params = {
        GenParams.MAX_NEW_TOKENS: 256, # The maximum number of tokens that the model can generate in a single run.
        GenParams.TEMPERATURE: 0.1,   # A parameter that controls the randomness of the token generation. A lower value makes the generation more d
    }

LLAMA2_model = Model(
        model_id= 'meta-llama/llama-2-70b-chat',
        credentials=my_credentials,
        params=params,
        project_id="YOUR_PROJECT_ID",  # <--- NOTE: replace "YOUR_PROJECT_ID" with your actual project ID
        )

llm = WatsonxLLM(LLAMA2_model)
```

Copied!

# Setting Up the Database

After setting up our LLM using watsonx's API, we'll be setting up a database named 'history' in the `db.py` file. This database will store our credit card transaction data for our LLM's later use.

## Database Compatibility

It's important to note that LangChain is compatible with various databases thanks to its integration via SQLAlchemy, a versatile database toolkit that can connect to various database types including MS SQL, MySQL, MariaDB, PostgreSQL, and Oracle SQL. For our project, we'll be using SQLite. You can learn more about connecting LangChain to different databases in the SQLAlchemy documentation.

## Setting Up SQLite Database

In the `db.py` file, we will incorporate code to create a connection with a SQLite database and to fill it with sample data from a `CSV` file that we've prepared. This sample data is designed to simulate transaction records like those you would download from an actual banking system. Copy and paste the following to dp.py.

Open **db.py** in IDE

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
```

```
from langchain.utilities import SQLDatabase
import sqlite3
import csv

# Create our database
db = SQLDatabase.from_uri("sqlite:///history.db")

# Function to connect to SQLite database
def get_db_connection():
    conn = sqlite3.connect('history.db', check_same_thread=False)
    conn.row_factory = sqlite3.Row
    return conn

def init_db():
    conn = get_db_connection()
    # Create a table
    conn.execute("""
        CREATE TABLE IF NOT EXISTS transactions (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            date DATE,
            category TEXT,
            merchant TEXT,
            amount INTEGER
        );
    """)
    # Insert sample data from CSV file only if the table is empty
    if conn.execute('SELECT COUNT(*) FROM transactions').fetchone()[0] == 0:
        with open('transactions.csv', 'r') as file:
            reader = csv.reader(file)
            next(reader)  # Skip the header row
            sample_data = list(reader)
            conn.executemany('INSERT INTO transactions (date, category, merchant, amount) VALUES (?, ?, ?, ?)', sample_data)

    # Commit the changes and close the connection
    conn.commit()
    conn.close()

init_db()
```

Copied!

To check if the setup is correct, add the following line to the bottom of the `db.py` file. This will print out information about the structure of your database and the sample data you've inserted..

```
1
```

```
print(db.table_info)
```

Copied!

