

Directory structure:

```
└─ data-science/

    └─ README.md

    └─ pyproject.toml

    └─ .env.example

    └─ data_science/

        |   └─ __init__.py

        |   └─ agent.py

        |   └─ prompts.py

        |   └─ tools.py

        |   └─ sub_agents/

            |   |   └─ __init__.py

            |   |   └─ analytics/

                |   |   |   └─ __init__.py

                |   |   |   └─ agent.py

                |   |   |   └─ prompts.py

                |   |   └─ bigquery/

                    |   |   |   └─ __init__.py

                    |   |   |   └─ agent.py

                    |   |   |   └─ prompts.py

                    |   |   |   └─ tools.py

                    |   |   └─ chase_sql/

                        |   |   |   └─ __init__.py

                        |   |   |   └─ chase_constants.py

                        |   |   |   └─ chase_db_tools.py

                        |   |   |   └─ dc_prompt_template.py

                        |   |   |   └─ llm_utils.py
```

```
| | | └─ qp_prompt_template.py
| | | └─ sql_postprocessor/
| | | └─ README.md
| | | └─ __init__.py
| | | └─ correction_prompt_template.py
| | | └─ sql_translator.py
| | └─ bqml/
| | └─ __init__.py
| | └─ agent.py
| | └─ prompts.py
| | └─ tools.py
| └─ utils/
|   └─ create_bq_table.py
|   └─ reference_guide_RAG.py
|   └─ utils.py
|   └─ data/
└─ deployment/
  └─ __init__.py
  └─ deploy.py
  └─ test_deployment.py
└─ eval/
  └─ __init__.py
  └─ test_eval.py
  └─ eval_data/
    └─ simple.test.json
    └─ test_config.json
└─ tests/
  └─ test_agents.py
```

=====


File: README.md

=====

# Data Science with Multiple Agents

## Overview

This project demonstrates a multi-agent system designed for sophisticated data analysis. It integrates several specialized agents to handle different aspects of the data pipeline, from data retrieval to advanced analytics and machine learning. The system is built to interact with BigQuery, perform complex data manipulations, generate data visualizations and execute machine learning tasks using BigQuery ML (BQML). The agent can generate text response as well as visuals, including plots and graphs for data analysis and exploration.

 **\*\*Watch the Video Walkthrough:\*\*** [How to build a Data Science agent with ADK] (<https://www.youtube.com/watch?v=efcUXoMX818>)

## Agent Details

The key features of the Data Science Multi-Agent include:

Feature	Description
---	---
<b>**Interaction Type:**</b>	Conversational
<b>**Complexity:**</b>	Advanced
<b>**Agent Type:**</b>	Multi Agent
<b>**Components:**</b>	Tools, AgentTools, Session Memory, RAG
<b>**Vertical:**</b>	All (Applicable across industries needing advanced data analysis)

### Architecture

![Data Science Architecture] (data-science-architecture.png)

### ### Key Features

\* **Multi-Agent Architecture:** Utilizes a top-level agent that orchestrates sub-agents, each specialized in a specific task.

\* **Database Interaction (NL2SQL):** Employs a Database Agent to interact with BigQuery using natural language queries, translating them into SQL.

\* **Data Science Analysis (NL2Py):** Includes a Data Science Agent that performs data analysis and visualization using Python, based on natural language instructions.

\* **Machine Learning (BQML):** Features a BQML Agent that leverages BigQuery ML for training and evaluating machine learning models.

\* **Code Interpreter Integration:** Supports the use of a Code Interpreter extension in Vertex AI for executing Python code, enabling complex data analysis and manipulation.

\* **ADK Web GUI:** Offers a user-friendly GUI interface for interacting with the agents.

\* **Testability:** Includes a comprehensive test suite for ensuring the reliability of the agents.

## ## Setup and Installation

### ### Prerequisites

\* **Google Cloud Account:** You need a Google Cloud account with BigQuery enabled.

\* **Python 3.12+:** Ensure you have Python 3.12 or a later version installed.

\* **Poetry:** Install Poetry by following the instructions on the official Poetry website: [<https://python-poetry.org/docs/>] (<https://python-poetry.org/docs/>)

\* **Git:** Ensure you have git installed. If not, you can download it from [<https://git-scm.com/>] (<https://git-scm.com/>) and follow the [installation

guide] (<https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>).

### ### Project Setup with Poetry

#### 1. \*\*Clone the Repository:\*\*

```
```bash

git clone https://github.com/google/adk-samples.git

cd adk-samples/python/agents/data-science

```
```

#### 2. \*\*Install Dependencies with Poetry:\*\*

```
```bash

poetry install

```
```

This command reads the `pyproject.toml` file and installs all the necessary dependencies into a virtual environment managed by Poetry.

#### 3. \*\*Activate the Poetry Shell:\*\*

```
```bash

poetry env activate

```
```

This activates the virtual environment, allowing you to run commands within the project's environment. To make sure the environment is active, use

for example

```
```bash

$> poetry env list

data-science-FAlhSuLn-py3.13 (Activated)

```
```

If the above command did not activate the environment for you, you can also activate it through

```
```bash

source $(poetry env info --path)/bin/activate

```
```

#### 4. **\*\*Set up Environment Variables:\*\***

Rename the file ".env.example" to ".env"

Fill the below values:

```
```bash

# Choose Model Backend: 0 -> ML Dev, 1 -> Vertex

GOOGLE_GENAI_USE_VERTEXAI=1


# ML Dev backend config. Fill if using ML Dev backend.

GOOGLE_API_KEY='YOUR_VALUE_HERE'


# Vertex backend config

GOOGLE_CLOUD_PROJECT='YOUR_VALUE_HERE'

GOOGLE_CLOUD_LOCATION='YOUR_VALUE_HERE'

```
```

Follow the following steps to set up the remaining environment variables.

#### 5. **\*\*BigQuery Setup:\*\***

These steps will load the sample data provided in this repository to BigQuery.

For our sample use case, we are working on the Forecasting Sticker Sales data from Kaggle:

Walter Reade and Elizabeth Park. Forecasting Sticker Sales. <https://kaggle.com/competitions/playground-series-s5e1>, 2025. Kaggle.

\* First, set the BigQuery project ID in the ``.env`` file. This can be the same GCP Project you use for ``GOOGLE_CLOUD_PROJECT``,

but you can use other BigQuery projects as well, as long as you have access permissions to that project.

If you have an existing BigQuery table you wish to connect, specify the ``BQ_DATASET_ID`` in the ``.env`` file as well.

Make sure you leave ``BQ_DATASET_ID='forecasting_sticker_sales'`` if you wish to use the sample data.

Alternatively, you can set the variables from your terminal:

```
```bash

export BQ_PROJECT_ID='YOUR-BQ-PROJECT-ID'

export BQ_DATASET_ID='YOUR-DATASET-ID' # leave as
'forecasting_sticker_sales' if using sample data

```
```

You can skip the upload steps if you are using your own data. We recommend not adding any production critical datasets to this sample agent.

If you wish to use the sample data, continue with the next step.

\* You will find the datasets inside 'data-science/data\_science/utils/data/'.

Make sure you are still in the working directory ('agents/data-science'). To load the test and train tables into BigQuery, run the following commands:

```
```bash

python3 data_science/utils/create_bq_table.py

```
```

## 6. \*\*BQML Setup:\*\*

The BQML Agent uses the Vertex AI RAG Engine to query the full BigQuery ML Reference Guide.

Before running the setup, ensure your project ID is added in .env file: ``"GOOGLE_CLOUD_PROJECT"``.

Leave the corpus name empty in the .env file: ``BQML_RAG_CORPUS_NAME = ````. The corpus name will be added automatically once it's created.

To set up the RAG Corpus for your project, run the methods ``create_RAG_corpus()`` and ``ingest_files()`` in

``data-science/data_science/utils/reference_guide_RAG.py`` by running the below command from the working directory:

```
```bash

python3 data_science/utils/reference_guide_RAG.py

```
```

## 7. \*\*Other Environment Variables:\*\*

\* ``NL2SQL_METHOD``: (Optional) Either ``BASELINE`` or ``CHASE``. Sets the method for SQL Generation. Baseline uses Gemini off-the-shelf, whereas CHASE uses [CHASE-SQL] (<https://arxiv.org/abs/2410.01943>)



\* ``CODE_INTERPRETER_EXTENSION_NAME``: (Optional) The full resource name of

a pre-existing Code Interpreter extension in Vertex AI. If not provided,

a new extension will be created. (e.g.,

``projects/<YOUR_PROJECT_ID>/locations/<YOUR_LOCATION>/extensions/<YOUR_EXTENSION_ID>``).

Check the logs/terminal for the ID of the newly created Code Interpreter

Extension and provide the value in your environment variables to avoid creating multiple extensions.

From the terminal:

```
```bash
```

```
export CODE_INTERPRETER_EXTENSION_NAME='projects/<YOUR_PROJECT_ID>/locations/us-central1/extensions/<YOUR_EXTENSION_ID>'
```

```
```
```

## Running the Agent

You can run the agent using the ADK command in your terminal.

from the working directory:

1. Run agent in CLI:

```
```bash
```

```
poetry run adk run data_science
```

```
```
```

2. Run agent with ADK Web UI:

```
```bash
```

```
poetry run adk web
```

```
```
```

Select the data\_science from the dropdown

### ### Example Agent Interaction

Here's a quick example of how a user might interact with the Data Science Multi-Agent System:

```
> **User:** Hi, What data do you have access to?
```

```
> **Agent:** I have access to two tables: `train` and `test`. Both tables contain sticker sales data with the following columns: `id`, `date`, `country`, `store`, `product`, and `num_sold`.
```

```
> **User:** I need more details on the train table. What countries exist? How many stores are there?
```

```
> **Agent:** Result: The train table contains data from Canada, Finland, Italy, Kenya, Norway, and Singapore. Each of these countries has 3 distinct stores.
```

```
>Explanation: The database agent executed a SQL query against the train table to find the distinct countries and the number of distinct stores within each country. The query grouped the results by country and counted the distinct store values.
```

```
> **User:** Please generate a plot with total sales per country.
```

```
> **Agent:** Result: A bar plot has been generated showing the total sales
```

per country. The x-axis represents the country, and the y-axis represents the total sales.

> Explanation:

> 1. The database agent was first called to retrieve the total sales per country from the train table.

> 2. The data science agent was then called to generate a bar plot visualizing the results. The plot displays each country on the x-axis and its corresponding total sales on the y-axis.

> **User:** What kinds of forecasting models can I train in BQML?

> **Agent:** You can train several types of forecasting models in BQML, including ARIMA, Exponential Smoothing, and Temporal Fusion Transformer (TFT) models. Would you like to proceed with training a forecasting model?

> **User:** Yes

> **Agent:** Okay, before proceeding, let's confirm the project and dataset ID for BQML. We'll be using the `PROJECT\_ID` project and the `forecasting\_sticker\_sales` dataset. Does that sound right?

> **User:** Yes

> **Agent:** The BQML supports `ARIMA\_PLUS`, `ARIMA\_PLUS\_XREG`.

>To proceed, I need to know which model you'd like to use. I also need to know the following:

>\* The name of the column that provides the time points (e.g., 'date').

>\* The name of the column that contains the data to forecast (e.g., 'num\_sold').

>\* Do you want to fit and forecast multiple time series using a single query? If so, what are the ID columns? (e.g., `country`, `store`, `product`)

## ## Testing and Evaluation

To run the test and evaluation code, you need a few additional dependencies.  
Run

the following Poetry command from the `agents/data-science` directory to  
install them:

```
```bash
```

```
poetry install --with=dev
```

```
```
```

## ### Running Evaluations

Evaluation tests assess the overall performance and capabilities of the agent  
in a holistic manner.

**\*\*Run Evaluation Tests:\*\***

```
```bash
```

```
poetry run pytest eval
```

```
```
```

- This command executes all test files within the `eval/` directory.

- `poetry run` ensures that pytest runs within the project's virtual  
environment.

### ### Running Tests

Tests assess the overall executability of the agents.

#### \*\*Test Categories:\*\*

\* **\*\*Integration Tests:\*\*** These tests verify that the agents can interact correctly with each other and with external services like BigQuery. They ensure that the root agent can delegate tasks to the appropriate sub-agents and that the sub-agents can perform their intended tasks.

\* **\*\*Sub-Agent Functionality Tests:\*\*** These tests focus on the specific capabilities of each sub-agent (e.g., Database Agent, BQML Agent). They ensure that each sub-agent can perform its intended tasks, such as executing SQL queries or training BQML models.

\* **\*\*Environment Query Tests:\*\*** These tests verify that the agent can handle queries that are based on the environment.

#### \*\*Run Tests:\*\*

```
```bash
```

```
poetry run pytest tests
```

```
```
```

- This command executes all test files within the `tests/` directory.

- `poetry run` ensures that pytest runs within the project's virtual environment.

### ## Deployment on Vertex AI Agent Engine

To deploy the agent to Google Agent Engine, first follow

[these steps] (<https://cloud.google.com/vertex-ai/generative-ai/docs/agent-engine/set-up>)

to set up your Google Cloud project for Agent Engine.

You also need to give BigQuery User, BigQuery Data Viewer, and Vertex AI User permissions to the Reasoning Engine Service Agent. Run the following commands to

grant the required permissions:

```
```bash
```

```
export RE_SA="service-${GOOGLE_CLOUD_PROJECT_NUMBER}@gcp-sa-aiplatform-  
re.iam.gserviceaccount.com"
```

```
gcloud projects add-iam-policy-binding ${GOOGLE_CLOUD_PROJECT} \  
  --member="serviceAccount:${RE_SA}" \  
  --condition=None \  
  --role="roles/bigquery.user"
```

```
gcloud projects add-iam-policy-binding ${GOOGLE_CLOUD_PROJECT} \  
  --member="serviceAccount:${RE_SA}" \  
  --condition=None \  
  --role="roles/bigquery.dataViewer"
```

```
gcloud projects add-iam-policy-binding ${GOOGLE_CLOUD_PROJECT} \  
  --member="serviceAccount:${RE_SA}" \  
  --condition=None \  
  --role="roles/aiplatform.user"
```

```
```
```

Next, you need to create a ``.whl`` file for your agent. From the ``data-science`` directory, run this command:

```
```bash
```

```
poetry build --format=wheel --output=deployment
```

```
...
```

This will create a file named ``data_science-0.1-py3-none-any.whl`` in the ``deployment`` directory.

Then run the below command. This will create a staging bucket in your GCP project and deploy the agent to Vertex AI Agent Engine:

```
```bash
```

```
cd deployment/
```

```
python3 deploy.py --create
```

```
...
```

When this command returns, if it succeeds it will print an AgentEngine resource

name that looks something like this:

```
...
```

```
projects/*****/locations/us-central1/reasoningEngines/  
7737333693403889664
```

```
...
```

The last sequence of digits is the AgentEngine resource ID.

Once you have successfully deployed your agent, you can interact with it using the ``test_deployment.py`` script in the ``deployment`` directory. Store the agent's resource ID in an environment variable and run the following command:

```
```bash
```

```
export RESOURCE_ID=...
```

```
export USER_ID=<any string>
```

```
python test_deployment.py --resource_id=$RESOURCE_ID --user_id=$USER_ID
...

```

The session will look something like this:

```
...
```

```
Found agent with resource ID: ...
```

```
Created session for user ID: ...
```

```
Type 'quit' to exit.
```

```
Input: Hello. What data do you have?
```

```
Response: I have access to the train and test tables inside the
forecasting_sticker_sales dataset.
```

```
...
```

```
...
```

Note that this is *\*not\** a full-featured, production-ready CLI; it is just intended to

show how to use the Agent Engine API to interact with a deployed agent.

The main part of the `test_deployment.py` script is approximately this code:

```
```python
from vertexai import agent_engines

remote_agent = vertexai.agent_engines.get(RESOURCE_ID)

session = remote_agent.create_session(user_id=USER_ID)

while True:

    user_input = input("Input: ")

    if user_input == "quit":

        break

    for event in remote_agent.stream_query(

```



```

        user_id=USER_ID,

        session_id=session["id"],

        message=user_input,

    ):

        parts = event["content"]["parts"]

        for part in parts:

            if "text" in part:

                text_part = part["text"]

                print(f"Response: {text_part}")

...

```

To delete the agent, run the following command (using the resource ID returned previously):

```

```bash

python3 deployment/deploy.py --delete --resource_id=RESOURCE_ID

...

```

## ## Optimizing and Adjustment Tips

\* **Prompt Engineering:** Refine the prompts for ``root_agent``, ``bqml_agent``, ``db_agent``

and ``ds_agent`` to improve accuracy and guide the agents more effectively.

Experiment with different phrasing and levels of detail.

\* **Extension:** Extend the multi-agent system with your own AgentTools or sub\_agents.

You can do so by adding additional tools and sub\_agents to the root agent inside

``agents/data-science/data_science/agent.py``.

\* **\*\*Partial imports:\*\*** If you only need certain capabilities inside the multi-agent system,

e.g. just the data agent, you can import the data\_agent as an AgentTool into your own root agent.

\* **\*\*Model Selection:\*\*** Try different language models for both the top-level agent and the sub-agents to find the best performance for your data and queries.

## ## Troubleshooting

\* If you face `500 Internal Server Errors` when running the agent, simply re-run your last command.

That should fix the issue.

\* If you encounter issues with the code interpreter, review the logs to understand the errors. Make sure you're using base-64 encoding for files/images if interacting directly with a code interpreter extension instead of through the agent's helper functions.

\* If you see errors in the SQL generated, try the following:

- including clear descriptions in your tables and columns help boost performance

- if your database is large, try setting up a RAG pipeline for schema linking by storing your table schema details in a vector store

## ## Disclaimer

This agent sample is provided for illustrative purposes only and is not intended for production use. It serves as a basic example of an agent and a foundational starting point for individuals or teams to develop their own agents.

This sample has not been rigorously tested, may contain bugs or limitations,

and does not include features or optimizations typically required for a production environment (e.g., robust error handling, security measures, scalability, performance considerations, comprehensive logging, or advanced configuration options).

Users are solely responsible for any further development, testing, security hardening, and deployment of agents based on this sample. We recommend thorough review, testing, and the implementation of appropriate safeguards before using any derived agent in a live or critical system.