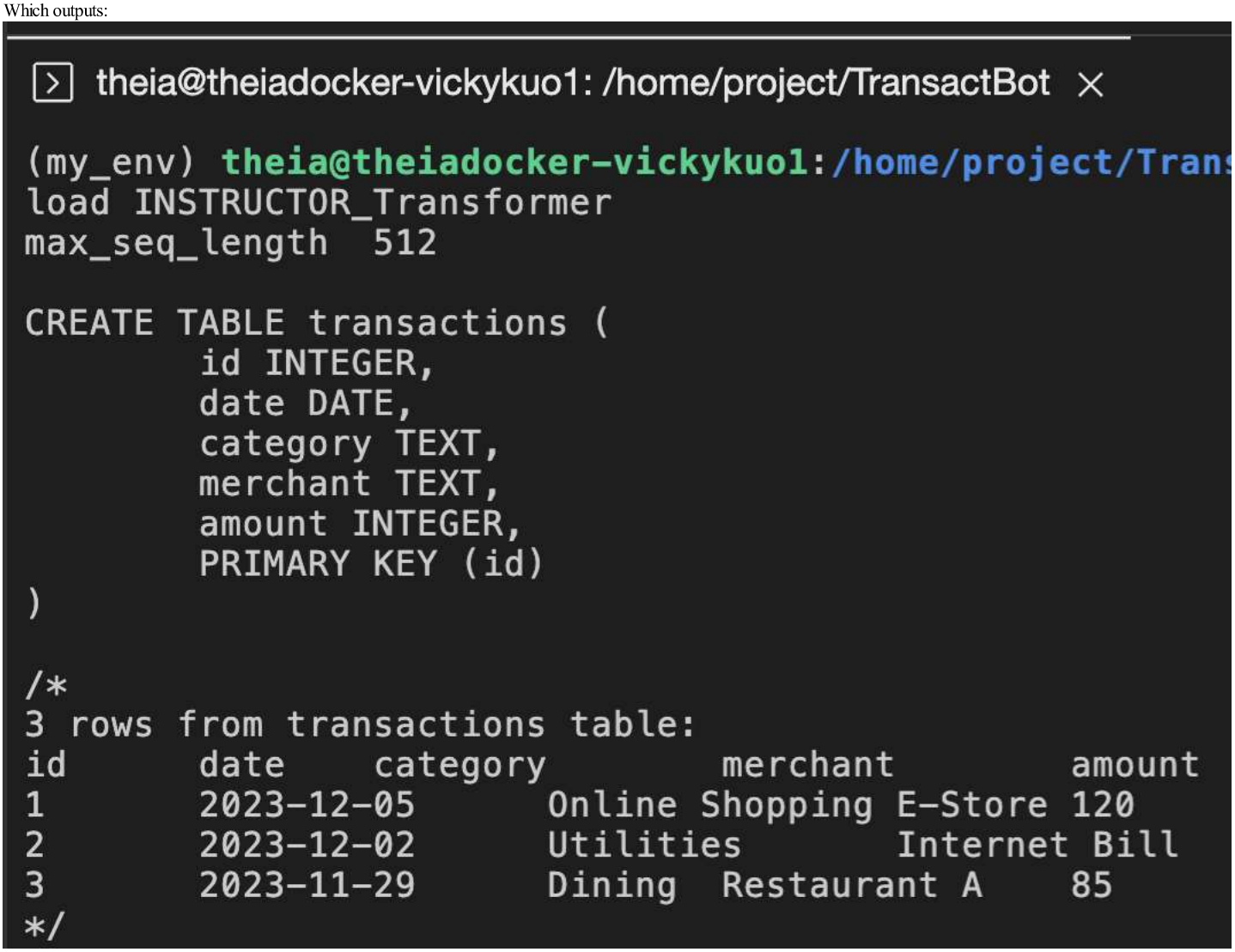
Run the command line in your terminal:

```
1
```

```
python3 db.py
```

Copied! Executed!

Which outputs:




# Crafting the AI Prompt Template

A prompt template is like a cheat sheet for how you want your LLM to behave. Just like you give instructions to a friend, you need to guide the LLM on how to respond to your questions about the database. This is especially important because LLMs work best when they clearly understand what you're asking.

## Extracting Table Columns for AI Prompts

Before crafting our AI prompt template, we need to retrieve the structure of our database table to help the LLM understand what info is stored in your database. For example, knowing there's a 'date' or 'amount' column lets the LLM know it can provide answers related to dates and transaction amounts. Therefore, we'll write a function to get the column names from our SQLite database, which will be used later in our AI template.

Add the following code into the end of the `db.py` file:

```
1
2
3
4
5
6
7
8
9
10
```

```
def get_table_columns(table_name):
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("PRAGMA table_info({})".format(table_name))
    columns = cursor.fetchall()
    print(f"columns:{columns}")
    return [column[1] for column in columns]

table_name = 'transactions'
columns = get_table_columns(table_name)
```

Copied!

This function fetches and prints the column names from our ‘transactions’ table, providing crucial information for our next step.

## Creating the Prompt Template

With the table columns identified, we now create a prompt template referring to the table columns. The prompt template you are about to create uses specific tags to differentiate between what the the system says and what the user says. These tags help the LLM to understand who is ‘speaking’ in the conversation. Here’s a quick breakdown:

- `<<SYS>>`/`<</SYS>>` tags: These tags signify the system (AI’s) message.
- `[INST]`/`[/INST]` tags: These tags indicate user messages.