```
=====

File: pyproject.toml

=====

[tool.poetry]

name = "data-science"

version = "0.1"

description = "Data Science and Data QnA Multi-Agent"

authors = ["Meltem Subasioglu <msubasioglu@google.com>"]

license = "Apache License 2.0"

readme = "README.md"

package-mode = true


[tool.poetry.dependencies]

python = "^3.12"

python-dotenv = "^1.0.1"

google-adk = "^1.0.0"

immutabledict = "^4.2.1"

sqlglot = "^26.10.1"

db-dtypes = "^1.4.2"
```

```
regex = "^2024.11.6"

tabulate = "^0.9.0"

google-cloud-aiplatform = { extras = [

    "adk",

    "agent-engines",

], version = "^1.93.0" }

absl-py = "^2.2.2"

pydantic = "^2.11.3"
```

```
[tool.poetry.group.dev.dependencies]

google-cloud-aiplatform = { extras = [

    "adk",

    "agent-engines",

    "evaluation",

], version = "^1.93.0" }

pytest = "^8.3.5"

pytest-asyncio = "^0.26.0"
```

```
[tool.pytest.ini_options]

console_output_style = "progress"

addopts = "-vv -s"

#addopts = "-vv -s --pdb"

testpaths = ["tests/"]

log_level = "DEBUG"

log_cli = true

log_auto_indent = true
```

```

log_cli_date_format = "%Y-%m-%d %H:%M:%S"

log_cli_format = "[%{asctime)s] %{levelname}s (%{funcName}s) \t [%{pathname}s:
%(lineno)d] %(message)s"

filterwarnings = [

    # "error",

    "ignore::UserWarning",

    # note the use of single quote below to denote "raw" strings in TOML

    #'ignore:function ham\(\) is deprecated:DeprecationWarning',

]

```

```

[build-system]

requires = ["poetry-core"]

build-backend = "poetry.core.masonry.api"

```

```

=====

File: .env.example

=====

# Copy as .env file and fill your values below

# Run ./update_dotenv_example.sh to update .env-example from your .env file.


# Choose Model Backend: 0 -> ML Dev, 1 -> Vertex

GOOGLE_GENAI_USE_VERTEXAI=1


# ML Dev backend config. Fill if using ML Dev backend.

GOOGLE_API_KEY=YOUR_VALUE_HERE

```

```
# Vertex backend config

GOOGLE_CLOUD_PROJECT=YOUR_VALUE_HERE

GOOGLE_CLOUD_LOCATION=YOUR_VALUE_HERE


# SQLGen method

NL2SQL_METHOD="BASELINE" # BASELINE or CHASE


# Set up BigQuery Agent

BQ_PROJECT_ID=YOUR_VALUE_HERE

BQ_DATASET_ID='forecasting_sticker_sales'


# Set up RAG Corpus for BQML Agent

BQML_RAG_CORPUS_NAME='' # Leave this empty as it will be
populated automatically


# Set up Code Interpreter, if it exists. Else leave empty

CODE_INTERPRETER_EXTENSION_NAME='' # Either '' or 'projects/
{GOOGLE_CLOUD_PROJECT}/locations/us-central1/extensions/{EXTENSION_ID}'


# Models used in Agents

ROOT_AGENT_MODEL='gemini-2.0-flash-001'

ANALYTICS_AGENT_MODEL='gemini-2.0-flash-001'

BIGQUERY_AGENT_MODEL='gemini-2.0-flash-001'

BASELINE_NL2SQL_MODEL='gemini-2.0-flash-001'

CHASE_NL2SQL_MODEL='gemini-2.0-flash-001'

BQML_AGENT_MODEL='gemini-2.0-flash-001'
```

=====

File: data\_science/\_\_init\_\_.py

=====

# Copyright 2025 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");

# you may not use this file except in compliance with the License.

# You may obtain a copy of the License at

#

# <http://www.apache.org/licenses/LICENSE-2.0>

#

# Unless required by applicable law or agreed to in writing, software

# distributed under the License is distributed on an "AS IS" BASIS,

# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

# See the License for the specific language governing permissions and

# limitations under the License.

import os

from . import agent

\_\_all\_\_ = ["agent"]

=====

File: data\_science/agent.py

=====

# Copyright 2025 Google LLC

```

#

# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at

#
#     http://www.apache.org/licenses/LICENSE-2.0
#

# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""Top level agent for data agent multi-agents.

-- it get data from database (e.g., BQ) using NL2SQL
-- then, it use NL2Py to do further data analysis as needed
"""

import os

from datetime import date

from google.genai import types

from google.adk.agents import Agent

from google.adk.agents.callback_context import CallbackContext

from google.adk.tools import load_artifacts

from .sub_agents import bqml_agent

from .sub_agents.bigquery.tools import (

```



```

        get_database_settings as get_bq_database_settings,
    )

    from .prompts import return_instructions_root

    from .tools import call_db_agent, call_ds_agent

    date_today = date.today()

def setup_before_agent_call(callback_context: CallbackContext):

    """Setup the agent."""

    # setting up database settings in session.state

    if "database_settings" not in callback_context.state:

        db_settings = dict()

        db_settings["use_database"] = "BigQuery"

        callback_context.state["all_db_settings"] = db_settings

    # setting up schema in instruction

    if callback_context.state["all_db_settings"]["use_database"] ==
    "BigQuery":

        callback_context.state["database_settings"] =
        get_bq_database_settings()

        schema = callback_context.state["database_settings"]["bq_ddl_schema"]

        callback_context._invocation_context.agent.instruction = (

            return_instructions_root()

            + f"""

            ----- The BigQuery schema of the relevant data with a few sample rows.
            -----

```

```

        {schema}

        """

    )

root_agent = Agent(

    model=os.getenv("ROOT_AGENT_MODEL"),

    name="db_ds_multiagent",

    instruction=return_instructions_root(),

    global_instruction=(

        f"""

        You are a Data Science and Data Analytics Multi Agent System.

        Todays date: {date_today}

        """

    ),

    sub_agents=[bqml_agent],

    tools=[

        call_db_agent,

        call_ds_agent,

        load_artifacts,

    ],

    before_agent_callback=setup_before_agent_call,

    generate_content_config=types.GenerateContentConfig(temperature=0.01),

)

```

```
=====

File: data_science/prompts.py

=====

# Copyright 2025 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at

#

#     http://www.apache.org/licenses/LICENSE-2.0

#

# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""Module for storing and retrieving agent instructions.

This module defines functions that return instruction prompts for the root
agent.

These instructions guide the agent's behavior, workflow, and tool usage.

"""

def return_instructions_root() -> str:

    instruction_prompt_root_v2 = """
```

You are a senior data scientist tasked to accurately classify the user's intent regarding a specific database and formulate specific questions about the database suitable for a SQL database agent (`call\_db\_agent`) and a Python data science agent (`call\_ds\_agent`), if necessary.

- The data agents have access to the database specified below.
- If the user asks questions that can be answered directly from the database schema, answer it directly without calling any additional agents.
- If the question is a compound question that goes beyond database access, such as performing data analysis or predictive modeling, rewrite the question into two parts: 1) that needs SQL execution and 2) that needs Python analysis. Call the database agent and/or the datascience agent as needed.
- If the question needs SQL executions, forward it to the database agent.
- If the question needs SQL execution and additional analysis, forward it to the database agent and the datascience agent.
- If the user specifically wants to work on BQML, route to the bqml\_agent.
- IMPORTANT: be precise! If the user asks for a dataset, provide the name. Don't call any additional agent if not absolutely necessary!

<TASK>

# \*\*Workflow:\*\*

# 1. \*\*Understand Intent

# 2. \*\*Retrieve Data TOOL (`call\_db\_agent` - if applicable):\*\* If you need to query the database, use this tool. Make sure to provide a proper query to it to fulfill the task.

# 3. \*\*Analyze Data TOOL (`call\_ds\_agent` - if applicable):\*\* If you need to run data science tasks and python analysis, use this tool. Make sure to provide a proper query to it to fulfill the task.

# 4a. \*\*BigQuery ML Tool (`call\_bqml\_agent` - if applicable):\*\* If the user specifically asks (!) for BigQuery ML, use this tool. Make sure to provide a proper query to it to fulfill the task, along with the dataset and

project ID, and context.

# 5. **Respond:** Return ``RESULT`` AND ``EXPLANATION``, and optionally ``GRAPH`` if there are any. Please USE the MARKDOWN format (not JSON) with the following sections:

# \* **Result:** "Natural language summary of the data agent findings"

# \* **Explanation:** "Step-by-step explanation of how the result was derived.",

# **Tool Usage Summary:**

# \* **Greeting/Out of Scope:** answer directly.

# \* **SQL Query:** ``call_db_agent``. Once you return the answer, provide additional explanations.

# \* **SQL & Python Analysis:** ``call_db_agent``, then ``call_ds_agent``. Once you return the answer, provide additional explanations.

# \* **BQ ML** ``call_bqml_agent``: Query the BQ ML Agent if the user asks for it. Ensure that:

# A. You provide the fitting query.

# B. You pass the project and dataset ID.

# C. You pass any additional context.

**Key Reminder:**

\* You do have access to the database schema! Do not ask the db agent about the schema, use your own information first!! \*

\* **Never** generate SQL code. That is not your task. Use tools instead.

\* **ONLY CALL THE BQML AGENT IF THE USER SPECIFICALLY ASKS FOR BQML / BIGQUERY ML.** This can be for any BQML related tasks, like checking models, training, inference, etc.\*

\* **DO NOT** generate python code, ALWAYS USE `call_ds_agent` to generate

further analysis if needed.\*\*

\* \*\*DO NOT generate SQL code, ALWAYS USE call\_db\_agent to generate the SQL if needed.\*\*

\* \*\*IF call\_ds\_agent is called with valid result, JUST SUMMARIZE ALL RESULTS FROM PREVIOUS STEPS USING RESPONSE FORMAT!\*\*

\* \*\*IF data is available from previous call\_db\_agent and call\_ds\_agent, YOU CAN DIRECTLY USE call\_ds\_agent TO DO NEW ANALYZE USING THE DATA FROM PREVIOUS STEPS\*\*

\* \*\*DO NOT ask the user for project or dataset ID. You have these details in the session context. For BQ ML tasks, just verify if it is okay to proceed with the plan.\*\*

</TASK>

<CONSTRAINTS>

\* \*\*Schema Adherence:\*\* \* \*\*Strictly adhere to the provided schema.\*\* Do not invent or assume any data or schema elements beyond what is given.

\* \*\*Prioritize Clarity:\*\* \* If the user's intent is too broad or vague (e.g., asks about "the data" without specifics), prioritize the \*\*Greeting/Capabilities\*\* response and provide a clear description of the available data based on the schema.

</CONSTRAINTS>

"""

instruction\_prompt\_root\_v1 = """You are an AI assistant answering data-related questions using provided tools.

Your task is to accurately classify the user's intent and formulate refined questions suitable for:

- a SQL database agent (`call\_db\_agent`)
- a Python data science agent (`call\_ds\_agent`) and
- a BigQuery ML agent (`call\_bqml\_agent`), if necessary.

# \*\*Workflow:\*\*

# 1. **Understand Intent TOOL** (`call\_intent\_understanding`):\*\* This tool classifies the user question and returns a JSON with one of four structures:

# \* **Greeting:** Contains a `greeting\_message`. Return this message directly.

# \* **Use Database:** (optional) Contains a `use\_database`. Use this to determine which database to use. Return we switch to XXX database.

# \* **Out of Scope:** Return: "Your question is outside the scope of this database. Please ask a question relevant to this database."

# \* **SQL Query Only:** Contains `nl\_to\_sql\_question`. Proceed to Step 2.

# \* **SQL and Python Analysis:** Contains `nl\_to\_sql\_question` and `nl\_to\_python\_question`. Proceed to Step 2.

# 2. **Retrieve Data TOOL** (`call\_db\_agent` - if applicable):\*\* If you need to query the database, use this tool. Make sure to provide a proper query to it to fulfill the task.

# 3. **Analyze Data TOOL** (`call\_ds\_agent` - if applicable):\*\* If you need to run data science tasks and python analysis, use this tool. Make sure to provide a proper query to it to fulfill the task.

# 4a. **BigQuery ML Tool** (`call\_bqml\_agent` - if applicable):\*\* If the user specifically asks (!) for BigQuery ML, use this tool. Make sure to provide a proper query to it to fulfill the task, along with the dataset and project ID, and context.

# 5. **Respond:** Return `RESULT` AND `EXPLANATION`, and optionally `GRAPH` if there are any. Please USE the MARKDOWN format (not JSON) with the following sections:

# \* **Result:** "Natural language summary of the data agent findings"

# \* **Explanation:** "Step-by-step explanation of how the result was

derived.",

# \*\*Tool Usage Summary:\*\*

# \* \*\*Greeting/Out of Scope:\*\* answer directly.

# \* \*\*SQL Query:\*\* `call\_db\_agent`. Once you return the answer, provide additional explanations.

# \* \*\*SQL & Python Analysis:\*\* `call\_db\_agent`, then `call\_ds\_agent`. Once you return the answer, provide additional explanations.

# \* \*\*BQ ML `call\_bqml\_agent`:\*\* Query the BQ ML Agent if the user asks for it. Ensure that:

# A. You provide the fitting query.

# B. You pass the project and dataset ID.

# C. You pass any additional context.

\*\*Key Reminder:\*\*

\* \*\* You do have access to the database schema. Use it. \*\*

\* \*\*ONLY CALL THE BQML AGENT IF THE USER SPECIFICALLY ASKS FOR BQML / BIGQUERY ML. This can be for any BQML related tasks, like checking models, training, inference, etc.\*\*

\* \*\*DO NOT generate python code, ALWAYS USE call\_ds\_agent to generate further analysis if needed.\*\*

\* \*\*DO NOT generate SQL code, ALWAYS USE call\_db\_agent to generate the SQL if needed.\*\*

\* \*\*IF call\_ds\_agent is called with valid result, JUST SUMMARIZE ALL RESULTS FROM PREVIOUS STEPS USING RESPONSE FORMAT!\*\*

\* \*\*IF data is available from previous call\_db\_agent and call\_ds\_agent, YOU CAN DIRECTLY USE call\_ds\_agent TO DO NEW ANALYZE USING THE DATA FROM PREVIOUS STEPS, skipping call\_intent\_understanding and call\_db\_agent!\*\*

\* \*\*DO NOT ask the user for project or dataset ID. You have these details in the session context. For BQ ML tasks, just verify if it is okay to proceed with the plan.\*\*

"""



```
instruction_prompt_root_v0 = """You are an AI assistant answering data-
related questions using provided tools.
```

```
    **Workflow:**
```

```
    1. **Understand Intent TOOL (`call_intent_understanding`):** This
    tool classifies the user question and returns a JSON with one of four
    structures:
```

```
        * **Greeting:** Contains a `greeting_message`. Return this message
        directly.
```

```
        * **Use Database:** (optional) Contains a `use_database`. Use this
        to determine which database to use. Return we switch to XXX database.
```

```
        * **Out of Scope:** Return: "Your question is outside the scope
        of this database. Please ask a question relevant to this database."
```

```
        * **SQL Query Only:** Contains `nl_to_sql_question`. Proceed to
        Step 2.
```

```
        * **SQL and Python Analysis:** Contains `nl_to_sql_question` and
        `nl_to_python_question`. Proceed to Step 2.
```

```
    2. **Retrieve Data TOOL (`call_db_agent` - if applicable):** If you
    need to query the database, use this tool. Make sure to provide a proper query
    to it to fulfill the task.
```

```
    3. **Analyze Data TOOL (`call_ds_agent` - if applicable):** If you
    need to run data science tasks and python analysis, use this tool. Make sure
    to provide a proper query to it to fulfill the task.
```

```
    4a. **BigQuery ML Tool (`call_bqml_agent` - if applicable):** If the
    user specifically asks (!) for BigQuery ML, use this tool. Make sure to
    provide a proper query to it to fulfill the task, along with the dataset and
    project ID, and context. Once this is done, check back the plan with the user
    before proceeding.
```

```
        If the user accepts the plan, call this tool again so it can
        execute.
```

5. **Respond:** Return ``RESULT`` AND ``EXPLANATION``, and optionally ``GRAPH`` if there are any. Please USE the MARKDOWN format (not JSON) with the following sections:

**Result:** "Natural language summary of the data agent findings"

**Explanation:** "Step-by-step explanation of how the result was derived.",

**Tool Usage Summary:**

**Greeting/Out of Scope:** answer directly.

**SQL Query:** ``call_db_agent``. Once you return the answer, provide additional explanations.

**SQL & Python Analysis:** ``call_db_agent``, then ``call_ds_agent``. Once you return the answer, provide additional explanations.

**BQ ML ``call_bqml_agent``:** Query the BQ ML Agent if the user asks for it. Ensure that:

A. You provide the fitting query.

B. You pass the project and dataset ID.

C. You pass any additional context.

**Key Reminder:**

**Do not fabricate any answers. Rely solely on the provided tools. ALWAYS USE `call_intent_understanding` FIRST!**

**DO NOT generate python code, ALWAYS USE `call_ds_agent` to generate further analysis if `nl_to_python_question` is not N/A!**

**IF `call_ds_agent` is called with valid result, JUST SUMMARIZE ALL RESULTS FROM PREVIOUS STEPS USING RESPONSE FORMAT!**

**IF data is available from previous `call_db_agent` and `call_ds_agent`, YOU CAN DIRECTLY USE `call_ds_agent` TO DO NEW ANALYZE USING THE**

```
DATA FROM PREVIOUS STEPS, skipping call_intent_understanding and
call_db_agent!**
```

```
    * **Never generate answers directly; For any question,always USING THE
    GIVEN TOOLS. Start with call_intent_understanding if not sure!**
```

```
    """
```

```
    return instruction_prompt_root_v2
```

```
=====
```

```
File: data_science/tools.py
```

```
=====
```

```
# Copyright 2025 Google LLC
```

```
#
```

```
# Licensed under the Apache License, Version 2.0 (the "License");
```

```
# you may not use this file except in compliance with the License.
```

```
# You may obtain a copy of the License at
```

```
#
```

```
# http://www.apache.org/licenses/LICENSE-2.0
```

```
#
```

```
# Unless required by applicable law or agreed to in writing, software
```

```
# distributed under the License is distributed on an "AS IS" BASIS,
```

```
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
```

```
# See the License for the specific language governing permissions and
```

```
# limitations under the License.
```

```
"""Top level agent for data agent multi-agents.
```

```
-- it get data from database (e.g., BQ) using NL2SQL
```

```

-- then, it use NL2Py to do further data analysis as needed

"""

from google.adk.tools import ToolContext

from google.adk.tools.agent_tool import AgentTool


from .sub_agents import ds_agent, db_agent


async def call_db_agent(

    question: str,

    tool_context: ToolContext,

):

    """Tool to call database (nl2sql) agent."""

    print(

        "\n call_db_agent.use_database:"

        f' {tool_context.state["all_db_settings"]["use_database"]}'

    )

    agent_tool = AgentTool(agent=db_agent)

    db_agent_output = await agent_tool.run_async(

        args={"request": question}, tool_context=tool_context

    )

    tool_context.state["db_agent_output"] = db_agent_output

    return db_agent_output

```

```

async def call_ds_agent(

    question: str,

    tool_context: ToolContext,

):

    """Tool to call data science (nl2py) agent."""

    if question == "N/A":

        return tool_context.state["db_agent_output"]

    input_data = tool_context.state["query_result"]

    question_with_data = f"""

    Question to answer: {question}

    Actual data to analyze prevoius quesiton is already in the following:

    {input_data}

    """

    agent_tool = AgentTool(agent=ds_agent)

    ds_agent_output = await agent_tool.run_async(

        args={"request": question_with_data, tool_context=tool_context

    )

    tool_context.state["ds_agent_output"] = ds_agent_output

    return ds_agent_output

```

=====

File: data\_science/sub\_agents/\_\_init\_\_.py

=====

# Copyright 2025 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");

# you may not use this file except in compliance with the License.

# You may obtain a copy of the License at

#

# <http://www.apache.org/licenses/LICENSE-2.0>

#

# Unless required by applicable law or agreed to in writing, software

# distributed under the License is distributed on an "AS IS" BASIS,

# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

# See the License for the specific language governing permissions and

# limitations under the License.

from .bqml.agent import root\_agent as bqml\_agent

from .analytics.agent import root\_agent as ds\_agent

from .bigquery.agent import database\_agent as db\_agent

\_\_all\_\_ = ["bqml\_agent", "ds\_agent", "db\_agent"]

=====

File: data\_science/sub\_agents/analytics/\_\_init\_\_.py

=====

```
# Copyright 2025 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at

#

#     http://www.apache.org/licenses/LICENSE-2.0

#

# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

=====

File: data\_science/sub\_agents/analytics/agent.py

=====

```
# Copyright 2025 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at

#

#     http://www.apache.org/licenses/LICENSE-2.0

#
```

```
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

```
"""Data Science Agent V2: generate nl2py and use code interpreter to run the
code."""
```

```
import os
```

```
from google.adk.code_executors import VertexAiCodeExecutor
```

```
from google.adk.agents import Agent
```

```
from .prompts import return_instructions_ds
```

```
root_agent = Agent(
    model=os.getenv("ANALYTICS_AGENT_MODEL"),
    name="data_science_agent",
    instruction=return_instructions_ds(),
    code_executor=VertexAiCodeExecutor(
        optimize_data_file=True,
        stateful=True,
    ),
)
```

```
=====
```

```
File: data_science/sub_agents/analytics/prompts.py
```

```
=====
```



```

# Copyright 2025 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""Module for storing and retrieving agent instructions.

This module defines functions that return instruction prompts for the
analytics (ds) agent.

These instructions guide the agent's behavior, workflow, and tool usage.
"""

def return_instructions_ds() -> str:

    instruction_prompt_ds_v1 = """

    # Guidelines

    **Objective:** Assist the user in achieving their data analysis goals within

```

the context of a Python Colab notebook, **\*\*with emphasis on avoiding assumptions and ensuring accuracy.\*\***

Reaching that goal can involve multiple steps. When you need to generate code, you **\*\*don't\*\*** need to solve the goal in one go. Only generate the next step at a time.

**\*\*Trustworthiness:\*\*** Always include the code in your response. Put it at the end in the section "Code:". This will ensure trust in your output.

**\*\*Code Execution:\*\*** All code snippets provided will be executed within the Colab environment.

**\*\*Statefulness:\*\*** All code snippets are executed and the variables stays in the environment. You NEVER need to re-initialize variables. You NEVER need to reload files. You NEVER need to re-import libraries.

**\*\*Imported Libraries:\*\*** The following libraries are ALREADY imported and should NEVER be imported again:

```
```tool_code
import io

import math

import re

import matplotlib.pyplot as plt

import numpy as np

import pandas as pd

import scipy
```
```

**\*\*Output Visibility:\*\*** Always print the output of code execution to visualize results, especially for data exploration and analysis. For example:

- To look at the shape of a pandas.DataFrame do:

```
```tool_code
```

```
print(df.shape)
```

```
...
```

The output will be presented to you as:

```
```tool_outputs
```

```
(49, 7)
```

```
...
```

- To display the result of a numerical computation:

```
```tool_code
```

```
x = 10 ** 9 - 12 ** 5
```

```
print(f'{{x=}}')
```

```
...
```

The output will be presented to you as:

```
```tool_outputs
```

```
x=999751168
```

```
...
```

- You **never** generate ```tool\_outputs yourself.
- You can then use this output to decide on next steps.
- Print variables (e.g., ``print(f'{{variable=}}')``).
- Give out the generated code under 'Code:'.

**\*\*No Assumptions:\*\*** **\*\*Crucially,** avoid making assumptions about the nature of the data or column names.**\*\*** Base findings solely on the data itself. Always use the information obtained from ``explore_df`` to guide your analysis.

**\*\*Available files:\*\*** Only use the files that are available as specified in the list of available files.

**\*\*Data in prompt:\*\*** Some queries contain the input data directly in the prompt. You have to parse that data into a pandas DataFrame. ALWAYS parse all

the data. NEVER edit the data that are given to you.

**\*\*Answerability:\*\*** Some queries may not be answerable with the available data. In those cases, inform the user why you cannot process their query and suggest what type of data would be needed to fulfill their request.

**\*\*WHEN YOU DO PREDICTION / MODEL FITTING, ALWAYS PLOT FITTED LINE AS WELL \*\***

#### TASK:

You need to assist the user with their queries by looking at the data and the context in the conversation.

Your final answer should summarize the code and code execution relevant to the user query.

You should include all pieces of data to answer the user query, such as the table from code execution results.

If you cannot answer the question directly, you should follow the guidelines above to generate the next step.

If the question can be answered directly with writing any code, you should do that.

If you don't have enough data to answer the question, you should ask for clarification from the user.

You should NEVER install any package on your own like ``pip install ...``.

When plotting trends, you should make sure to sort and order the data by the x-axis.

NOTE: for pandas `pandas.core.series.Series` object, you can use `.iloc[0]` to access the first element rather than assuming it has the integer index 0"

correct one: `predicted_value = prediction.predicted_mean.iloc[0]`

error one: `predicted_value = prediction.predicted_mean[0]`

correct one: `confidence_interval_lower = confidence_intervals.iloc[0, 0]`

error one: `confidence_interval_lower = confidence_intervals[0][0]`

```
"""
```

```
    return instruction_prompt_ds_v1
```

```
=====
```

```
File: data_science/sub_agents/bigquery/__init__.py
```

```
=====
```

```
# Copyright 2025 Google LLC
```

```
#
```

```
# Licensed under the Apache License, Version 2.0 (the "License");
```

```
# you may not use this file except in compliance with the License.
```

```
# You may obtain a copy of the License at
```

```
#
```

```
# http://www.apache.org/licenses/LICENSE-2.0
```

```
#
```

```
# Unless required by applicable law or agreed to in writing, software
```

```
# distributed under the License is distributed on an "AS IS" BASIS,
```

```
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
```

```
# See the License for the specific language governing permissions and
```

```
# limitations under the License.
```

```
=====
```

File: data\_science/sub\_agents/bigquery/agent.py

=====

# Copyright 2025 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");

# you may not use this file except in compliance with the License.

# You may obtain a copy of the License at

#

# <http://www.apache.org/licenses/LICENSE-2.0>

#

# Unless required by applicable law or agreed to in writing, software

# distributed under the License is distributed on an "AS IS" BASIS,

# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

# See the License for the specific language governing permissions and

# limitations under the License.

"""Database Agent: get data from database (BigQuery) using NL2SQL."""

import os

from google.adk.agents import Agent

from google.adk.agents.callback\_context import CallbackContext

from google.genai import types

from . import tools

from .chase\_sql import chase\_db\_tools

from .prompts import return\_instructions\_bigquery

NL2SQL\_METHOD = os.getenv("NL2SQL\_METHOD", "BASELINE")

```

def setup_before_agent_call(callback_context: CallbackContext) -> None:

    """Setup the agent."""

    if "database_settings" not in callback_context.state:

        callback_context.state["database_settings"] = \

            tools.get_database_settings()


database_agent = Agent(

    model=os.getenv("BIGQUERY_AGENT_MODEL"),

    name="database_agent",

    instruction=return_instructions_bigquery(),

    tools=[

        (

            chase_db_tools.initial_bq_nl2sql

            if NL2SQL_METHOD == "CHASE"

            else tools.initial_bq_nl2sql

        ),

        tools.run_bigquery_validation,

    ],

    before_agent_callback=setup_before_agent_call,

    generate_content_config=types.GenerateContentConfig(temperature=0.01),

)

```

```
=====

File: data_science/sub_agents/bigquery/prompts.py

=====

# Copyright 2025 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at

#

#     http://www.apache.org/licenses/LICENSE-2.0

#

# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""Module for storing and retrieving agent instructions.

This module defines functions that return instruction prompts for the bigquery
agent.

These instructions guide the agent's behavior, workflow, and tool usage.

"""

import os

def return_instructions_bigquery() -> str:
```



```

NL2SQL_METHOD = os.getenv("NL2SQL_METHOD", "BASELINE")

if NL2SQL_METHOD == "BASELINE" or NL2SQL_METHOD == "CHASE":

    db_tool_name = "initial_bq_nl2sql"

else:

    db_tool_name = None

    raise ValueError(f"Unknown NL2SQL method: {NL2SQL_METHOD}")


instruction_prompt_bqml_v1 = f"""

You are an AI assistant serving as a SQL expert for BigQuery.

Your job is to help users generate SQL answers from natural language
questions (inside Nl2sqlInput).

You should proeuce the result as NL2SQLOutput.


Use the provided tools to help generate the most accurate SQL:


1. First, use {db_tool_name} tool to generate initial SQL from the
question.


2. You should also validate the SQL you have created for syntax and
function errors (Use run_bigquery_validation tool). If there are any errors,
you should go back and address the error in the SQL. Recreate the SQL based by
addressing the error.


4. Generate the final result in JSON format with four keys: "explain",
"sql", "sql_results", "nl_results".


    "explain": "write out step-by-step reasoning to explain how you are
generating the query based on the schema, example, and question.",

    "sql": "Output your generated SQL!",

    "sql_results": "raw sql execution query_result from
run_bigquery_validation if it's available, otherwise None",

    "nl_results": "Natural language about results, otherwise it's None
if generated SQL is invalid"

    ...


You should pass one tool call to another tool call as needed!


NOTE: you should ALWAYS USE THE TOOLS ({db_tool_name} AND
run_bigquery_validation) to generate SQL, not make up SQL WITHOUT CALLING

```

TOOLS.

Keep in mind that you are an orchestration agent, not a SQL expert, so use the tools to help you generate SQL, but do not make up SQL.

"""

return instruction\_prompt\_bqml\_v1

=====

File: data\_science/sub\_agents/bigquery/tools.py

=====

# Copyright 2025 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");

# you may not use this file except in compliance with the License.

# You may obtain a copy of the License at

#

# <http://www.apache.org/licenses/LICENSE-2.0>

#

# Unless required by applicable law or agreed to in writing, software

# distributed under the License is distributed on an "AS IS" BASIS,

# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

# See the License for the specific language governing permissions and

# limitations under the License.

"""This file contains the tools used by the database agent."""

```
import datetime

import logging

import os

import re


from data_science.utils.utils import get_env_var

from google.adk.tools import ToolContext

from google.cloud import bigquery

from google.genai import Client


from .chase_sql import chase_constants


# Assume that `BQ_PROJECT_ID` is set in the environment. See the
# `data_agent` README for more details.

project = os.getenv("BQ_PROJECT_ID", None)

location = os.getenv("GOOGLE_CLOUD_LOCATION", "us-central1")

llm_client = Client(vertexai=True, project=project, location=location)


MAX_NUM_ROWS = 80


database_settings = None

bq_client = None


def get_bq_client():

    """Get BigQuery client."""

    global bq_client

    if bq_client is None:
```

```

        bq_client = bigquery.Client(project=get_env_var("BQ_PROJECT_ID"))

    return bq_client

def get_database_settings():

    """Get database settings."""

    global database_settings

    if database_settings is None:

        database_settings = update_database_settings()

    return database_settings

def update_database_settings():

    """Update database settings."""

    global database_settings

    ddl_schema = get_bigquery_schema(

        get_env_var("BQ_DATASET_ID"),

        client=get_bq_client(),

        project_id=get_env_var("BQ_PROJECT_ID"),

    )

    database_settings = {

        "bq_project_id": get_env_var("BQ_PROJECT_ID"),

        "bq_dataset_id": get_env_var("BQ_DATASET_ID"),

        "bq_ddl_schema": ddl_schema,

        # Include ChaseSQL-specific constants.

        **chase_constants.chase_sql_constants_dict,

    }

    return database_settings

```

```

def get_bigquery_schema(dataset_id, client=None, project_id=None):

    """Retrieves schema and generates DDL with example values for a BigQuery
    dataset.

    Args:

        dataset_id (str): The ID of the BigQuery dataset (e.g., 'my_dataset').

        client (bigquery.Client): A BigQuery client.

        project_id (str): The ID of your Google Cloud Project.

    Returns:

        str: A string containing the generated DDL statements.

    """

    if client is None:

        client = bigquery.Client(project=project_id)

    # dataset_ref = client.dataset(dataset_id)

    dataset_ref = bigquery.DatasetReference(project_id, dataset_id)

    ddl_statements = ""

    for table in client.list_tables(dataset_ref):

        table_ref = dataset_ref.table(table.table_id)

        table_obj = client.get_table(table_ref)

        # Check if table is a view

        if table_obj.table_type != "TABLE":

```

```

        continue

ddl_statement = f"CREATE OR REPLACE TABLE `{table_ref}` (\n"

for field in table_obj.schema:

    ddl_statement += f"    `{field.name}` {field.field_type}"

    if field.mode == "REPEATED":

        ddl_statement += " ARRAY"

    if field.description:

        ddl_statement += f" COMMENT '{field.description}'"

    ddl_statement += ",\n"

ddl_statement = ddl_statement[:-2] + "\n);\n\n"

# Add example values if available (limited to first row)
rows = client.list_rows(table_ref, max_results=5).to_dataframe()

if not rows.empty:

    ddl_statement += f"-- Example values for table `{table_ref}`:\n"

    for _, row in rows.iterrows(): # Iterate over DataFrame rows

        ddl_statement += f"INSERT INTO `{table_ref}` VALUES\n"

        example_row_str = "("

        for value in row.values: # Now row is a pandas Series and has
values

            if isinstance(value, str):

                example_row_str += f"'{value}',"

            elif value is None:

                example_row_str += "NULL,"

            else:

                example_row_str += f"{value},"

```

```

        example_row_str = (

            example_row_str[:-1] + ");\n\n"

        ) # remove trailing comma

        ddl_statement += example_row_str

    ddl_statements += ddl_statement

return ddl_statements

def initial_bq_nl2sql(

    question: str,

    tool_context: ToolContext,

) -> str:

    """Generates an initial SQL query from a natural language question.

    Args:

        question (str): Natural language question.

        tool_context (ToolContext): The tool context to use for generating the
SQL

        query.

    Returns:

        str: An SQL statement to answer this question.

    """

    prompt_template = """

You are a BigQuery SQL expert tasked with answering user's questions about
BigQuery tables by generating SQL queries in the GoogleSql dialect. Your task
is to write a Bigquery SQL query that answers the following question while
using the provided context.
```

**\*\*Guidelines:\*\***

- **\*\*Table Referencing:\*\*** Always use the full table name with the database prefix in the SQL statement. Tables should be referred to using a fully qualified name with enclosed in backticks (``) e.g. ``project_name.dataset_name.table_name``. Table names are case sensitive.
- **\*\*Joins:\*\*** Join as few tables as possible. When joining tables, ensure all join columns are the same data type. Analyze the database and the table schema provided to understand the relationships between columns and tables.
- **\*\*Aggregations:\*\*** Use all non-aggregated columns from the ``SELECT`` statement in the ``GROUP BY`` clause.
- **\*\*SQL Syntax:\*\*** Return syntactically and semantically correct SQL for BigQuery with proper relation mapping (i.e., `project_id`, `owner`, `table`, and `column` relation). Use SQL ``AS`` statement to assign a new name temporarily to a table column or even a table wherever needed. Always enclose subqueries and union queries in parentheses.
- **\*\*Column Usage:\*\*** Use **\*ONLY\*** the column names (`column_name`) mentioned in the Table Schema. Do **\*NOT\*** use any other column names. Associate ``column_name`` mentioned in the Table Schema only to the ``table_name`` specified under Table Schema.
- **\*\*FILTERS:\*\*** You should write query effectively to reduce and minimize the total rows to be returned. For example, you can use filters (like ``WHERE``, ``HAVING``, etc. (like `'COUNT'`, `'SUM'`, etc.) in the SQL query.
- **\*\*LIMIT ROWS:\*\*** The maximum number of rows returned should be less than `{MAX_NUM_ROWS}`.

**\*\*Schema:\*\***

The database structure is defined by the following table schemas (possibly with sample rows):

...

{SCHEMA}

...

**\*\*Natural language question:\*\***



```
...
```

```
{QUESTION}
```

```
...
```

```
**Think Step-by-Step:** Carefully consider the schema, question, guidelines,  
and best practices outlined above to generate the correct BigQuery SQL.
```

```
"""
```

```
ddl_schema = tool_context.state["database_settings"]["bq_ddl_schema"]
```

```
prompt = prompt_template.format(
```

```
    MAX_NUM_ROWS=MAX_NUM_ROWS, SCHEMA=ddl_schema, QUESTION=question
```

```
)
```

```
response = llm_client.models.generate_content(
```

```
    model=os.getenv("BASELINE_NL2SQL_MODEL"),
```

```
    contents=prompt,
```

```
    config={"temperature": 0.1},
```

```
)
```

```
sql = response.text
```

```
if sql:
```

```
    sql = sql.replace("`sql", "").replace("`", "").strip()
```

```
print("\n sql:", sql)
```

```
tool_context.state["sql_query"] = sql
```

```
return sql
```

```
def run_bigquery_validation(  
    sql_string: str,  
    tool_context: ToolContext,  
    ) -> str:
```

```
    """Validates BigQuery SQL syntax and functionality.
```

```
    This function validates the provided SQL string by attempting to execute  
    it
```

```
    against BigQuery in dry-run mode. It performs the following checks:
```

1. **SQL Cleanup:** Preprocesses the SQL string using a ``cleanup_sql``  
function
2. **DML/DDL Restriction:** Rejects any SQL queries containing DML or DDL  
statements (e.g., `UPDATE`, `DELETE`, `INSERT`, `CREATE`, `ALTER`) to ensure  
read-only operations.
3. **Syntax and Execution:** Sends the cleaned SQL to BigQuery for  
validation.  
  
If the query is syntactically correct and executable, it retrieves the  
results.
4. **Result Analysis:** Checks if the query produced any results. If so,  
it  
  
formats the first few rows of the result set for inspection.

Args:

`sql_string (str)`: The SQL query string to validate.

`tool_context (ToolContext)`: The tool context to use for validation.

Returns:

```
    str: A message indicating the validation outcome. This includes:

        - "Valid SQL. Results: ..." if the query is valid and returns
data.
        - "Valid SQL. Query executed successfully (no results)." if the
query
        is valid but returns no data.

        - "Invalid SQL: ..." if the query is invalid, along with the
error
        message from BigQuery.

    """
```

```
def cleanup_sql(sql_string):
```

```
    """Processes the SQL string to get a printable, valid SQL string."""
```

```
    # 1. Remove backslashes escaping double quotes
```

```
    sql_string = sql_string.replace('\\"', '"')
```

```
    # 2. Remove backslashes before newlines (the key fix for this issue)
```

```
    sql_string = sql_string.replace("\\\\n", "\\n") # Corrected regex
```

```
    # 3. Replace escaped single quotes
```

```
    sql_string = sql_string.replace("\\'", "'")
```

```
    # 4. Replace escaped newlines (those not preceded by a backslash)
```

```
    sql_string = sql_string.replace("\\n", "\n")
```

```
    # 5. Add limit clause if not present
```

```
    if "limit" not in sql_string.lower():
```

```

        sql_string = sql_string + " limit " + str(MAX_NUM_ROWS)

    return sql_string

    logging.info("Validating SQL: %s", sql_string)

    sql_string = cleanup_sql(sql_string)

    logging.info("Validating SQL (after cleanup): %s", sql_string)

    final_result = {"query_result": None, "error_message": None}

    # More restrictive check for BigQuery - disallow DML and DDL

    if re.search(

        r"(?i)(update|delete|drop|insert|create|alter|truncate|merge)",
        sql_string

    ):

        final_result["error_message"] = (

            "Invalid SQL: Contains disallowed DML/DDL operations."

        )

    return final_result

try:

    query_job = get_bq_client().query(sql_string)

    results = query_job.result() # Get the query results

    if results.schema: # Check if query returned data

        rows = [

            {

                key: (

                    value

```

```

        if not isinstance(value, datetime.date)

        else value.strftime("%Y-%m-%d")

    )

    for (key, value) in row.items()

}

for row in results

][

    :MAX_NUM_ROWS

] # Convert BigQuery RowIterator to list of dicts

# return f"Valid SQL. Results: {rows}"

final_result["query_result"] = rows

tool_context.state["query_result"] = rows

else:

    final_result["error_message"] = (

        "Valid SQL. Query executed successfully (no results)."

    )

except (

    Exception

) as e: # Catch generic exceptions from BigQuery # pylint:
disable=broad-exception-caught

    final_result["error_message"] = f"Invalid SQL: {e}"

print("\n run_bigquery_validation final_result: \n", final_result)

return final_result

```

=====

File: data\_science/sub\_agents/bigquery/chase\_sql/\_\_init\_\_.py

=====

# Copyright 2025 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");

# you may not use this file except in compliance with the License.

# You may obtain a copy of the License at

#

# <http://www.apache.org/licenses/LICENSE-2.0>

#

# Unless required by applicable law or agreed to in writing, software

# distributed under the License is distributed on an "AS IS" BASIS,

# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

# See the License for the specific language governing permissions and

# limitations under the License.

=====

File: data\_science/sub\_agents/bigquery/chase\_sql/chase\_constants.py

=====

# Copyright 2025 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");

```
# you may not use this file except in compliance with the License.

# You may obtain a copy of the License at

#

#     http://www.apache.org/licenses/LICENSE-2.0

#

# Unless required by applicable law or agreed to in writing, software

# distributed under the License is distributed on an "AS IS" BASIS,

# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

# See the License for the specific language governing permissions and

# limitations under the License.
```

```
"""Constants used by the ChaseSQL algorithm."""
```

```
import os
```

```
from typing import Any
```

```
import immutabledict
```

```
# Parameters for ChaseSQL.
```

```
chase_sql_constants_dict: immutabledict.immutabledict[str, Any] = (
```

```
    immutabledict.immutabledict(
```

```
        {
```

```
            # Whether to transpile the SQL to BigQuery.
```

```
            "transpile_to_bigquery": True,
```

```
            # Whether to process input errors.
```

```
            "process_input_errors": True,
```

```
            # Whether to process SQLGlot tool output errors.
```

```
            "process_tool_output_errors": True,
```

```
            # Number of candidates to generate.
```

```
            "number_of_candidates": 1,
```

```
        # Model to use for generation.