Now, let’s put what we’ve learned into practice. We’ll use these tags to construct a template. This template will guide the LLM on how to interpret your database queries and how to respond. Here’s the code to add at the end of your `app.py` file:

Open **app.py** in IDE

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
```

```
# LLaMA uses [INST] and [/INST] to indicate user messages, and <<SYS>> and <</SYS>> to indicate system messages
QUERY = """
<<SYS>>
You are a powerful text-to-SQLite model. Your job is to answer questions about a database. You are given a question and context regarding the table
The table name is {table_name} and corresponding columns are {columns}.
You must run SQLite queries to the table to find an answer. Ensure your query does not include any non-SQLite syntax such as DATE_TRUNC or any use
Provide strategies to manage expenses or tips to reduce the expenses in your answer as well. Compare the result with the spending in the previous m

Guidelines:
- Filter results using the current time zone: {time} only when query specifis a specific date/time period. You should use ">=" or "<=" operators to
- If the query result is [(None,)], run the SQLite query again to double check the answer.
- If a specific category is mentioned in the inquiry, such as 'Groceries', 'Dining', or 'Utilities', use the "WHERE" condition in your SQL query to
- If not asked for a specific category, yuou shouldn't filter any category out. On the other hand, you should use "where" condition to do the filte
- When asked for \'highest\' or \'lowest\', use SQL function MAX() or MIN() respectively.


Use the following format to answer the inquiry:

Response: Result of the SQLite-compatible SQL query. If you know th transaction detailes such as the date, category, merchant, and amount, mention
---------------------- line break
<br>
---------------------- line break
<br>
Explanation: Concise and succinct explanation on your thought process on how to get the final answer including the relevant transaction details suc
---------------------- line break
<br>
---------------------- line break
<br>
Advice: Provide strategies to manage expenses or tips to reduce the expenses here.

<</SYS>>

[INST]
{inquiry}
[/INST]
"""
```

Copied!

You might’ve noticed the query above have another two variables: `{inquiry}` and `{time}` in addition to `{table_name}` and `columns` just identified. All of these variables will be assigned in the next page.

# Prompting the Database

## Initializing the Database Chain

To bridge our database with the LLM, we use LangChain's SQLDatabaseChain. This library connects our SQLite database to the LLM, allowing for seamless interaction.

Add the following code at the end of your `app.py` file to initialize the database chain:

```
1
2
3
4
5
6
```

```
from db import *
from langchain_experimental.sql import SQLDatabaseChain
from model import llm

# Create the database chain
db_chain = SQLDatabaseChain.from_llm(llm, db, verbose=True)
```

Copied!

## Interacting with the Database

Our Flask app's `/inquiry` route will handle incoming queries to retrieve the desired data from the database. In this function, we merge user prompts with the prompt template we created earlier and put the values of variables into it.

Add the following code at the end of your `app.py` file to manage interactions:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
```

```
from datetime import datetime
import pytz

@app.route('/inquiry', methods=['POST'])
def submit_inquiry():
    inquiry = request.form['inquiry']

    # Handle transaction queries
    prompt = QUERY.format(table_name=table_name, columns=columns, time=datetime.now(pytz.timezone('America/New_York')), inquiry=inquiry)
    response = db_chain.run(prompt)

    # Fetch transactions data again to pass to the template
    conn = get_db_connection()
    cursor = conn.execute('SELECT id, * FROM transactions ORDER BY date DESC')
    transactions = [dict(ix) for ix in cursor.fetchall()]
    conn.close()

    # Replace newline characters with HTML break tags
    response = response.replace('\n', '<br>')

    return render_template('index.html', inquiry=prompt, answer=response, transactions=transactions)
```

Copied!

# Running the App

Finally, it's time to see our setup in action. Run the Flask app by adding the following code at the end of your `app.py` file:

```
1
2
```

```
if __name__ == '__main__':
    app.run(debug=True)
```

Copied!