        "model": os.getenv("CHASE_NL2SQL_MODEL"),

        # Temperature for generation.

        "temperature": 0.5,

        # Type of SQL generation method.

        "generate_sql_type": "dc",

    }

)

)
```

=====

File: data\_science/sub\_agents/bigquery/chase\_sql/chase\_db\_tools.py

=====

```
# Copyright 2025 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at

#

#     http://www.apache.org/licenses/LICENSE-2.0

#

# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```



```
"""This code contains the implementation of the tools used for the CHASE-SQL
agent."""
```

```
import enum
```

```
import os
```

```
from google.adk.tools import ToolContext
```

```
# pylint: disable=g-importing-member
```

```
from .dc_prompt_template import DC_PROMPT_TEMPLATE
```

```
from .llm_utils import GeminiModel
```

```
from .qp_prompt_template import QP_PROMPT_TEMPLATE
```

```
from .sql_postprocessor import sql_translator
```

```
# pylint: enable=g-importing-member
```

```
BQ_PROJECT_ID = os.getenv("BQ_PROJECT_ID")
```

```
class GenerateSQLType(enum.Enum):
```

```
    """Enum for the different types of SQL generation methods.
```

```
    DC: Divide and Conquer ICL prompting
```

```
    QP: Query Plan-based prompting
```

```
    """
```

```
    DC = "dc"
```

```
    QP = "qp"
```

```

def exception_wrapper(func):

    """A decorator to catch exceptions in a function and return the exception
    as a string.

    Args:

        func (callable): The function to wrap.

    Returns:

        callable: The wrapped function.

    """

    def wrapped_function(*args, **kwargs):

        try:

            return func(*args, **kwargs)

        except Exception as e: # pylint: disable=broad-exception-caught

            return f"Exception occurred in {func.__name__}: {str(e)}"

    return wrapped_function


def parse_response(response: str) -> str:

    """Parses the output to extract SQL content from the response.

    Args:

        response (str): The output string containing SQL query.

    Returns:

```

```

        str: The SQL query extracted from the response.
    """

    query = response

    try:

        if "`sql" in response and "`" in response:

            query = response.split("`sql")[1].split("`")[0]

    except ValueError as e:

        print(f"Error in parsing response: {e}")

        query = response

    return query.strip()


def initial_bq_nl2sql(

    question: str,

    tool_context: ToolContext,

) -> str:

    """Generates an initial SQL query from a natural language question.

    Args:

        question: Natural language question.

        tool_context: Function context.

    Returns:

        str: An SQL statement to answer this question.

    """

    print("***** Running agent with ChaseSQL algorithm.")

    ddl_schema = tool_context.state["database_settings"]["bq_ddl_schema"]

    project = tool_context.state["database_settings"]["bq_project_id"]

    db = tool_context.state["database_settings"]["bq_dataset_id"]

```

```

transpile_to_bigquery = tool_context.state["database_settings"][
    "transpile_to_bigquery"
]

process_input_errors = tool_context.state["database_settings"][
    "process_input_errors"
]

process_tool_output_errors = tool_context.state["database_settings"][
    "process_tool_output_errors"
]

number_of_candidates = tool_context.state["database_settings"][
    "number_of_candidates"
]

model = tool_context.state["database_settings"]["model"]

temperature = tool_context.state["database_settings"]["temperature"]

generate_sql_type = tool_context.state["database_settings"]
["generate_sql_type"]

if generate_sql_type == GenerateSQLType.DC.value:
    prompt = DC_PROMPT_TEMPLATE.format(
        SCHEMA=ddl_schema, QUESTION=question, BQ_PROJECT_ID=BQ_PROJECT_ID
    )

elif generate_sql_type == GenerateSQLType.QP.value:
    prompt = QP_PROMPT_TEMPLATE.format(
        SCHEMA=ddl_schema, QUESTION=question, BQ_PROJECT_ID=BQ_PROJECT_ID
    )

else:
    raise ValueError(f"Unsupported generate_sql_type:
{generate_sql_type}")

```

```

model = GeminiModel(model_name=model, temperature=temperature)

requests = [prompt for _ in range(number_of_candidates)]

responses = model.call_parallel(requests, parser_func=parse_response)

# Take just the first response.

responses = responses[0]


# If postprocessing of the SQL to transpile it to BigQuery is required,
# then do it here.

if transpile_to_bigquery:

    translator = sql_translator.SqlTranslator(

        model=model,

        temperature=temperature,

        process_input_errors=process_input_errors,

        process_tool_output_errors=process_tool_output_errors,

    )

    # pylint: disable=g-bad-todo

    # pylint: enable=g-bad-todo

    responses: str = translator.translate(

        responses, ddl_schema=ddl_schema, db=db, catalog=project

    )


return responses

```

=====

File: data\_science/sub\_agents/bigquery/chase\_sql/dc\_prompt\_template.py

=====

# Copyright 2025 Google LLC

```
#

# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

""""Divide-and-Conquer prompt template."""

DC_PROMPT_TEMPLATE = """"

You are an experienced database expert.

Now you need to generate a GoogleSQL or BigQuery query given the database
information, a question and some additional information.

The database structure is defined by table schemas (some columns provide
additional column descriptions in the options).

Given the table schema information description and the `Question`. You will be
given table creation statements and you need understand the database and
columns.

You will be using a way called "recursive divide-and-conquer approach to SQL
query generation from natural language".

Here is a high level description of the steps.

1. **Divide (Decompose Sub-question with Pseudo SQL):** The complex natural
```

language question is recursively broken down into simpler sub-questions. Each sub-question targets a specific piece of information or logic required for the final SQL query.

2. **\*\*Conquer (Real SQL for sub-questions):\*\*** For each sub-question (and the main question initially), a "pseudo-SQL" fragment is formulated. This pseudo-SQL represents the intended SQL logic but might have placeholders for answers to the decomposed sub-questions.

3. **\*\*Combine (Reassemble):\*\*** Once all sub-questions are resolved and their corresponding SQL fragments are generated, the process reverses. The SQL fragments are recursively combined by replacing the placeholders in the pseudo-SQL with the actual generated SQL from the lower levels.

4. **\*\*Final Output:\*\*** This bottom-up assembly culminates in the complete and correct SQL query that answers the original complex question.

Database admin instructions (please *\*unconditionally\** follow these instructions. Do *\*not\** ignore them or use them as hints.):

1. **\*\*SELECT Clause:\*\***

- Select only the necessary columns by explicitly specifying them in the `SELECT` statement. Avoid redundant columns or values.

2. **\*\*Aggregation (MAX/MIN):\*\***

- Ensure `JOIN`s are completed before applying `MAX()` or `MIN()`.

GoogleSQL supports similar syntax for aggregation functions, so use `MAX()` and `MIN()` as needed after `JOIN` operations.

3. **\*\*ORDER BY with Distinct Values:\*\***

- In GoogleSQL, `GROUP BY <column>` can be used before `ORDER BY <column> ASC|DESC` to get distinct values and sort them.

4. **\*\*Handling NULLs:\*\***

- To filter out NULL values, use `JOIN` or add a `WHERE <column> IS NOT NULL` clause.

5. **\*\*FROM/JOIN Clauses:\*\***

- Only include tables essential to the query. BigQuery supports `JOIN` types like `INNER JOIN`, `LEFT JOIN`, and `RIGHT JOIN`, so use these based on the relationships needed.

6. **\*\*Strictly Follow Hints:\*\***

- Carefully adhere to any specified conditions in the instructions for precise query construction.

7. **\*\*Thorough Question Analysis:\*\***

- Review all specified conditions or constraints in the question to ensure they are fully addressed in the query.

8. **\*\*DISTINCT Keyword:\*\***

- Use ``SELECT DISTINCT`` when unique values are needed, such as for IDs or URLs.

9. **\*\*Column Selection:\*\***

- Pay close attention to column descriptions and any hints to select the correct column, especially when similar columns exist across tables.

10. **\*\*String Concatenation:\*\***

- GoogleSQL uses ``CONCAT()`` for string concatenation. Avoid using ``||`` and instead use ``CONCAT(column1, ' ', column2)`` for concatenation.

11. **\*\*JOIN Preference:\*\***

- Use ``INNER JOIN`` when appropriate, and avoid nested ``SELECT`` statements if a ``JOIN`` will achieve the same result.

12. **\*\*GoogleSQL Functions Only:\*\***

- Use functions available in GoogleSQL. Avoid SQLite-specific functions and replace them with GoogleSQL equivalents (e.g., ``FORMAT_DATE`` instead of ``STRFTIME``).

13. **\*\*Date Processing:\*\***

- GoogleSQL supports ``FORMAT_DATE('%Y', date_column)`` for extracting the year. Use date functions like ``FORMAT_DATE``, ``DATE_SUB``, and ``DATE_DIFF`` for date manipulation.



#### 14. **\*\*Table Names and reference:\*\***

- As required by BigQuery, always use the full table name with the database prefix in the SQL statement. For example, "SELECT \* FROM example\_bigquery\_database.table\_a", not just "SELECT \* FROM table\_a"

#### 15. **\*\*GROUP BY or AGGREGATE:\*\***

- In queries with GROUP BY, all columns in the SELECT list must either: Be included in the GROUP BY clause, or Be used in an aggregate function (e.g., MAX, MIN, AVG, COUNT, SUM).

Here are some examples

=====

Example 1

\*\*\*\*\*

**【Table creation statements】**

```
CREATE TABLE {BQ_PROJECT_ID}.restaurant.generalinfo
```

```
(
```

```
  id_restaurant INT64,
```

```
  food_type STRING OPTIONS(description="the food type"),
```

```
  city STRING OPTIONS(description="the city where the restaurant is located in"),
```

```
);
```

```
CREATE TABLE {BQ_PROJECT_ID}.restaurant.location
```

```
(
```

```
  id_restaurant INT64,
```

```
  street_name STRING OPTIONS(description="the street name of the restaurant"),
```

```
  city STRING OPTIONS(description="the city where the restaurant is located in  
foreign key (id_restaurant) references generalinfo (id_restaurant) on update  
cascade on delete cascade"),
```

);

\*\*\*\*\*

### 【Question】

Question:

How many Thai restaurants can be found in San Pablo Ave, Albany? Thai restaurant refers to food\_type = 'thai'; San Pablo Ave Albany refers to street\_name = 'san pablo ave' AND T1.city = 'albany'

\*\*\*\*\*

### 【Answer】

Repeating the question and generating the SQL with Recursive Divide-and-Conquer.

**\*\*Question\*\***: How many Thai restaurants can be found in San Pablo Ave, Albany? Thai restaurant refers to food\_type = 'thai'; San Pablo Ave Albany refers to street\_name = 'san pablo ave' AND T1.city = 'albany'

**\*\*1. Divide and Conquer:\*\***

**\* \*\*Main Question:\*\*** How many Thai restaurants can be found in San Pablo Ave, Albany?

**\* \*\*Analysis:\*\*** The question asks for a count of restaurants, so we'll use 'COUNT()' for that. The count should include only Thai restaurants, which we can identify using the 'food\_type' column in the 'restaurant.generalinfo' table. The location "San Pablo Ave, Albany" spans two columns ('street\_name' and 'city') in the 'restaurant.location' table, requiring us to join these two tables.

**\* \*\*Pseudo SQL:\*\*** SELECT COUNT('T1`.`id\_restaurant`) FROM 'restaurantgeneralinfo' AS 'T1' INNER JOIN 'restaurant.location' AS 'T2' ON 'T1`.`id\_restaurant' = 'T2`.`id\_restaurant' WHERE <Thai restaurant> AND <in San Pablo Ave, Albany>

**\* \*\*Sub-question 1:\*\*** Thai restaurant

**\* \*\*Analysis:\*\*** This is a straightforward filter on the 'restaurant.generalinfo' table using the 'food\_type' column.

**\* \*\*Pseudo SQL:\*\*** 'T1`.`food\_type' = 'thai'

\* \*\*Sub-question 2:\*\* in San Pablo Ave, Albany

\* \*\*Analysis:\*\* This location information is spread across two columns in the `restaurant.location` table. We need to combine these conditions with an "AND" operator to ensure both are met.

\* \*\*Pseudo SQL:\*\* `T2`.`street\_name` = 'san pablo ave' AND `T2`.`city` = 'albany'

\*\*2. Assembling SQL:\*\*

\* \*\*Sub-question 1 (Thai restaurant):\*\*

\* \*\*SQL:\*\* `T1`.`food\_type` = 'thai'

\* \*\*Sub-question 2 (in San Pablo Ave, Albany):\*\*

\* \*\*SQL:\*\* `T2`.`street\_name` = 'san pablo ave' AND `T2`.`city` = 'albany'

\* \*\*Main Question (count of restaurants):\*\*

\* \*\*SQL:\*\*  
SELECT COUNT(`T1`.`id\_restaurant`) FROM  
{BQ\_PROJECT\_ID}.restaurant.generalinfo AS `T1` INNER JOIN  
{BQ\_PROJECT\_ID}.restaurant.location AS `T2` ON `T1`.`id\_restaurant` =  
`T2`.`id\_restaurant` WHERE `T1`.`food\_type` = 'thai' AND `T2`.`street\_name` =  
'san pablo ave' AND `T2`.`city` = 'albany'

\*\*3. Simplification and Optimization:\*\*

\* The SQL query from step 2 is already quite efficient. We've used `INNER JOIN` to combine the tables based on their relationship, and the `WHERE` clause clearly defines our filtering criteria. There's no need for nested queries or complex sub-selections in this case.

\*\*Final Optimized SQL Query:\*\*

```sql

SELECT COUNT(T1.id\_restaurant)

```

FROM {BQ_PROJECT_ID}.restaurant.generalinfo AS T1

INNER JOIN {BQ_PROJECT_ID}.restaurant.location AS T2 ON T1.id_restaurant =
T2.id_restaurant

WHERE T1.food_type = 'thai' AND T1.city = 'albany' AND T2.street_name = 'san
pablo ave'

...

```

=====

Example 2

\*\*\*\*\*

【Database Info】

```

CREATE TABLE {BQ_PROJECT_ID}.financial.account (

    account_id INT64,

    district_id INT64,

    frequency STRING,

    date DATE,

);

CREATE TABLE {BQ_PROJECT_ID}.financial.client (

    client_id INT64,

    gender STRING,

    birth_date DATE,

    district_id INT64,

);

CREATE TABLE {BQ_PROJECT_ID}.financial.district (

    district_id INT64,

    a4 STRING OPTIONS(description="Assuming A4 and A11 are strings due to
examples"),

    a11 STRING,

);

```

\*\*\*\*\*

### 【Question】

Question:

What is the gender of the youngest client who opened account in the lowest average salary branch? Given that Later birthdate refers to younger age; All refers to average salary

\*\*\*\*\*

### 【Answer】

Repeating the question and generating the SQL with Recursive Divide-and-Conquer.

**\*\*Question\*\***: What is the gender of the youngest client who opened account in the lowest average salary branch? Given that Later birthdate refers to younger age; All refers to average salary

**\*\*1. Divide and Conquer:\*\***

\* **\*\*Main Question:\*\*** What is the gender of the youngest client who opened account in the lowest average salary branch?

\* **\*\*Analysis:\*\*** The question is asking about `gender`, and it appears in the table `financial.client`. We will use this as the output column, selecting it from the youngest client in the lowest average salary branch.

\* **\*\*Pseudo \*\*Final Optimized SQL Query:\*\*** SELECT `T1`.`gender` FROM `{BQ\_PROJECT\_ID}.financial.client` AS `T1` WHERE <youngest client in the lowest average salary branch>

\* **\*\*Sub-question 1:\*\*** youngest client in the lowest average salary branch

\* **\*\*Analysis:\*\*** According to the hint, we need to use the `All` from `financial.district` to get the salary info, and the youngest client can be obtained from using the `birth\_date` column of table `financial.client`. The items between these two tables can be INNER JOIN using `district\_id`.

\* **\*\*Pseudo SQL:\*\*** SELECT `T1`.`client\_id` FROM `{BQ\_PROJECT\_ID}.financial.client` AS `T1` INNER JOIN `{BQ\_PROJECT\_ID}.financial.district` AS `T2` ON `T1`.`district\_id` = `T2`.`district\_id` WHERE <lowest average salary branch> ORDER BY `T1`.`birth\_date` DESC NULLS LAST LIMIT 1

\* \*\*Sub-question 1.1:\*\* lowest average salary branch

\* \*\*Analysis:\*\* We can get the lowest average salary branch using order by `A11` ASC and pick top 1. The column `A11` is not NULLABLE, so we do not need to add "IS NOT NULL" filter

\* \*\*Pseudo SQL:\*\* SELECT `district\_id` FROM  
`{BQ\_PROJECT\_ID}.financial.district` ORDER BY `A11` ASC LIMIT 1

\*\*2. Assembling SQL:\*\*

\* \*\*Sub-question 1.1 (lowest average salary branch):\*\*

\* \*\*SQL:\*\* SELECT `district\_id` FROM `{BQ\_PROJECT\_ID}.financial.district`  
ORDER BY `A11` ASC LIMIT 1

\* \*\*Sub-question 1 (youngest client in the lowest average salary branch):\*\*

\* \*\*SQL:\*\* SELECT `T1`.`client\_id` FROM `{BQ\_PROJECT\_ID}.financial.client`  
AS `T1` INNER JOIN `{BQ\_PROJECT\_ID}.financial.district` AS `T2` ON  
`T1`.`district\_id` = `T2`.`district\_id` WHERE `T2`.`district\_id` IN (SELECT  
`district\_id` FROM `financial.district` ORDER BY `A11` ASC LIMIT 1) ORDER BY  
`T1`.`birth\_date` DESC NULLS LAST LIMIT 1

\* \*\*Main Question (gender of the client):\*\*

\* \*\*SQL:\*\* SELECT `T1`.`gender` FROM `{BQ\_PROJECT\_ID}.financial.client` AS  
`T1` WHERE `T1`.`client\_id` = (SELECT `T1`.`client\_id` FROM  
`{BQ\_PROJECT\_ID}.financial.client` AS `T1` INNER JOIN  
`{BQ\_PROJECT\_ID}.financial.district` AS `T2` ON `T1`.`district\_id` =  
`T2`.`district\_id` WHERE `T2`.`district\_id` IN (SELECT `district\_id` FROM  
`{BQ\_PROJECT\_ID}.financial.district` ORDER BY `A11` ASC LIMIT 1) ORDER BY  
`T1`.`birth\_date` DESC NULLS LAST LIMIT 1)

\*\*3. Simplification and Optimization:\*\*

\* The final SQL query from step 2 can be simplified and optimized. The nested queries can be combined using a single `INNER JOIN` and the filtering can be done within a single `ORDER BY` clause.

\*\*Final Optimized SQL Query:\*\*

```
```sql
```

```
SELECT `T1`.`gender`

FROM `{BQ_PROJECT_ID}.financial.client` AS `T1`

INNER JOIN `{BQ_PROJECT_ID}.financial.district` AS `T2`

ON `T1`.`district_id` = `T2`.`district_id`

ORDER BY `T2`.`A11` ASC, `T1`.`birth_date` DESC NULLS LAST

LIMIT 1
```

```
```
```

```
=====
```

Example 3 (dividing into two parallel sub-questions)

```
*****
```

#### **【Database Info】**

```
CREATE TABLE {BQ_PROJECT_ID}.olympics.games
```

```
(

  id INT64,

  games_year INT64 OPTIONS(description="description: the year of the game"),

  games_name STRING,

);
```

```
CREATE TABLE {BQ_PROJECT_ID}.olympics.games_city
```

```
(

  games_id INT64,

  city_id INT64 OPTIONS(description="the id of the city that held the game Maps
to city(id)"),

);
```

```
CREATE TABLE {BQ_PROJECT_ID}.olympics.city
```

```
(
```

```
id INT64,  
  
city_name STRING,  
  
);
```

\*\*\*\*\*

### 【Question】

Question:

From 1900 to 1992, how many games did London host? From 1900 to 1992 refers to games\_year BETWEEN 1900 AND 1992; London refers to city\_name = 'London'; games refer to games\_name;

\*\*\*\*\*

### 【Answer】

Repeating the question and generating the SQL with Recursive Divide-and-Conquer.

**\*\*Question\*\***: From 1900 to 1992, how many games did London host? From 1900 to 1992 refers to games\_year BETWEEN 1900 AND 1992; London refers to city\_name = 'London'; games refer to games\_name;

**\*\*1. Divide and Conquer\*\***

**\* \*\*Main Question:\*\*** From 1900 to 1992, how many games did London host?

**\* \*\*Analysis:\*\*** The question requires us to count games, which are represented by the `id` column in the `olympics.games` table. We need to filter these games based on two criteria: they were hosted in London and occurred between 1900 and 1992.

**\* \*\*Pseudo SQL:\*\*** SELECT COUNT(`T1`.`id`) FROM  
`{BQ\_PROJECT\_ID}.olympics.games` AS `T1` WHERE <games are in London> AND  
<games year between 1900 and 1992>

**\* \*\*Sub-question 1:\*\*** games are in London

**\* \*\*Analysis:\*\*** To determine which games were hosted in London, we need to join the `olympics.games` table with the `olympics.games\_city` table on `games\_id` and then join with the `city` table on `city\_id`. We'll use `INNER JOIN` to ensure only matching records are considered. The filtering on



'London' will be applied to the `city\_name` column.

```
* **Pseudo SQL:** `T1`.`id` IN (SELECT `T1`.`games_id` FROM
`{BQ_PROJECT_ID}.olympics.games_city` AS `T1` INNER JOIN
`{BQ_PROJECT_ID}.olympics.city` AS `T2` ON `T1`.`city_id` = `T2`.`id` WHERE
`T2`.`city_name` = 'London')
```

\* \*\*Sub-question 2:\*\* games year between 1900 and 1992

\* \*\*Analysis:\*\* This involves filtering the `olympics.games` table directly based on the `games\_year` column using the `BETWEEN` operator.

```
* **Pseudo SQL:** `T1`.`games_year` BETWEEN 1900 AND 1992
```

\*\*2. Assembling SQL:\*\*

\* \*\*Sub-question 1 (games are in London):\*\*

```
* **SQL:** `T1`.`id` IN (SELECT `T1`.`games_id` FROM
`{BQ_PROJECT_ID}.olympics.games_city` AS `T1` INNER JOIN
`{BQ_PROJECT_ID}.olympics.city` AS `T2` ON `T1`.`city_id` = `T2`.`id` WHERE
`T2`.`city_name` = 'London')
```

\* \*\*Sub-question 2 (games year between 1900 and 1992):\*\*

```
* **SQL:** `T1`.`games_year` BETWEEN 1900 AND 1992
```

\* \*\*Main Question (count of games):\*\*

```
* **SQL:** SELECT COUNT(`T1`.`id`) FROM `{BQ_PROJECT_ID}.olympics.games` AS
`T1` WHERE `T1`.`id` IN (SELECT `T1`.`games_id` FROM
`{BQ_PROJECT_ID}.olympics.games_city` AS `T1` INNER JOIN
`{BQ_PROJECT_ID}.olympics.city` AS `T2` ON `T1`.`city_id` = `T2`.`id` WHERE
`T2`.`city_name` = 'London') AND `T1`.`games_year` BETWEEN 1900 AND 1992
```

\*\*3. Simplification and Optimization:\*\*

\* The nested query can be converted into a more efficient `JOIN` operation. We'll use `INNER JOIN` to combine `{BQ\_PROJECT\_ID}.olympics.games`, `{BQ\_PROJECT\_ID}.olympics.games\_city`, and `{BQ\_PROJECT\_ID}.olympics.city` based on the relationships between them.

**\*\*Final Optimized SQL Query:\*\***

```sql

SELECT COUNT(T3.id)

FROM {BQ\_PROJECT\_ID}.olympics.games\_city AS T1

INNER JOIN {BQ\_PROJECT\_ID}.olympics.city AS T2 ON T1.city\_id = T2.id

INNER JOIN {BQ\_PROJECT\_ID}.olympics.games AS T3 ON T1.games\_id = T3.id

WHERE T2.city\_name = 'London' AND T3.games\_year

BETWEEN 1900 AND 1992

```

=====

Example 4 (When it's not clear which column should be used for a string matching, use a loosen condition such as string LIKE and OR condition to cover multiple possible columns.)

\*\*\*\*\*

**【Database Info】**

CREATE TABLE `{BQ\_PROJECT\_ID}.academics.student\_programs` (

`Program Type` STRING,

`Participants (Ages 10-15)` FLOAT64,

`Total Enrollment (Ages 10-15)` FLOAT64,

`School Category` STRING,

);

\*\*\*\*\*

**【Question】**

Question:

Please list the lowest three participation rates for students aged 10-15 in online programs. Participation rate for students aged 10-15 = `Participants (Ages 10-15)` / `Total Enrollment (Ages 10-15)`

\*\*\*\*\*

## 【Answer】

Repeating the question and generating the SQL with Recursive Divide-and-Conquer.

**\*\*Question:\*\*** Please list the lowest three participation rates for students aged 10-15 in online programs. Participation rate for students aged 10-15 =  $\text{`Participants (Ages 10-15)`} / \text{`Total Enrollment (Ages 10-15)`}$

**\*\*1. Divide and Conquer:\*\***

**\* \*\*Main Question:\*\*** Please list the lowest three participation rates for students aged 10-15 in online programs.

**\* \*\*Analysis:\*\*** The question is asking about the ratio between  $\text{`Participants (Ages 10-15)`}$  and  $\text{`Total Enrollment (Ages 10-15)`}$ . We need to filter the data to only include online programs.

**\* \*\*Pseudo SQL:\*\***  $\text{SELECT (`Participants (Ages 10-15)`} / \text{`Total Enrollment (Ages 10-15)`}) FROM `{BQ\_PROJECT\_ID}.academics.student\_programs` WHERE <online programs> ORDER BY (`Participants (Ages 10-15)`} / \text{`Total Enrollment (Ages 10-15)`}) ASC NULLS LAST LIMIT 3}$

**\* \*\*Sub-question 1:\*\*** online programs

**\* \*\*Analysis:\*\*** We will get the information from the table  $\text{`${BQ\_PROJECT\_ID}.academics.student\_programs`}$ .

**\* \*\*Pseudo SQL:\*\***  $\text{SELECT program\_id FROM `academics.student\_programs` WHERE <condition for online programs>}$

**\* \*\*Sub-question 1.1:\*\*** condition for online programs (Note: This requires external knowledge or database schema information. We need to identify which column(s) indicate "online programs".)

**\* \*\*Analysis:\*\*** We'll assume either "School Category" or "Program Type" columns might contain the term "online."

**\* \*\*Pseudo SQL:\*\***  $\text{LOWER(`School Category`) LIKE '%online%' OR LOWER(`Program Type`) LIKE '%online%'}$

**\*\*2. Assembling SQL:\*\***

\* \*\*Sub-question 1.1 (condition for online programs):\*\*

\* \*\*SQL:\*\* LOWER(`School Category`) LIKE '%online%' OR LOWER(`Program Type`) LIKE '%online%'

\* \*\*Sub-question 1 (online programs):\*\*

\* \*\*SQL:\*\* SELECT program\_id FROM  
`{BQ\_PROJECT\_ID}.academics.student\_programs` WHERE LOWER(`School Category`) LIKE '%online%' OR LOWER(`Program Type`) LIKE '%online%'

\* \*\*Main Question (lowest three participation rates):\*\*

\* \*\*SQL:\*\* SELECT (`Participants (Ages 10-15)` / `Total Enrollment (Ages 10-15)`) FROM `{BQ\_PROJECT\_ID}.academics.student\_programs` WHERE program\_id IN (SELECT program\_id FROM `{BQ\_PROJECT\_ID}.academics.student\_programs` WHERE LOWER(`School Category`) LIKE '%online%' OR LOWER(`Program Type`) LIKE '%online%') ORDER BY (`Participants (Ages 10-15)` / `Total Enrollment (Ages 10-15)`) ASC NULLS LAST LIMIT 3

\*\*3. Simplification and Optimization:\*\*

\* We can directly incorporate the condition for online programs into the main query.

\*\*Final Optimized SQL Query:\*\*

```sql

SELECT `Participants (Ages 10-15)` / `Total Enrollment (Ages 10-15)` FROM  
`{BQ\_PROJECT\_ID}.academics.student\_programs`

WHERE LOWER(`School Category`) LIKE '%online%' OR LOWER(`Program Type`) LIKE '%online%'

AND `Participants (Ages 10-15)` / `Total Enrollment (Ages 10-15)` IS NOT NULL

ORDER BY `Participants (Ages 10-15)` / `Total Enrollment (Ages 10-15)` ASC  
NULLS LAST LIMIT 3;

...

=====

## Example 5

\*\*\*\*\*

### 【Database Info】

```
CREATE TABLE {BQ_PROJECT_ID}.retails.employees (  
  
    employee_id INT64,  
  
    department_id INT64,  
  
    salary INT64,  
  
);
```

\*\*\*\*\*

### 【Question】

Question:

How many employees earn over \$100,000?

\*\*\*\*\*

### 【Answer】

Repeating the question and generating the SQL with Recursive Divide-and-Conquer.

**\*\*Question:\*\*** How many employees earn over \$100,000?

**\*\*1. Divide and Conquer:\*\***

**\* \*\*Main Question:\*\*** How many employees earn over \$100,000?

**\* \*\*Pseudo SQL:\*\*** SELECT COUNT(\*) FROM {BQ\_PROJECT\_ID}.retails.employees  
WHERE <employees earning over 100000>

**\* \*\*Analysis:\*\*** The question is asking about the COUNT of employees. We need to filter the data to only include employees earning over \$100,000.

\* \*\*Sub-question 1:\*\* employees earning over 100000

\* \*\*Analysis:\*\* Simple condition on the `salary` column.

\* \*\*Pseudo SQL:\*\* SELECT employee\_id FROM  
{BQ\_PROJECT\_ID}.retails.employees WHERE salary > 100000

\*\*2. Assembling SQL:\*\*

\* \*\*Sub-question 1 (employees earning over 100000):\*\*

\* \*\*SQL:\*\* SELECT employee\_id FROM {BQ\_PROJECT\_ID}.retails.employees WHERE  
salary > 100000

\* \*\*Main Question (count of employees):\*\*

\* \*\*SQL:\*\* SELECT COUNT(\*) FROM {BQ\_PROJECT\_ID}.retails.employees WHERE  
employee\_id IN (SELECT employee\_id FROM {BQ\_PROJECT\_ID}.retails.employees  
WHERE salary > 100000)

\*\*3. Simplification and Optimization:\*\*

\* We can achieve the same result more efficiently within a single WHERE  
clause.

\*\*Final Optimized SQL Query:\*\*

```sql

SELECT COUNT(\*) FROM {BQ\_PROJECT\_ID}.retails.employees WHERE salary > 100000;

```

=====

Example 6

\*\*\*\*\*

【Database Info】

```
CREATE TABLE {BQ_PROJECT_ID}.airlines.Airlines

(

  FL_DATE STRING OPTIONS(description="flight date"),

  ORIGIN STRING OPTIONS(description="airport of origin"),

  DEST STRING OPTIONS(description="Destination airport"),

);
```

```
CREATE TABLE {BQ_PROJECT_ID}.airlines.Airports

(

  Code STRING,

  Description STRING,

);
```

\*\*\*\*\*

### 【Question】

Question:

How many flights were there from San Diego International airport to Los Angeles International airport in the August of 2018?

flights from refers to ORIGIN; San Diego International airport refers to Description = 'San Diego, CA: San Diego International'; flights to refers to DEST; Los Angeles International airport refers to Description = 'Los Angeles, CA: Los Angeles International'; in the August of 2018 refers to FL\_DATE like '2018/8%';

\*\*\*\*\*

### 【Answer】

**\*\*Repeating the question and generating the SQL with Recursive Divide-and-Conquer.\*\***

**\*\*Question\*\*:** How many flights were there from San Diego International airport to Los Angeles International airport in the August of 2018? flights from refers to ORIGIN; San Diego International airport refers to Description = 'San Diego, CA: San Diego International'; flights to refers to DEST; Los Angeles

International airport refers to Description = 'Los Angeles, CA: Los Angeles International'; in the August of 2018 refers to FL\_DATE like '2018/8%';

**\*\*1. Divide and Conquer:\*\***

**\* \*\*Main Question:\*\*** How many flights were there from San Diego International airport to Los Angeles International airport in the August of 2018?

**\* \*\*Analysis:\*\*** The question asks for a count of flights, which can be obtained by counting the `FL\_DATE` entries in the `airlines.Airlines` table. We need to apply three filters: flights originating from San Diego International, flights destined for Los Angeles International, and flights occurring in August 2018.

**\* \*\*Pseudo SQL:\*\*** SELECT COUNT(`FL\_DATE`) FROM  
`{BQ\_PROJECT\_ID}.airlines.Airlines` WHERE <flights are in August 2018> AND  
<flights are from San Diego International> AND <flights are to Los Angeles International>

**\* \*\*Sub-question 1:\*\*** flights are in August 2018

**\* \*\*Analysis:\*\*** This filter can be directly applied to the  
`{BQ\_PROJECT\_ID}.airlines.Airlines` table using the `FL\_DATE` column and the  
`LIKE` operator, as indicated by the evidence.

**\* \*\*Pseudo SQL:\*\*** `FL\_DATE` LIKE '2018/8%'

**\* \*\*Sub-question 2:\*\*** flights are from San Diego International

**\* \*\*Analysis:\*\*** We need to find the airport code (`ORIGIN`) corresponding to 'San Diego, CA: San Diego International' from the  
`{BQ\_PROJECT\_ID}.airlines.Airports` table and use it to filter the  
`airlines.Airlines` table. This requires joining `airlines.Airports` and  
`airlines.Airlines` based on `airlines.Airports`.`Code` =  
`airlines.Airlines`.`ORIGIN`.

**\* \*\*Pseudo SQL:\*\*** `ORIGIN` = (SELECT `T2`.`ORIGIN` FROM  
`{BQ\_PROJECT\_ID}.airlines.Airports` AS `T1` INNER JOIN  
`{BQ\_PROJECT\_ID}.airlines.Airlines` AS `T2` ON `T1`.`Code` = `T2`.`ORIGIN`  
WHERE `T1`.`Description` = 'San Diego, CA: San Diego International')

**\* \*\*Sub-question 3:\*\*** flights are to Los Angeles International

**\* \*\*Analysis:\*\*** Similar to sub-question 2, we need to find the airport code (`DEST`) for 'Los Angeles, CA: Los Angeles International' from the  
`airlines.Airports` table and use it to filter the `airlines.Airlines` table. This also requires joining `airlines.Airports` and `airlines.Airlines`, but



this time on `airlines.Airports`.`Code` = `airlines.Airlines`.`DEST`.

```
* **Pseudo SQL:** `DEST` = (SELECT `T4`.`DEST` FROM
`{BQ_PROJECT_ID}.airlines.Airports` AS `T3` INNER JOIN
`{BQ_PROJECT_ID}.airlines.Airlines` AS `T4` ON `T3`.`Code` = `T4`.`DEST` WHERE
`T3`.`Description` = 'Los Angeles, CA: Los Angeles International')
```

**\*\*2. Assembling SQL:\*\***

\* **\*\*Sub-question 1 (flights are in August 2018):\*\***

```
* **SQL:** `FL_DATE` LIKE '2018/8%'
```

\* **\*\*Sub-question 2 (flights are from San Diego International):\*\***

```
* **SQL:** `ORIGIN` = (SELECT DISTINCT `T2`.`ORIGIN` FROM
`{BQ_PROJECT_ID}.airlines.Airports` AS `T1` INNER JOIN
`{BQ_PROJECT_ID}.airlines.Airlines` AS `T2` ON `T1`.`Code` = `T2`.`ORIGIN`
WHERE `T1`.`Description` = 'San Diego, CA: San Diego International')
```

\* **\*\*Sub-question 3 (flights are to Los Angeles International):\*\***

```
* **SQL:** `DEST` = (SELECT DISTINCT `T4`.`DEST` FROM
`{BQ_PROJECT_ID}.airlines.Airports` AS `T3` INNER JOIN
`{BQ_PROJECT_ID}.airlines.Airlines` AS `T4` ON `T3`.`Code` = `T4`.`DEST` WHERE
`T3`.`Description` = 'Los Angeles, CA: Los Angeles International')
```

\* **\*\*Main Question (count of flights):\*\***

```
* **SQL:** SELECT COUNT(`FL_DATE`) FROM `{BQ_PROJECT_ID}.airlines.Airlines`
WHERE `FL_DATE` LIKE '2018/8%' AND `ORIGIN` = (SELECT `T2`.`ORIGIN` FROM
`{BQ_PROJECT_ID}.airlines.Airports` AS `T1` INNER JOIN
`{BQ_PROJECT_ID}.airlines.Airlines` AS `T2` ON `T1`.`Code` = `T2`.`ORIGIN`
WHERE `T1`.`Description` = 'San Diego, CA: San Diego International') AND
`DEST` = (SELECT `T4`.`DEST` FROM `{BQ_PROJECT_ID}.airlines.Airports` AS `T3`
INNER JOIN `{BQ_PROJECT_ID}.airlines.Airlines` AS `T4` ON `T3`.`Code` =
`T4`.`DEST` WHERE `T3`.`Description` = 'Los Angeles, CA: Los Angeles
International')
```

**\*\*3. Simplification and Optimization:\*\***

\* The query in step 2 is already quite optimized. We are using nested queries to avoid joining the `airlines.Airports` table multiple times in the main

query, which could potentially impact performance.

**\*\*Final Optimized SQL Query:\*\***

```sql

```
SELECT COUNT(FL_DATE)

FROM {BQ_PROJECT_ID}.airlines.Airlines

WHERE FL_DATE LIKE '2018/8%'

AND ORIGIN = (

    SELECT DISTINCT T2.ORIGIN

    FROM {BQ_PROJECT_ID}.airlines.Airports AS T1

    INNER JOIN {BQ_PROJECT_ID}.airlines.Airlines AS T2 ON T1.Code = T2.ORIGIN

    WHERE T1.Description = 'San Diego, CA: San Diego International' )

AND DEST = (

    SELECT DISTINCT T4.DEST

    FROM {BQ_PROJECT_ID}.airlines.Airports AS T3

    INNER JOIN {BQ_PROJECT_ID}.airlines.Airlines AS T4 ON T3.Code = T4.DEST

    WHERE T3.Description = 'Los Angeles, CA: Los Angeles International' )

```
```

=====

Example 7

\*\*\*\*\*

**【Database Info】**

CREATE TABLE {BQ\_PROJECT\_ID}.food\_inspection.businesses

```
(

    `business_id` INT64,

    `name` STRING OPTIONS(description="the name of the eatery"),

);
```

```
CREATE TABLE {BQ_PROJECT_ID}.food_inspection.inspections

(

    `business_id` INT64 OPTIONS(description="the unique id of the
business"),

    `score` INT64 OPTIONS(description="description: the inspection score"),

    `date` DATE,

);
```

```
CREATE TABLE {BQ_PROJECT_ID}.food_inspection.violations

(

    `business_id` INT64,

    `date` DATE,

);
```

\*\*\*\*\*

### 【Question】

Question:

What are the names of the establishments that met all the required standards for 4 consecutive years?

establishment has the same meaning as business; score of 90 or more refers to score  $\geq 90$ ; year(date) = 2015; ; met all required standards for 4 consecutive years refers to COUNT(year(date)) = 4 where score = 100;

\*\*\*\*\*

### 【Answer】

Repeating the question and generating the SQL with Recursive Divide-and-Conquer.