Run this command in your terminal to start the server:

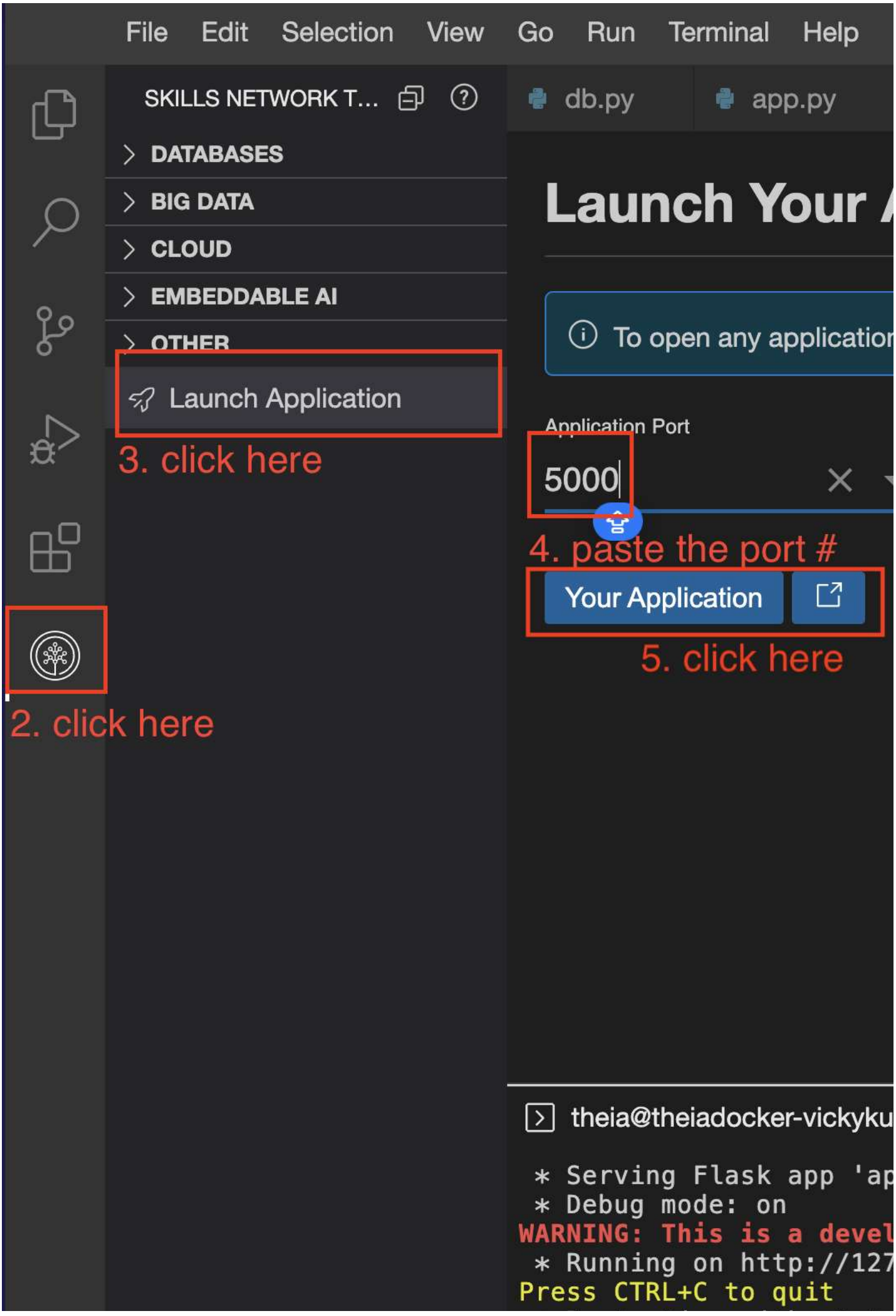
```
1
```

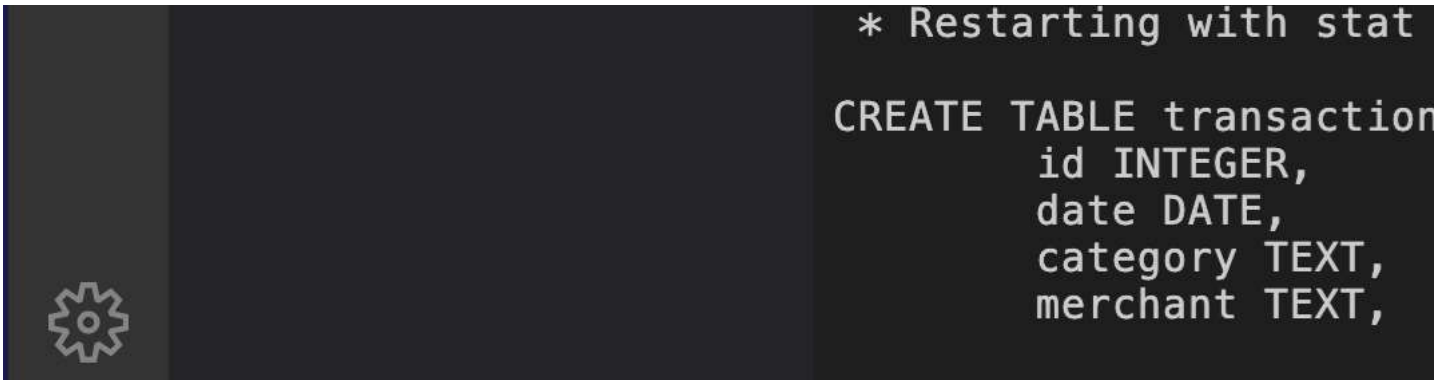
```
python3 app.py
```

Copied! Executed!

To access your server, follow these steps:

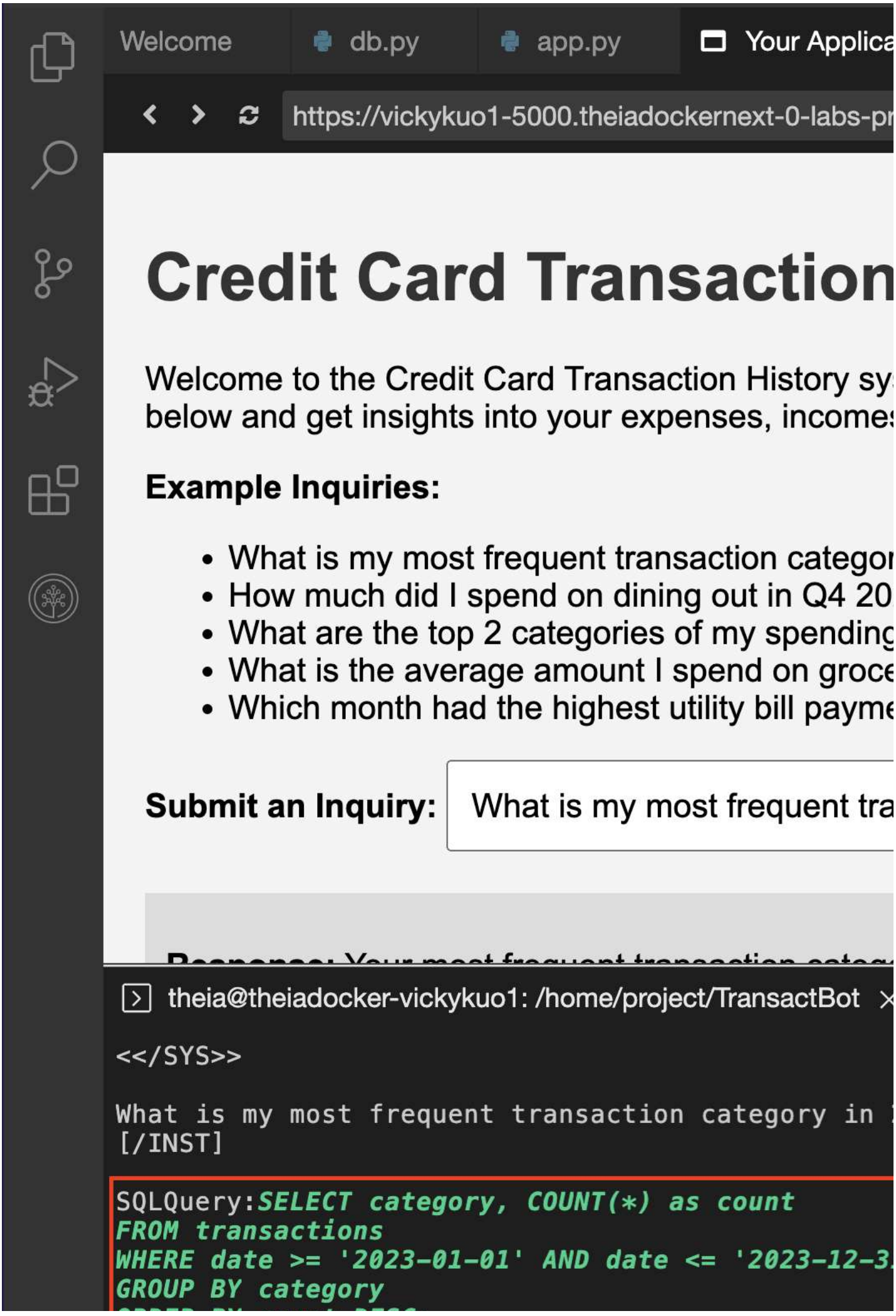
1. Copy the application port number from your terminal-usualy port 5000
2. Navigate to the Skills Network Toolbox from the toolbar on the left side of the IDE
3. Click on **“Launch Application”** in the adjacent vertical sidebar
4. Paste your application port number
5. Launch the application in a new browser tab