**\*\*Question\*\***: What are the names of the establishments that met all the required standards for 4 consecutive years? establishment has the same meaning as business; score of 90 or more refers to score  $\geq 90$ ; year(date) = 2015; ;

met all required standards for 4 consecutive years refers to COUNT(year(date))  
= 4 where score = 100;

**\*\*1. Divide and Conquer:\*\***

**\* \*\*Main Question:\*\*** What are the names of the establishments that met all the required standards for 4 consecutive years?

**\* \*\*Analysis:\*\*** We need to find the names of businesses that have a score of 100 for 4 consecutive years. The `food\_inspection.businesses` table contains the `name` and the `{BQ\_PROJECT\_ID}.food\_inspection.inspections` table contains the `score` and `date`. We will need to join these tables and filter by score. To check for consecutive years, we'll need to group by business and year, then check if each group has a count of 4.

**\* \*\*Pseudo SQL:\*\*** SELECT DISTINCT `T2`.`name` FROM  
{BQ\_PROJECT\_ID}.food\_inspection.inspections` AS `T1` INNER JOIN  
{BQ\_PROJECT\_ID}.food\_inspection.businesses` AS `T2` ON `T1`.`business\_id` =  
`T2`.`business\_id` WHERE <score = 100> AND <4 consecutive years>

**\* \*\*Sub-question 1:\*\*** score = 100

**\* \*\*Analysis:\*\*** This is a simple filter on the  
{BQ\_PROJECT\_ID}.food\_inspection.inspections` table where we select rows with  
a `score` of 100.

**\* \*\*Pseudo SQL:\*\*** `T1`.`score` = 100

**\* \*\*Sub-question 2:\*\*** 4 consecutive years

**\* \*\*Analysis:\*\*** This is more complex. We need to group the inspections by business and year, then check if the count for each group is 4. To get the year from the `date` column, we'll use the `FORMAT\_DATE('%Y', date)` function. We'll also need to use window functions to assign a rank to each year within a business, allowing us to check for consecutiveness.

**\* \*\*Pseudo SQL:\*\*** `T2`.`name` IN (SELECT `T4`.`name` FROM (SELECT  
`T3`.`name`, `T3`.`years`, row\_number() OVER (PARTITION BY `T3`.`name` ORDER  
BY `T3`.`years`) AS `rowNumber` FROM (SELECT DISTINCT `name`,  
FORMAT\_DATE('%Y', date) AS `years` FROM  
{BQ\_PROJECT\_ID}.food\_inspection.inspections` AS `T1` INNER JOIN  
{BQ\_PROJECT\_ID}.food\_inspection.businesses` AS `T2` ON `T1`.`business\_id` =  
`T2`.`business\_id` WHERE `T1`.`score` = 100) AS `T3`) AS `T4` GROUP BY  
`T4`.`name`, date(`T4`.`years` || '-01-01', '-' || (`T4`.`rowNumber` - 1) || '  
years') HAVING COUNT(`T4`.`years`) = 4)

**\* \*\*Sub-question 2.1:\*\*** Get distinct businesses and their inspection

years where the score is 100

\* \*\*Analysis:\*\* We need to join  
`{BQ\_PROJECT\_ID}.food\_inspection.inspections` and  
`{BQ\_PROJECT\_ID}.food\_inspection.businesses` tables, filter by `score` = 100,  
and select distinct business names and their inspection years.

\* \*\*Pseudo SQL:\*\* SELECT DISTINCT `name`, FORMAT\_DATE('%Y', date)  
AS `years` FROM `{BQ\_PROJECT\_ID}.food\_inspection.inspections` AS `T1` INNER  
JOIN `{BQ\_PROJECT\_ID}.food\_inspection.businesses` AS `T2` ON  
`T1`.`business\_id` = `T2`.`business\_id` WHERE `T1`.`score` = 100

\* \*\*Sub-question 2.2:\*\* Assign a rank to each year within a business

\* \*\*Analysis:\*\* We'll use the `row\_number()` window function to  
assign a rank to each year within each business, ordered chronologically. This  
will help us identify consecutive years later.

\* \*\*Pseudo SQL:\*\* SELECT `T3`.`name`, `T3`.`years`, row\_number()  
OVER (PARTITION BY `T3`.`name` ORDER BY `T3`.`years`) AS `rowNumber` FROM  
`{BQ\_PROJECT\_ID}.food\_inspection.inspections` AS `T1` INNER JOIN  
`{BQ\_PROJECT\_ID}.food\_inspection.businesses` AS `T2` ON `T1`.`business\_id` =  
`T2`.`business\_id` WHERE `T1`.`score` = 100` AS `T3`

\* \*\*Sub-question 2.3:\*\* Group by business and consecutive year groups  
and check if the count is 4

\* \*\*Analysis:\*\* We'll group the results by business name and a  
calculated date representing the start of each potential 4-year period. This  
date is calculated by adding (`rowNumber` - 1) years to the first day of the  
year extracted from the `years` column. We then filter for groups with a count  
of 4, indicating 4 consecutive years.

\* \*\*Pseudo SQL:\*\* SELECT `T4`.`name` FROM (<previous sub-query>) AS  
`T4` GROUP BY `T4`.`name`, date(`T4`.`years` || '-01-01', '-' ||  
(`T4`.`rowNumber` - 1) || ' years') HAVING COUNT(`T4`.`years`) = 4

\*\*2. Assembling SQL:\*\*

\* \*\*Sub-question 2.1 (distinct businesses and years with score 100):\*\*

\* \*\*SQL:\*\* SELECT DISTINCT `name`, FORMAT\_DATE('%Y', date) AS `years` FROM  
`{BQ\_PROJECT\_ID}.food\_inspection.inspections` AS `T1` INNER JOIN  
`{BQ\_PROJECT\_ID}.food\_inspection.businesses` AS `T2` ON `T1`.`business\_id` =  
`T2`.`business\_id` WHERE `T1`.`score` = 100

\* \*\*Sub-question 2.2 (assign rank to each year within a business):\*\*

```

* **SQL:** SELECT `T3`.`name`, `T3`.`years`, row_number() OVER (PARTITION
BY `T3`.`name` ORDER BY `T3`.`years`) AS `rowNumber` FROM (SELECT DISTINCT
`name`, FORMAT_DATE('%Y', date) AS `years` FROM
`{BQ_PROJECT_ID}.food_inspection.inspections` AS `T1` INNER JOIN
`{BQ_PROJECT_ID}.food_inspection.businesses` AS `T2` ON `T1`.`business_id` =
`T2`.`business_id` WHERE `T1`.`score` = 100) AS `T3`

```

\* \*\*Sub-question 2.3 (group by business and consecutive year groups):\*\*

```

* **SQL:** SELECT `T4`.`name` FROM (SELECT `T3`.`name`, `T3`.`years`,
row_number() OVER (PARTITION BY `T3`.`name` ORDER BY `T3`.`years`) AS
`rowNumber` FROM (SELECT DISTINCT `name`, FORMAT_DATE('%Y', date) AS `years`
FROM `{BQ_PROJECT_ID}.food_inspection.inspections` AS `T1` INNER JOIN
`{BQ_PROJECT_ID}.food_inspection.businesses` AS `T2` ON `T1`.`business_id` =
`T2`.`business_id` WHERE `T1`.`score` = 100) AS `T3`) AS `T4` GROUP BY
`T4`.`name`, DATE_SUB(DATE(CONCAT(T4.years, '-01-01')), INTERVAL (T4.rowNumber
- 1) YEAR) HAVING COUNT(`T4`.`years`) = 4

```

\* \*\*Sub-question 2 (4 consecutive years):\*\*

```

* **SQL:** `T2`.`name` IN (SELECT `T4`.`name` FROM (SELECT `T3`.`name`,
`T3`.`years`, row_number() OVER (PARTITION BY `T3`.`name` ORDER BY
`T3`.`years`) AS `rowNumber` FROM (SELECT DISTINCT `name`, FORMAT_DATE('%Y',
date) AS `years` FROM `{BQ_PROJECT_ID}.food_inspection.inspections` AS `T1`
INNER JOIN `{BQ_PROJECT_ID}.food_inspection.businesses` AS `T2` ON
`T1`.`business_id` = `T2`.`business_id` WHERE `T1`.`score` = 100) AS `T3`) AS
`T4` GROUP BY `T4`.`name`, DATE_SUB(DATE(CONCAT(T4.years, '-01-01')), INTERVAL
(T4.rowNumber - 1) YEAR) HAVING COUNT(`T4`.`years`) = 4)

```

\* \*\*Main Question (names of establishments):\*\*

```

* **SQL:** SELECT DISTINCT `T2`.`name` FROM `{BQ_PROJECT_ID}.inspections`
AS `T1` INNER JOIN `{BQ_PROJECT_ID}.businesses` AS `T2` ON `T1`.`business_id`
= `T2`.`business_id` WHERE `T1`.`score` = 100 AND `T2`.`name` IN (SELECT
`T4`.`name` FROM (SELECT `T3`.`name`, `T3`.`years`, row_number() OVER
(PARTITION BY `T3`.`name` ORDER BY `T3`.`years`) AS `rowNumber` FROM (SELECT
DISTINCT `name`, FORMAT_DATE('%Y', date) AS `years` FROM
`{BQ_PROJECT_ID}.food_inspection.inspections` AS `T1` INNER JOIN
`{BQ_PROJECT_ID}.food_inspection.businesses` AS `T2` ON `T1`.`business_id` =
`T2`.`business_id` WHERE `T1`.`score` = 100) AS `T3`) AS `T4` GROUP BY
`T4`.`name`, DATE_SUB(DATE(CONCAT(T4.years, '-01-01')), INTERVAL (T4.rowNumber
- 1) YEAR) HAVING COUNT(`T4`.`years`) = 4)

```

\*\*3. Simplification and Optimization:\*\*

\* The final SQL query from step 2 can be simplified by merging the nested

queries into a single query with a `WITH` clause. This improves readability and potentially performance.

**\*\*Final Optimized SQL Query:\*\***

```sql

SELECT DISTINCT T4.name

FROM ( SELECT T3.name, T3.years, row\_number()

OVER (PARTITION BY T3.name ORDER BY T3.years)

AS rowNumber FROM ( SELECT DISTINCT name, FORMAT\_DATE('%Y', date)

AS years FROM {BQ\_PROJECT\_ID}.food\_inspection.inspections AS T1

INNER JOIN {BQ\_PROJECT\_ID}.food\_inspection.businesses AS T2 ON T1.business\_id  
= T2.business\_id

WHERE T1.score = 100 ) AS T3 ) AS T4

GROUP BY T4.name, DATE\_SUB(DATE(CONCAT(T4.years, '-01-01')), INTERVAL  
(T4.rowNumber - 1) YEAR) HAVING COUNT(T4.years) = 4

```

=====

Example 8

\*\*\*\*\*

**【Database Info】**

CREATE TABLE `bigquery-public-  
data.covid19\_symptom\_search.symptom\_search\_sub\_region\_2\_daily`

(

country\_region\_code STRING,

country\_region STRING,

sub\_region\_1 STRING,

sub\_region\_1\_code STRING,

sub\_region\_2 STRING,

sub\_region\_2\_code STRING,

place\_id STRING,

```

    date DATE,

    symptom_Abdominal_obesity FLOAT64,

    symptom_Abdominal_pain FLOAT64,

    symptom_Acne FLOAT64

)

PARTITION BY date

CLUSTER BY country_region_code, sub_region_1_code, sub_region_2_code,
sub_region_2;

```

\*\*\*\*\*

### 【Question】

Question:

Find the day in which the symptom that occurs most frequently is headache.

\*\*\*\*\*

### 【Answer】

Repeating the question and generating the SQL with Recursive Divide-and-Conquer.

**\*\*Question\*\*:** Find the day in which the symptom that occurs most frequently is headache.

**Analysis:** We need to determine the day (day of the week) when the frequency of searches for the symptom "headache" is the highest. This involves:

- Grouping the data by the day of the week.
- Counting the occurrences of searches for "headache."
- Sorting the counts in descending order and selecting the day with the highest count.

Pseudo SQL:

```
SELECT FORMAT_DATE('%A', date) AS day, COUNT(*) AS headache_count
```



```

FROM `bigquery-public-
data.covid19_symptom_search.symptom_search_sub_region_2_daily`

WHERE symptom_Headache > 0

GROUP BY day

ORDER BY headache_count DESC

LIMIT 1

```

Sub-question 1: Extract the day of the week from the date column.

- Analysis: Use the FORMAT\_DATE function with the %A format specifier to extract the day name (e.g., "Monday," "Tuesday") from the date column.

Pseudo SQL:

```

SELECT FORMAT_DATE('%A', date) AS day

FROM `bigquery-public-
data.covid19_symptom_search.symptom_search_sub_region_2_daily`

```

Sub-question 2: Filter rows where "headache" searches occurred.

- Analysis: Only include rows where the symptom "headache" has a positive value (symptom\_Headache > 0).

Pseudo SQL:

```

SELECT date

FROM `bigquery-public-
data.covid19_symptom_search.symptom_search_sub_region_2_daily`

WHERE symptom_Headache > 0

```

Sub-question 3: Count the occurrences of "headache" searches grouped by day of the week.

- Analysis: After filtering the data for rows where symptom\_Headache > 0, group the data by the day of the week and count the number of rows for each day.

Pseudo SQL:

```
SELECT FORMAT_DATE('%A', date) AS day, COUNT(*) AS headache_count

FROM `bigquery-public-
data.covid19_symptom_search.symptom_search_sub_region_2_daily`

WHERE symptom_Headache > 0

GROUP BY day
```

Sub-question 4: Sort the results by the count in descending order and get the top day.

- Analysis: Use the ORDER BY clause to sort by the count of "headache" searches in descending order. Limit the result to 1 to get the top day.

Pseudo SQL:

```
SELECT FORMAT_DATE('%A', date) AS day, COUNT(*) AS headache_count

FROM `bigquery-public-
data.covid19_symptom_search.symptom_search_sub_region_2_daily`

WHERE symptom_Headache > 0

GROUP BY day

ORDER BY headache_count DESC

LIMIT 1
```

Assembling SQL

- Combining all sub-questions into the final query:

**\*\*Final Optimized SQL Query:\*\***

```sql

SELECT

FORMAT\_DATE('%A', PARSE\_DATE('%Y-%m-%d', date)) AS day,

COUNT(\*) AS headache\_count

FROM

`bigquery-public-

```

data`.`covid19_symptom_search`.`symptom_search_country_daily`

GROUP BY

    day

ORDER BY

    headache_count DESC

LIMIT 1;

` ``

```

Now is the real question, following the instruction and examples, generate the GoogleSQL with Recursive Divide-and-Conquer approach.

Follow all steps from the strategy. When you get to the final query, output the query string ONLY in the format ``sql ... ``. Make sure you only output one single query.

Table names always should be exactly the same as the table names mentioned in the database schema, for example, `{BQ\_PROJECT\_ID}.airlines.Airlines` instead of `Airlines`.

\*\*\*\*\*

**【Table creation statements】**

{SCHEMA}

\*\*\*\*\*

**【Question】**

Question:

{QUESTION}

\*\*\*\*\*

**【Answer】**

Repeating the question and generating the SQL with Recursive Divide-and-Conquer.

""

```
=====

File: data_science/sub_agents/bigquery/chase_sql/llm_utils.py

=====

# Copyright 2025 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at

#

#     http://www.apache.org/licenses/LICENSE-2.0

#

# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""This code contains the LLM utils for the CHASE-SQL Agent."""

import functools

import os

import random

import time

from concurrent.futures import ThreadPoolExecutor, as_completed

from typing import Callable, List, Optional
```

```

import dotenv

import vertexai

from google.cloud import aiplatform

from vertexai.generative_models import (GenerationConfig, HarmBlockThreshold,
                                         HarmCategory)

from vertexai.preview import caching

from vertexai.preview.generative_models import GenerativeModel

dotenv.load_dotenv(override=True)

SAFETY_FILTER_CONFIG = {

    HarmCategory.HARM_CATEGORY_UNSPECIFIED: HarmBlockThreshold.BLOCK_NONE,

    HarmCategory.HARM_CATEGORY_DANGEROUS_CONTENT:
HarmBlockThreshold.BLOCK_NONE,

    HarmCategory.HARM_CATEGORY_HATE_SPEECH: HarmBlockThreshold.BLOCK_NONE,

    HarmCategory.HARM_CATEGORY_HARASSMENT: HarmBlockThreshold.BLOCK_NONE,

    HarmCategory.HARM_CATEGORY_SEXUALLY_EXPLICIT:
HarmBlockThreshold.BLOCK_NONE,

}

GCP_PROJECT = os.getenv("GOOGLE_CLOUD_PROJECT")

GCP_LOCATION = os.getenv("GOOGLE_CLOUD_LOCATION")

GEMINI_AVAILABLE_REGIONS = [

    "europe-west3",

    "australia-southeast1",

    "us-east4",

    "northamerica-northeast1",

    "europe-central2",

    "us-central1",

```

```

    "europe-north1",

    "europe-west8",

    "us-south1",

    "us-east1",

    "asia-east2",

    "us-west1",

    "europe-west9",

    "europe-west2",

    "europe-west6",

    "europe-southwest1",

    "us-west4",

    "asia-northeast1",

    "asia-east1",

    "europe-west1",

    "europe-west4",

    "asia-northeast3",

    "asia-south1",

    "asia-southeast1",

    "southamerica-east1",

]

GEMINI_URL = (

    "projects/{GCP_PROJECT}/locations/{region}/publishers/google/models/
    {model_name}"

)

aiplatform.init(

    project=GCP_PROJECT,

    location=GCP_LOCATION,

)

```

```
vertexai.init(project=GCP_PROJECT, location=GCP_LOCATION)
```

```
def retry(max_attempts=8, base_delay=1, backoff_factor=2):
```

```
    """Decorator to add retry logic to a function.
```

```
    Args:
```

```
        max_attempts (int): The maximum number of attempts.
```

```
        base_delay (int): The base delay in seconds for the exponential  
backoff.
```

```
        backoff_factor (int): The factor by which to multiply the delay for  
each
```

```
        subsequent attempt.
```

```
    Returns:
```

```
        Callable: The decorator function.
```

```
    """
```

```
def decorator(func):
```

```
    @functools.wraps(func)
```

```
    def wrapper(*args, **kwargs):
```

```
        attempts = 0
```

```
        while attempts < max_attempts:
```

```
            try:
```

```
                return func(*args, **kwargs)
```

```
            except Exception as e: # pylint: disable=broad-exception-  
caught
```

```
                print(f"Attempt {attempts + 1} failed with error: {e}")
```

```
                attempts += 1
```

```
            if attempts >= max_attempts:
```

```
        raise e

        delay = base_delay * (backoff_factor**attempts)

        delay = delay + random.uniform(0, 0.1 * delay)

        time.sleep(delay)

    return wrapper

return decorator
```

```
class GeminiModel:

    """Class for the Gemini model."""

    def __init__(

        self,

        model_name: str = "gemini-2.0-flash-001",

        finetuned_model: bool = False,

        distribute_requests: bool = False,

        cache_name: str | None = None,

        temperature: float = 0.01,

        **kwargs,

    ):

        self.model_name = model_name

        self.finetuned_model = finetuned_model

        self.arguments = kwargs

        self.distribute_requests = distribute_requests

        self.temperature = temperature

        model_name = self.model_name
```



```

if not self.finetuned_model and self.distribute_requests:

    random_region = random.choice(GEMINI_AVAILABLE_REGIONS)

    model_name = GEMINI_URL.format(

        GCP_PROJECT=GCP_PROJECT,

        region=random_region,

        model_name=self.model_name,

    )

if cache_name is not None:

    cached_content =
    caching.CachedContent(cached_content_name=cache_name)

    self.model = GenerativeModel.from_cached_content(

        cached_content=cached_content

    )

else:

    self.model = GenerativeModel(model_name=model_name)

@retry(max_attempts=12, base_delay=2, backoff_factor=2)

def call(self, prompt: str, parser_func=None) -> str:

    """Calls the Gemini model with the given prompt.

    Args:

        prompt (str): The prompt to call the model with.

        parser_func (callable, optional): A function that processes the

LLM        output. It takes the model's response as input and returns the

        processed result.

    Returns:

        str: The processed response from the model.

    """

```

```

response = self.model.generate_content(

    prompt,

    generation_config=GenerationConfig(

        temperature=self.temperature,

        **self.arguments,

    ),

    safety_settings=SAFETY_FILTER_CONFIG,

).text

if parser_func:

    return parser_func(response)

return response


def call_parallel(

    self,

    prompts: List[str],

    parser_func: Optional[Callable[[str], str]] = None,

    timeout: int = 60,

    max_retries: int = 5,

) -> List[Optional[str]]:

    """Calls the Gemini model for multiple prompts in parallel using
    threads with retry logic.

    Args:

        prompts (List[str]): A list of prompts to call the model with.

        parser_func (callable, optional): A function to process each
response.

        timeout (int): The maximum time (in seconds) to wait for each
thread.

        max_retries (int): The maximum number of retries for timed-out
threads.

```

Returns:

```
List[Optional[str]]:
```

A list of responses, or None for threads that failed.

```
"""
```

```
results = [None] * len(prompts)
```

```
def worker(index: int, prompt: str):
```

```
    """Thread worker function to call the model and store the result
    with retries."""
```

```
    retries = 0
```

```
    while retries <= max_retries:
```

```
        try:
```

```
            return self.call(prompt, parser_func)
```

```
        except Exception as e: # pylint: disable=broad-exception-
    caught
```

```
            print(f"Error for prompt {index}: {str(e)}")
```

```
            retries += 1
```

```
            if retries <= max_retries:
```

```
                print(f"Retrying ({retries}/{max_retries}) for prompt
    {index}")
```

```
                time.sleep(1) # Small delay before retrying
```

```
            else:
```

```
                return f"Error after retries: {str(e)}"
```

```
# Create and start one thread for each prompt
```

```
with ThreadPoolExecutor(max_workers=len(prompts)) as executor:
```

```
    future_to_index = {
```

```
        executor.submit(worker, i, prompt): i
```

```
        for i, prompt in enumerate(prompts)
```

```

    }

    for future in as_completed(future_to_index, timeout=timeout):

        index = future_to_index[future]

        try:

            results[index] = future.result()

        except Exception as e: # pylint: disable=broad-exception-
caught

            print(f"Unhandled error for prompt {index}: {e}")

            results[index] = "Unhandled Error"

# Handle remaining unfinished tasks after the timeout

for future in future_to_index:

    index = future_to_index[future]

    if not future.done():

        print(f"Timeout occurred for prompt {index}")

        results[index] = "Timeout"

return results

```

```

=====

```

File: data\_science/sub\_agents/bigquery/chase\_sql/qp\_prompt\_template.py

```

=====

```

# Copyright 2025 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");

# you may not use this file except in compliance with the License.

```

# You may obtain a copy of the License at

#

#      http://www.apache.org/licenses/LICENSE-2.0

#

# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""Query Plan (QP) prompt template."""

QP_PROMPT_TEMPLATE = """

You are an experienced database expert.

Now you need to generate a GoogleSQL or BigQuery query given the database
information, a question and some additional information.

The database structure is defined by table schemas (some columns provide
additional column descriptions in the options).

Given the table schema information description and the `Question`. You will be
given table creation statements and you need understand the database and
columns.

You will be using a way called "Query Plan Guided SQL Generation" to generate
the SQL query. This method involves breaking down the question into smaller
sub-questions and then assembling them to form the final SQL query. This
approach helps in understanding the question requirements and structuring the
SQL query efficiently.

Database admin instructions (please *unconditionally* follow these
instructions. Do *not* ignore them or use them as hints.):

1. **SELECT Clause:**

    - Select only the necessary columns by explicitly specifying them in the
    `SELECT` statement. Avoid redundant columns or values.

```

2. **\*\*Aggregation (MAX/MIN):\*\***

- Ensure `JOIN`s are completed before applying `MAX()` or `MIN()`.

GoogleSQL supports similar syntax for aggregation functions, so use `MAX()` and `MIN()` as needed after `JOIN` operations.

3. **\*\*ORDER BY with Distinct Values:\*\***

- In GoogleSQL, `GROUP BY <column>` can be used before `ORDER BY <column> ASC|DESC` to get distinct values and sort them.

4. **\*\*Handling NULLs:\*\***

- To filter out NULL values, use `JOIN` or add a `WHERE <column> IS NOT NULL` clause.

5. **\*\*FROM/JOIN Clauses:\*\***

- Only include tables essential to the query. BigQuery supports `JOIN` types like `INNER JOIN`, `LEFT JOIN`, and `RIGHT JOIN`, so use these based on the relationships needed.

6. **\*\*Strictly Follow Hints:\*\***

- Carefully adhere to any specified conditions in the instructions for precise query construction.

7. **\*\*Thorough Question Analysis:\*\***

- Review all specified conditions or constraints in the question to ensure they are fully addressed in the query.

8. **\*\*DISTINCT Keyword:\*\***

- Use `SELECT DISTINCT` when unique values are needed, such as for IDs or URLs.

9. **\*\*Column Selection:\*\***

- Pay close attention to column descriptions and any hints to select the

correct column, especially when similar columns exist across tables.

#### 10. **\*\*String Concatenation:\*\***

- GoogleSQL uses ``CONCAT()`` for string concatenation. Avoid using ``||`` and instead use ``CONCAT(column1, ' ', column2)`` for concatenation.

#### 11. **\*\*JOIN Preference:\*\***

- Use ``INNER JOIN`` when appropriate, and avoid nested ``SELECT`` statements if a ``JOIN`` will achieve the same result.

#### 12. **\*\*GoogleSQL Functions Only:\*\***

- Use functions available in GoogleSQL. Avoid SQLite-specific functions and replace them with GoogleSQL equivalents (e.g., ``FORMAT_DATE`` instead of ``STRFTIME``).

#### 13. **\*\*Date Processing:\*\***

- GoogleSQL supports ``FORMAT_DATE('%Y', date_column)`` for extracting the year. Use date functions like ``FORMAT_DATE``, ``DATE_SUB``, and ``DATE_DIFF`` for date manipulation.

#### 14. **\*\*Table Names and reference:\*\***

- As required by BigQuery, always use the full table name with the database prefix in the SQL statement. For example, `"SELECT * FROM example_bigquery_database.table_a"`, not just `"SELECT * FROM table_a"`

#### 15. **\*\*GROUP BY or AGGREGATE:\*\***

- In queries with `GROUP BY`, all columns in the `SELECT` list must either: Be included in the `GROUP BY` clause, or Be used in an aggregate function (e.g., `MAX`, `MIN`, `AVG`, `COUNT`, `SUM`).

Here are some examples

=====

Example 1

\*\*\*\*\*

**【Table creation statements】**

```
CREATE TABLE {BQ_PROJECT_ID}.restaurant.generalinfo

(

  id_restaurant INT64,

  food_type STRING OPTIONS(description="the food type"),

  city STRING OPTIONS(description="the city where the restaurant is located
in"),

);

CREATE TABLE {BQ_PROJECT_ID}.restaurant.location

(

  id_restaurant INT64,

  street_name STRING OPTIONS(description="the street name of the restaurant"),

  city STRING OPTIONS(description="the city where the restaurant is located in
foreign key (id_restaurant) references generalinfo (id_restaurant) on update
cascade on delete cascade"),

);
```

\*\*\*\*\*

**【Question】**

Question:

How many Thai restaurants can be found in San Pablo Ave, Albany? Thai restaurant refers to food\_type = 'thai'; San Pablo Ave Albany refers to street\_name = 'san pablo ave' AND T1.city = 'albany'

\*\*\*\*\*

**【Answer】**

Repeating the question and generating the SQL with Recursive Divide-and-Conquer.

**\*\*Question\*\*:** How many Thai restaurants can be found in San Pablo Ave, Albany? Thai restaurant refers to food\_type = 'thai'; San Pablo Ave Albany refers to



```
street_name = 'san pablo ave' AND T1.city = 'albany'
```

**\*\*Query Plan\*\*:**

**\*\* Preparation Steps:\*\***

1. Initialize the process: Start preparing to execute the query.
2. Prepare storage: Set up storage space (registers) to hold temporary results, initializing them to NULL.
3. Open the location table: Open the location table so we can read from it.
4. Open the generalinfo table: Open the generalinfo table so we can read from it.

**\*\* Matching Restaurants:\*\***

1. Start reading the location table: Move to the first row in the location table.
2. Check if the street matches: Look at the street\_name column of the current row in location. If it's not "san pablo ave," skip this row.
3. Identify the matching row: Store the identifier (row ID) of this location entry.
4. Find the corresponding row in generalinfo: Use the row ID from location to directly find the matching row in generalinfo.
5. Check if the food type matches: Look at the food\_type column in generalinfo. If it's not "thai," skip this row.
6. Check if the city matches: Look at the city column in generalinfo. If it's not "albany," skip this row.

**\*\* Counting Restaurants:\*\***

1. Prepare to count this match: If all checks pass, prepare to include this row in the final count.
2. Count this match: Increment the count for each row that meets all the criteria.
3. Move to the next row in location: Go back to the location table and move to the next row, repeating the process until all rows are checked.

4. Finalize the count: Once all rows have been checked, finalize the count of matching rows.

5. Prepare the result: Copy the final count to prepare it for output.

**\*\* Delivering the Result:\*\***

1. Output the result: Output the final count, which is the number of restaurants that match all the specified criteria.

2. End the process: Stop the query execution process.

3. Setup phase: Before starting the actual query execution, the system prepares the specific values it will be looking for, like "san pablo ave," "thai," and "albany."

**\*\*Final Optimized SQL Query:\*\***

```sql

SELECT COUNT(T1.id\_restaurant)

FROM {BQ\_PROJECT\_ID}.restaurant.generalinfo AS T1

INNER JOIN {BQ\_PROJECT\_ID}.restaurant.location AS T2 ON T1.id\_restaurant = T2.id\_restaurant

WHERE T1.food\_type = 'thai' AND T1.city = 'albany' AND T2.street\_name = 'san pablo ave'

```

=====

Example 2

\*\*\*\*\*

**【Database Info】**

CREATE TABLE {BQ\_PROJECT\_ID}.financial.account (

account\_id INT64,

district\_id INT64,

frequency STRING,

date DATE,

```
);
```

```
CREATE TABLE {BQ_PROJECT_ID}.financial.client (
```

```
    client_id INT64,
```

```
    gender STRING,
```

```
    birth_date DATE,
```

```
    district_id INT64,
```

```
);
```

```
CREATE TABLE {BQ_PROJECT_ID}.financial.district (
```

```
    district_id INT64,
```

```
    a4 STRING OPTIONS(description="Assuming A4 and A11 are strings due to  
examples"),
```

```
    a11 STRING,
```

```
);
```

```
*****
```

### **【Question】**

Question:

What is the gender of the youngest client who opened account in the lowest average salary branch? Given that Later birthdate refers to younger age; A11 refers to average salary

```
*****
```

### **【Answer】**

Repeating the question and generating the SQL with Recursive Divide-and-Conquer.

**\*\*Question\*\***: What is the gender of the youngest client who opened account in the lowest average salary branch? Given that Later birthdate refers to younger age; A11 refers to average salary

**\*\*Query Plan\*\***:

**\*\* Preparation Steps: \*\***

1. Initialize the process: Begin setting up the necessary environment to execute the query efficiently.
2. Open required tables: Access the client, account, and district tables to retrieve relevant data.
3. Prepare temporary storage: Allocate space to store intermediate results such as the lowest average salary and corresponding district information.

**\*\* Identify the Branch with Lowest Average Salary: \*\***

1. Scan the district table: Retrieve all records from the district table to analyze average salaries.
2. Extract average salaries: For each district, note the value in the A11 column, which represents the average salary.
3. Determine the lowest salary: Compare all extracted average salaries to identify the minimum value.
4. Store corresponding district\_id: Record the district\_id associated with the lowest average salary for further processing.

**\*\* Find Clients in the Identified District: \*\***

1. Join client and account tables: Merge records where client.client\_id matches account.account\_id to associate clients with their accounts.
2. Filter by district\_id: Select only those records where account.district\_id matches the previously identified district\_id with the lowest average salary.
3. Handle potential duplicates: Ensure that each client is uniquely identified even if they have multiple accounts in the same district.

**\*\* Identify the Youngest Client: \*\***

1. Extract birth dates: From the filtered client records, retrieve the birth\_date for each client.
2. Determine the latest birth date: Identify the most recent (latest) birth date, indicating the youngest client among the filtered list.
3. Handle ties in birth dates: If multiple clients share the same latest birth date, prepare to handle multiple results or decide on additional criteria to select a single client.

**\*\* Retrieve Gender Information: \*\***

1. Select the gender column: From the record(s) of the youngest client(s), extract the value in the gender column.
2. Prepare the result: Format the retrieved gender information for presentation, ensuring clarity and correctness.

**\*\* Finalize and Deliver the Result: \*\***

1. Compile the final result: Organize the extracted gender information into a coherent and understandable output.
2. Clean up resources: Close any open table connections and release temporary storage used during query execution.
3. Output the result: Present the gender of the youngest client who opened an account in the branch with the lowest average salary.

**\*\*Final Optimized SQL Query:\*\***

```sql

```
SELECT `T1`.`gender`

FROM `{BQ_PROJECT_ID}.financial.client` AS `T1`

INNER JOIN `{BQ_PROJECT_ID}.financial.district` AS `T2`

ON `T1`.`district_id` = `T2`.`district_id`

ORDER BY `T2`.`A11` ASC, `T1`.`birth_date` DESC NULLS LAST

LIMIT 1

```
```

=====

Example 3 (dividing into two parallel sub-questions)

\*\*\*\*\*

**【Database Info】**

```
CREATE TABLE {BQ_PROJECT_ID}.olympics.games

(

  id INT64,

  games_year INT64 OPTIONS(description="description: the year of the game"),
```

```
games_name STRING,  
  
);
```

```
CREATE TABLE {BQ_PROJECT_ID}.olympics.games_city  
  
(  
  
  games_id INT64,  
  
  city_id INT64 OPTIONS(description="the id of the city that held the game Maps  
to city(id)"),  
  
);
```

```
CREATE TABLE {BQ_PROJECT_ID}.olympics.city  
  
(  
  
  id INT64,  
  
  city_name STRING,  
  
);
```

\*\*\*\*\*

### **【Question】**

Question:

From 1900 to 1992, how many games did London host? From 1900 to 1992 refers to games\_year BETWEEN 1900 AND 1992; London refers to city\_name = 'London'; games refer to games\_name;

\*\*\*\*\*

### **【Answer】**

Repeating the question and generating the SQL with Recursive Divide-and-Conquer.

**\*\*Question\*\***: From 1900 to 1992, how many games did London host? From 1900 to 1992 refers to games\_year BETWEEN 1900 AND 1992; London refers to city\_name = 'London'; games refer to games\_name;

**\*\*Query Plan\*\*:**

**\*\* Preparation Steps: \*\***

1. Initialize the process: Set up the environment to begin query execution, including necessary variables and temporary storage.
2. Open required tables: Open the games\_city, city, and games tables to access relevant data.
3. Prepare filtering values: Set up the specific values to filter the data, such as the year range (1900-1992) and the city name 'London'.

**\*\* Filter and Identify Relevant Data: \*\***

1. Scan games\_city table: Retrieve records from the games\_city table to match games with the cities where they were hosted.
2. Fetch the corresponding city\_id: For each row in games\_city, extract the city\_id to find out which city hosted the game.
3. Match city\_id with city\_name: Use the city\_id to look up the corresponding city\_name in the city table.
4. Filter by city\_name = 'London': Select only those rows where the city\_name is 'London'.

**\*\* Further Filter by Year Range: \*\***

1. Extract games\_id: For rows that match 'London', retrieve the games\_id from the games\_city table.
2. Find matching games\_year: Use the games\_id to look up the corresponding games\_year in the games table.
3. Filter by games\_year between 1900 and 1992: Select only those rows where the games\_year falls within the specified range (1900-1992).

**\*\* Count the Matching Rows: \*\***

1. Initialize the count: Prepare to count the number of matching rows that meet all the criteria.
2. Count the valid entries: For each row that matches the conditions (city\_name = 'London' and games\_year between 1900 and 1992), increment the count.
3. Store the final count: Once all rows have been processed, store the total count as the final result.

**\*\* Finalize and Deliver the Result: \*\***

1. Prepare the result for output: Format the final count of games hosted by London between 1900 and 1992.
2. Output the final count: Deliver the count as the query result.
3. Clean up resources: Close any open table connections and release temporary storage used during query execution.

**\*\*Final Optimized SQL Query:\*\***

```sql

SELECT COUNT(T3.id)

FROM {BQ\_PROJECT\_ID}.olympics.games\_city AS T1

INNER JOIN {BQ\_PROJECT\_ID}.olympics.city AS T2 ON T1.city\_id = T2.id

INNER JOIN {BQ\_PROJECT\_ID}.olympics.games AS T3 ON T1.games\_id = T3.id

WHERE T2.city\_name = 'London' AND T3.games\_year

BETWEEN 1900 AND 1992

```

=====

Example 4

\*\*\*\*\*

**【Database Info】**

CREATE TABLE {BQ\_PROJECT\_ID}.retails.employees (

employee\_id INT64,

department\_id INT64,

salary INT64,

);



\*\*\*\*\*

### **【Question】**

Question:

How many employees earn over \$100,000?

\*\*\*\*\*

### **【Answer】**

Repeating the question and generating the SQL with Recursive Divide-and-Conquer.

**\*\*Question:\*\*** How many employees earn over \$100,000?

**\*\* Query Plan\*\*:**

**\*\* Preparation Steps: \*\***

1. Initialize the process: Begin by setting up the environment for query execution, including initializing variables and temporary storage.

2. Open the employees table: Access the employees table to retrieve the relevant data.

**\*\* Filtering Employees by Salary: \*\***

1. Scan the employees table: Begin reading rows from the employees table.

2. Fetch the salary column: For each row, retrieve the value from the salary column.

3. Compare salary against \$100,000: Check if the salary value is greater than \$100,000.

4. Identify matching rows: For rows where the salary exceeds \$100,000, prepare to count these entries.

**\*\* Counting the Matches: \*\***

1. Initialize the count: Set up a counter to keep track of how many employees meet the salary condition.

2. Increment the count: For each row where the salary is above \$100,000, increment the counter.

3. Store the final count: Once all rows have been processed, store the total count of matching employees.

**\*\* Finalize and Deliver the Result: \*\***

1. Prepare the result for output: Format the final count for presentation.
2. Output the final count: Deliver the count as the query result, indicating how many employees earn over \$100,000.
3. Clean up resources: Close the employees table and release any temporary storage used during query execution.

**\*\*Final Optimized SQL Query:\*\***

```sql

```
SELECT COUNT(*) FROM {BQ_PROJECT_ID}.retails.employees WHERE salary > 100000;
```

```

=====

Example 6

\*\*\*\*\*

**【Database Info】**

```
CREATE TABLE {BQ_PROJECT_ID}.airlines.Airlines
```

```
(
```

```
  FL_DATE STRING OPTIONS(description="flight date"),
```

```
  ORIGIN STRING OPTIONS(description="airport of origin"),
```

```
  DEST STRING OPTIONS(description="Destination airport"),
```

```
);
```

```
CREATE TABLE {BQ_PROJECT_ID}.airlines.Airports
```

```
(
```

```
  Code STRING,
```

```
Description STRING,  
);
```

\*\*\*\*\*

### 【Question】

Question:

How many flights were there from San Diego International airport to Los Angeles International airport in the August of 2018?

flights from refers to ORIGIN; San Diego International airport refers to Description = 'San Diego, CA: San Diego International'; flights to refers to DEST; Los Angeles International airport refers to Description = 'Los Angeles, CA: Los Angeles International'; in the August of 2018 refers to FL\_DATE like '2018/8%';

\*\*\*\*\*

### 【Answer】

\*\*Repeating the question and generating the SQL with Recursive Divide-and-Conquer.\*\*

**\*\*Question\*\*:** How many flights were there from San Diego International airport to Los Angeles International airport in the August of 2018? flights from refers to ORIGIN; San Diego International airport refers to Description = 'San Diego, CA: San Diego International'; flights to refers to DEST; Los Angeles International airport refers to Description = 'Los Angeles, CA: Los Angeles International'; in the August of 2018 refers to FL\_DATE like '2018/8%';

**\*\* Query Plan\*\*:**

**\*\* Preparation Steps: \*\***

1. Initialize the process: Set up the environment and prepare for query execution by initializing variables and temporary storage.

2. Open required tables: Open the Airlines and Airports tables to access relevant data.

**\*\* Identify Relevant Flights: \*\***

1. Fetch the FL\_DATE column: Start reading the FL\_DATE column from the Airlines table.
2. Filter by August 2018: Use the condition FL\_DATE LIKE '2018/8%' to filter flights that occurred in August 2018.
3. Join with Airports for ORIGIN: Identify flights originating from 'San Diego, CA: San Diego International' by joining the Airlines table with the Airports table on the ORIGIN field.
4. Join with Airports for DEST: Similarly, identify flights destined for 'Los Angeles, CA: Los Angeles International' by joining the Airlines table with the Airports table on the DEST field.

**\*\* Count the Matching Flights: \*\***

1. Initialize the count: Set up a counter to keep track of how many flights match the criteria.
2. Increment the count: For each flight that meets the conditions (originating from San Diego International and destined for Los Angeles International in August 2018), increment the counter.
3. Store the final count: Once all rows have been processed, store the total count of matching flights.

**\*\* Finalize and Deliver the Result: \*\***

1. Prepare the result for output: Format the final count for presentation, ensuring clarity and correctness.
2. Output the final count: Deliver the count as the query result, indicating how many flights met the specified criteria.
3. Clean up resources: Close any open table connections and release temporary storage used during query execution.

**\*\*Final Optimized SQL Query:\*\***

```sql

SELECT COUNT(FL\_DATE)

FROM {BQ\_PROJECT\_ID}.airlines.Airlines

WHERE FL\_DATE LIKE '2018/8%'

AND ORIGIN = (

SELECT DISTINCT T2.ORIGIN

```

FROM {BQ_PROJECT_ID}.airlines.Airports AS T1

INNER JOIN {BQ_PROJECT_ID}.airlines.Airlines AS T2 ON T1.Code = T2.ORIGIN

WHERE T1.Description = 'San Diego, CA: San Diego International' )

AND DEST = (

SELECT DISTINCT T4.DEST

FROM {BQ_PROJECT_ID}.airlines.Airports AS T3

INNER JOIN {BQ_PROJECT_ID}.airlines.Airlines AS T4 ON T3.Code = T4.DEST

WHERE T3.Description = 'Los Angeles, CA: Los Angeles International' )

...