File Edit Selection View Go Run Terminal Help
SKILLS NETWORK T...
db.py
app.py
DATABASES
BIG DATA
CLOUD
EMBEDDABLE AI
OTHER
Launch Application
3. click here
Launch Your
To open any application
Application Port
5000
4. paste the port #
Your Application
5. click here
2. click here
theia@theiadocker-vickyku
* Serving Flask app 'ap
* Debug mode: on
WARNING: This is a devel
* Running on http://127
Press CTRL+C to quit




Now you can submit a question and check your terminal to see how the LLM constructs the SQL query.

**Important:** **Remember, AI's responses may vary, so you might need to adjust your prompt template for better results.**


Welcome
db.py
app.py
Your Applica
https://vickykuo1-5000.theiadockernext-0-labs-pr
Credit Card Transaction
Welcome to the Credit Card Transaction History sy
below and get insights into your expenses, income
Example Inquiries:
What is my most frequent transaction categor
How much did I spend on dining out in Q4 20
What are the top 2 categories of my spending
What is the average amount I spend on groce
Which month had the highest utility bill payme
Submit an Inquiry:
What is my most frequent tra
theia@theiadocker-vickykuo1: /home/project/TransactBot
<</SYS>>
What is my most frequent transaction category in
[/INST]
SQLQuery:SELECT category, COUNT(*) as count
FROM transactions
WHERE date >= '2023-01-01' AND date <= '2023-12-3
GROUP BY category


## View the Complete Code in app.py

To review the entire code as it should appear in your current app.py file, click the following collapsible section:

▼ Click here to expand and view the full code in the app.py

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
```

```
from flask import Flask, render_template, request

app = Flask(__name__)

```

```
@app.route('/')
def index():
    conn = get_db_connection()
    transactions = conn.execute('SELECT id, * FROM transactions ORDER BY date DESC').fetchall()
    conn.close()
    return render_template('index.html', transactions=transactions)


# LLaMA uses [INST] and [/INST] to indicate user messages, and <<SYS>> and <</SYS>> to indicate system messages
QUERY = """
<<SYS>>
You are a powerful text-to-SQLite model. Your job is to answer questions about a database. You are given a question and context regarding the table
The table name is {table_name} and corresponding columns are {columns}.
You must run SQLite queries to the table to find an answer. Ensure your query does not include any non-SQLite syntax such as DATE_TRUNC or any use
Provide strategies to manage expenses or tips to reduce the expenses in your answer as well. Compare the result with the spending in the previous m

Guidelines:
- Filter results using the current time zone: {time} only when query specifis a specific date/time period. You should use ">=" or "<=" operators to
- If the query result is [(None,)], run the SQLite query again to double check the answer.
- If a specific category is mentioned in the inquiry, such as 'Groceries', 'Dining', or 'Utilities', use the "WHERE" condition in your SQL query to
- If not asked for a specific category, yuou shouldn't filter any category out. On the other hand, you should use "where" condition to do the filte
- When asked for \'highest\' or \'lowest\', use SQL function MAX() or MIN() respectively.


Use the following format to answer the inquiry:

Response: Result of the SQLite-compatible SQL query. If you know th transaction detailes such as the date, category, merchant, and amount, mention
---------------------- line break
<br>
---------------------- line break
<br>
Explanation: Concise and succinct explanation on your thought process on how to get the final answer including the relevant transaction details suc
---------------------- line break
<br>
---------------------- line break
<br>
Advice: Provide strategies to manage expenses or tips to reduce the expenses here.

<</SYS>>

[INST]
{inquiry}
[/INST]
"""

from db import *
from langchain_experimental.sql import SQLDatabaseChain
from model import llm

# Create the database chain
db_chain = SQLDatabaseChain.from_llm(llm, db, verbose=True)

from datetime import datetime
import pytz

@app.route('/inquiry', methods=['POST'])
def submit_inquiry():
    inquiry = request.form['inquiry']

    # Handle transaction queries
    prompt = QUERY.format(table_name=table_name, columns=columns, time=datetime.now(pytz.timezone('America/New_York')), inquiry=inquiry)
    response = db_chain.run(prompt)

    # Fetch transactions data again to pass to the template
    conn = get_db_connection()
    cursor = conn.execute('SELECT id, * FROM transactions ORDER BY date DESC')
    transactions = [dict(ix) for ix in cursor.fetchall()]
    conn.close()

    # Replace newline characters with HTML break tags
    response = response.replace('\n', '<br>')

    return render_template('index.html', inquiry=prompt, answer=response, transactions=transactions)

if __name__ == '__main__':
    app.run(debug=True)
```

Copied!

## Sample Questions and Responses

The response of the sample questions we tested are summarized below.

**Important: AI responses can vary, so you might need to fine-tune your prompt template for improved results.**

| Question | SQL Query | SQL Result | Analysis |
|---|---|---|---|
| What is my most frequent transaction category in 2023? | `SELECT category, COUNT(*) as count FROM transactions WHERE date >= '2023-01-01' AND date <= '2023-12-31' GROUP BY category ORDER BY count DESC;` | [('Utilities', 6), ('Travel', 4), ('Online Shopping', 4), ('Dining', 4), ('Groceries', 3), ('Entertainment', 2)] | Correct |
| How much did I spend on dining out in Q4 2023? | `SELECT SUM(amount) FROM transactions WHERE category = 'Dining' AND date >= '2023-10-01' AND date <= '2023-12-31'` | [(105,)] | Correct |
| What are the top 2 categories of my spendings in 2023? | `SELECT category, SUM(amount) AS total_amount FROM transactions WHERE date >= '2023-01-01' AND date <= '2023-12-31' GROUP BY category ORDER BY total_amount DESC LIMIT 2;` | [('Travel', 615), ('Online Shopping', 615)] | Correct |
| What is the average amount I spend on | `SELECT AVG(amount) FROM transactions WHERE category = 'Groceries';` | [(63.333333333333336,)] | Correct |

| Question | SQL Query | SQL Result | Analysis |
|---|---|---|---|
| groceries? | | | |
| Which month had the highest utility bill payments in 2023? | `SELECT strftime('%m', date) AS month, SUM(amount) AS total_utility_bill FROM transactions WHERE category = 'Utilities' GROUP BY month ORDER BY month desc` | [('12', 80), ('11', 75), ('10', 40), ('09', 100), ('08', 60), ('07', 50)] | Correct |

# Conclusion

🧨 And that's a wrap! We've just scratched the surface of what's possible with LangChain, and what a thrilling journey it's been! We've turned database interactions into easy conversations thanks to the magic of LLMs. It's like having a friendly chat with your data, making insights accessible and fun.

Imagine taking this concept to the next level – applying it across all your data, structured or unstructured. We're talking about a total game-changer in how we interact with and derive value from our data. So, embrace the future with excitement and let LangChain be your guide to a world where data conversations are as natural as your morning coffee chat. Here's to many more data adventures! 🚀🎇🧨

## Author(s)

Vicky Kuo

## Changelog

| Date | Version | Changed by | Change Description |
|---|---|---|---|
| 2024-11-30 | 0.1 | Vicky Kuo | Initial version created |