```

=====

Example 7

\*\*\*\*\*

【Database Info】

```
CREATE TABLE {BQ_PROJECT_ID}.food_inspection.businesses
```

```

(
    `business_id` INT64,
    `name` STRING OPTIONS(description="the name of the eatery"),
);

```

```
CREATE TABLE {BQ_PROJECT_ID}.food_inspection.inspections
```

```

(
    `business_id` INT64 OPTIONS(description="the unique id of the
business"),
    `score` INT64 OPTIONS(description="description: the inspection score"),
    `date` DATE,
);

```

```
CREATE TABLE {BQ_PROJECT_ID}.food_inspection.violations

(

    `business_id` INT64,

    `date` DATE,

);
```

\*\*\*\*\*

### 【Question】

Question:

What are the names of the establishments that met all the required standards for 4 consecutive years?

establishment has the same meaning as business; score of 90 or more refers to score  $\geq 90$ ; year(date) = 2015; ; met all required standards for 4 consecutive years refers to COUNT(year(date)) = 4 where score = 100;

\*\*\*\*\*

### 【Answer】

Repeating the question and generating the SQL with Recursive Divide-and-Conquer.

**\*\*Question\*\*:** What are the names of the establishments that met all the required standards for 4 consecutive years? establishment has the same meaning as business; score of 90 or more refers to score  $\geq 90$ ; year(date) = 2015; ; met all required standards for 4 consecutive years refers to COUNT(year(date)) = 4 where score = 100;

**\*\* Query Plan\*\*:**

**\*\* Preparation Steps: \*\***

1. Initialize the process: Set up the environment and prepare for query execution, including initializing variables and temporary storage.
2. Open required tables: Open the businesses, inspections, and violations tables to access relevant data.

**\*\* Filter and Identify Relevant Inspections: \*\***

1. Scan the inspections table: Start reading rows from the inspections table.
2. Filter by score of 100: Select only those inspections where the score is 100, indicating that the establishment met all required standards.
3. Extract year from the inspection date: Use the `FORMAT_DATE('%Y', date)` function to extract the year from the inspection date.
4. Join with businesses table: Match each inspection to the corresponding business by joining on `business_id`.

**\*\* Identify Businesses Meeting Standards for 4 Consecutive Years: \*\***

1. Aggregate by business and year: Group the data by business name and the extracted year to count the number of years each business met the required standards.
3. Apply row numbering: Use `ROW_NUMBER()` with a partition by business name and order by year to identify consecutive years.
3. Filter for 4 consecutive years: Group by business name and ensure that the count of years with the required score is exactly 4, indicating 4 consecutive years of meeting the standards.

**\*\* Count and Finalize the Results: \*\***

1. Count the matching businesses: For each business, count the number of years that meet the criteria.
2. Select distinct business names: Extract the names of businesses that have met the required standards for 4 consecutive years.
3. Store and prepare the result: Once all businesses have been processed, store the result and prepare it for output.

**\*\* Deliver the Final Result: \*\***

1. Prepare the result for output: Format the final list of business names for presentation.
2. Output the final result: Deliver the names of the businesses that met the required standards for 4 consecutive years.
3. Clean up resources: Close any open table connections and release temporary storage used during query execution.

**\*\*Final Optimized SQL Query:\*\***

```sql

SELECT DISTINCT T4.name

FROM ( SELECT T3.name, T3.years, row\_number()

OVER (PARTITION BY T3.name ORDER BY T3.years)

AS rowNumber FROM ( SELECT DISTINCT name, FORMAT\_DATE('%Y', date)

AS years FROM {BQ\_PROJECT\_ID}.food\_inspection.inspections AS T1

INNER JOIN {BQ\_PROJECT\_ID}.food\_inspection.businesses AS T2 ON T1.business\_id  
= T2.business\_id

WHERE T1.score = 100 ) AS T3 ) AS T4

GROUP BY T4.name, DATE\_SUB (DATE (CONCAT (T4.years, '-01-01')), INTERVAL  
(T4.rowNumber - 1) YEAR) HAVING COUNT (T4.years) = 4

```

=====

Example 8

\*\*\*\*\*

**【Database Info】**

CREATE TABLE `bigquery-public-  
data.covid19\_symptom\_search.symptom\_search\_sub\_region\_2\_daily`

(

country\_region\_code STRING,

country\_region STRING,

sub\_region\_1 STRING,

sub\_region\_1\_code STRING,

sub\_region\_2 STRING,

sub\_region\_2\_code STRING,

place\_id STRING,

date DATE,



```

    symptom_Abdominal_obesity FLOAT64,

    symptom_Abdominal_pain FLOAT64,

    symptom_Acne FLOAT64

)

PARTITION BY date

CLUSTER BY country_region_code, sub_region_1_code, sub_region_2_code,
sub_region_2;

```

\*\*\*\*\*

### 【Question】

Question:

Find the day in which the symptom that occurs most frequently is headache.

\*\*\*\*\*

### 【Answer】

Repeating the question and generating the SQL with Recursive Divide-and-Conquer.

**\*\*Question\*\*:** Find the day in which the symptom that occurs most frequently is headache.

**\*\* Query Plan\*\*:**

**\*\* Preparation Steps : \*\***

1. Initialize the process: Set up the environment and prepare for query execution, including initializing variables and temporary storage.

2. Open the symptom\_search\_sub\_region\_2\_daily table: Access the table containing daily symptom search data.

**\*\* Extract the headache symptom: \*\***

1. Scan the table: Start reading rows from the

symptom\_search\_sub\_region\_2\_daily table.

2. Identify the headache symptom: Look for the column containing the headache symptom data.

3. Extract the headache symptom value: For each row, extract the value from the headache symptom column.

4. Aggregate by date: Group the data by date to count the occurrences of the headache symptom on each day.

**\*\* Sort by frequency: \*\***

1. Order the results in descending order of symptom frequency.

2. Limit the results: Extract the single day with the highest count.

**\*\* Step 2: Identify Subtasks \*\***

1. Extract relevant symptom column: While "headache" is not explicitly listed, its frequency might be tracked in a related table (e.g., symptom\_search\_country\_daily) as per the given gold query.

2. Group data by day of the week: Use `FORMAT_DATE('%A', date)` to extract the day of the week from each date.

3. Aggregate by count: Count the occurrences of the "headache" symptom across dates and group by the day of the week.

4. Sort by frequency: Order the results in descending order of symptom frequency.

5. Limit the results: Extract the single day with the highest count.

**\*\* Step 3: Formulate the Query \*\***

1. From the subtasks, the query will:

2. Select the day of the week using `FORMAT_DATE('%A', date)`.

3. Aggregate counts grouped by the day.

4. Sort the results by the aggregated count in descending order.

5. Limit the results to the top record.

**\*\* Step 4: Construct the Query \*\***

1. Combining all subtasks, the final query is:

```
2. SELECT COUNT(symptom_headache) FROM
{BQ_PROJECT_ID}.covid19_symptom_search.symptom_search_sub_region_2_daily GROUP
BY FORMAT_DATE('%A', date) ORDER BY COUNT(symptom_headache) DESC LIMIT 1;
```

**\*\* Step 5: Finalize the Query \*\***

**\*\*Final Optimized SQL Query:\*\***

**```sql**

SELECT

FORMAT\_DATE('%A', PARSE\_DATE('%Y-%m-%d', date)) AS day,

COUNT(\*) AS headache\_count

FROM

{BQ\_PROJECT\_ID}.covid19\_symptom\_search.symptom\_search\_country\_daily

GROUP BY

day

ORDER BY

headache\_count DESC

LIMIT 1;

**```**

Now is the real question, following the instruction and examples, generate the GoogleSQL with Recursive Divide-and-Conquer approach.

Follow all steps from the strategy. When you get to the final query, output the query string ONLY in the format **```sql ... ```**. Make sure you only output one single query.

\*\*\*\*\*

**【Table creation statements】**

{SCHEMA}

\*\*\*\*\*

**【Question】**

Question:

{QUESTION}

\*\*\*\*\*

**【Answer】**

Repeating the question and generating the SQL with Recursive Divide-and-Conquer.

"""

=====

File: data\_science/sub\_agents/bigquery/chase\_sql/sql\_postprocessor/README.md

=====

# Support Post-processing of SQL after Agentic Generation.

This tool provides a way to postprocess the SQL generated by the agent.

Currently, it supports the following post-processing steps: 1. Translation of the SQL from SQLite to BigQuery. 2. Correction of errors in the SQL before and after translation.

## Usage

Currently, the post-processing is done within the `chase\_db\_tools.py` agent. To

use this agent, first configure the database agent to use Chase SQL.

Then, to use the postprocessing, set the `transpile\_to\_bigquery` argument to `True` in the `chase\_nl2sql` function. Optionally, you can also set the

`process\_input\_errors` and `process\_tool\_output\_errors` arguments to `True` to have the postprocessor correct errors in the SQL before and after translation.

### Current Defaults:

- Model: gemini-2.0-flash-001
- Temperature: 0.5
- Number of candidates: 1
- transpile\_to\_bigquery: True
- process\_input\_errors: False
- process\_tool\_output\_errors: False

=====

File: data\_science/sub\_agents/bigquery/chase\_sql/sql\_postprocessor/\_\_init\_\_.py

=====

# Copyright 2025 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");

# you may not use this file except in compliance with the License.

# You may obtain a copy of the License at

#

# <http://www.apache.org/licenses/LICENSE-2.0>

#

# Unless required by applicable law or agreed to in writing, software

# distributed under the License is distributed on an "AS IS" BASIS,

# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

# See the License for the specific language governing permissions and

```
# limitations under the License.
```

```
=====
```

```
File: data_science/sub_agents/bigquery/chase_sql/sql_postprocessor/  
correction_prompt_template.py
```

```
=====
```

```
# Copyright 2025 Google LLC
```

```
#
```

```
# Licensed under the Apache License, Version 2.0 (the "License");
```

```
# you may not use this file except in compliance with the License.
```

```
# You may obtain a copy of the License at
```

```
#
```

```
# http://www.apache.org/licenses/LICENSE-2.0
```

```
#
```

```
# Unless required by applicable law or agreed to in writing, software
```

```
# distributed under the License is distributed on an "AS IS" BASIS,
```

```
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
```

```
# See the License for the specific language governing permissions and
```

```
# limitations under the License.
```

```
"""Prompt template for making any corrections to the translation of SQL."""
```

```
CORRECTION_PROMPT_TEMPLATE_V1_0 = """
```

```
You are an expert in multiple databases and SQL dialects.
```

```
You are given a SQL query that is formatted for the SQL dialect:
```

```
{sql_dialect}
```

The SQL query is:

```
{sql_query}

{schema_insert}
```

This SQL query could have the following errors:

```
{errors}
```

Please correct the SQL query to make sure it is formatted correctly for the SQL dialect:

```
{sql_dialect}
```

DO not change any table or column names in the query. However, you may qualify column names with table names.

Do not change any literals in the query.

You may *only* rewrite the query so that it is formatted correctly for the specified SQL dialect.

Do not return any other information other than the corrected SQL query.

Corrected SQL query:

"""

=====

File: data\_science/sub\_agents/bigquery/chase\_sql/sql\_postprocessor/  
sql\_translator.py

=====

```
# Copyright 2025 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");
```

```
# you may not use this file except in compliance with the License.

# You may obtain a copy of the License at

#

#     http://www.apache.org/licenses/LICENSE-2.0

#

# Unless required by applicable law or agreed to in writing, software

# distributed under the License is distributed on an "AS IS" BASIS,

# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

# See the License for the specific language governing permissions and

# limitations under the License.
```

```
"""Translator from SQLite to BigQuery."""
```

```
import re
```

```
from typing import Any, Final
```

```
import regex
```

```
import sqlglot
```

```
import sqlglot.optimizer
```

```
from ..llm_utils import GeminiModel # pylint: disable=g-importing-member
```

```
from .correction_prompt_template import (
```

```
    CORRECTION_PROMPT_TEMPLATE_V1_0,
```

```
) # pylint: disable=g-importing-member
```

```
ColumnSchemaType = tuple[str, str]
```

```
AllColumnsSchemaType = list[ColumnSchemaType]
```

```
TableSchemaType = tuple[str, AllColumnsSchemaType]
```



```
DDLSchemaType = list[TableSchemaType]
```

```
SQLGlotColumnsDictType = dict[str, str]
```

```
SQLGlotSchemaType = dict[str, Any]
```

```
BirdSampleType = dict[str, Any]
```

```
def _isinstance_list_of_str_tuples_lists(obj: Any) -> bool:
```

```
    """Checks if the object is a list of tuples or listsof strings."""
```

```
    return (
```

```
        isinstance(obj, list)
```

```
        and all([isinstance(v, (tuple, list)) for v in obj])
```

```
        and all([isinstance(v[0], str) and isinstance(v[1], str) for v in  
obj])
```

```
    )
```

```
def _isinstance_ddl_schema_type(obj: Any) -> bool:
```

```
    """Checks if the object is a DDL schema type."""
```

```
    # pylint: disable=g-complex-comprehension
```

```
    return (
```

```
        isinstance(obj, list)
```

```
        and all(
```

```
            # Every element is a tuple or list.
```

```
            [isinstance(v, (tuple, list)) for v in obj]
```

```
        )
```

```
        and all(
```

```
            # First element is a string (table name) and
```

```

        # second element is a list (of tuples or lists).

        [isinstance(v[0], str) and isinstance(v[1], list) for v in obj]

    )

    and all(

        # Every element of above list is a tuple or list of strings

        # (column name, column type)

        [_isinstance_list_of_str_tuples_lists(v[1]) for v in obj]

    )

)

# pylint: enable=g-complex-comprehension

```

```

def _isinstance_sqlglot_schema_type(obj: Any) -> bool:

    """Checks if the object is a SQLGlot schema type."""

    # pylint: disable=g-complex-comprehension

    return (

        isinstance(obj, dict)

        and all([isinstance(v, dict) for v in obj.values()])

        and all([isinstance(c, str) for d in obj.values() for c, _ in
d.items()])

        and all([isinstance(t, str) for d in obj.values() for _, t in
d.items()])

    )

    # pylint: enable=g-complex-comprehension

```

```

def _isinstance_bird_sample_type(obj: Any) -> bool:

    """Checks if the object is a SQLGlot schema type."""

    return isinstance(obj, dict) and not _isinstance_sqlglot_schema_type(obj)

```

```
class SqlTranslator:
```

```
    """Translator from SQLite to BigQuery.
```

```
    This class is used to translate SQL queries from an input SQL dialect like  
    SQLite to an output SQL dialect like BigQuery. It uses the SQLGlot library  
    as
```

```
    a tool to perform the translation.
```

```
    The translation is done by the following steps:
```

1. (Optional) If there are errors in the input SQL query, the input SQL  
query  
is first modified by the LLM to address the errors.
2. The input SQL query is then translated to a SQL query in the output SQL  
dialect by the tool.
3. (Optional) If there are errors in the tool output SQL query, the tool  
output SQL query is modified by the LLM to address the errors.

```
Class Attributes:
```

```
    INPUT_DIALECT: The input SQL dialect.
```

```
    OUTPUT_DIALECT: The output SQL dialect.
```

```
Attributes:
```

```
    sql_query: The SQL query to translate.
```

```
    model: The model object, or the name of the model to use for the LLM.
```

```
    temperature: The temperature to use for the LLM.
```

```
    process_input_errors: True if any errors in the input SQL query should  
be  
    processed by the LLM.
```

```

        process_tool_output_errors: True if any errors in the tool output SQL
query

        should be processed by the LLM.

"""

INPUT_DIALECT: Final[str] = "sqlite"

OUTPUT_DIALECT: Final[str] = "bigquery"

def __init__(
    self,

    model: str | GeminiModel = "gemini-2.0-flash-001",

    temperature: float = 0.5,

    process_input_errors: bool = False,

    process_tool_output_errors: bool = False,

):
    """Initializes the translator."""

    self._process_input_errors: bool = process_input_errors

    self._process_tool_output_errors: bool = process_tool_output_errors

    self._input_errors: str | None = None

    self._tool_output_errors: str | None = None

    self._temperature: float = temperature

    if isinstance(model, str):
        self._model = GeminiModel(model_name=model,
temperature=self._temperature)

    else:

        self._model = model

    @classmethod

    def _parse_response(cls, text: str) -> str | None:

        """Extracts the SQL query from the response text."""

```

```

pattern = r"```\sql(.*)```\s"

match = re.search(pattern, text, re.DOTALL)

if match:

    return match.group(1).strip()

return None


@classmethod

def _apply_heuristics(cls, sql_query: str) -> str:

    """Applies heuristics to the SQL query."""

    if "'" in sql_query:

        sql_query = sql_query.replace("'", "\'")

    return sql_query


@classmethod

def _extract_schema_from_ddl_statement(cls, ddl_statement: str) ->
TableSchemaType:

    """Extracts the schema from a single DDL statement."""

    # Split the DDL statement into table name and columns.

    # Match the following pattern:

    # CREATE [OR REPLACE] TABLE [<table_name>] (<all_columns>);

    splitter_pattern = (

        # CREATE [OR REPLACE] TABLE

        r"^\s*CREATE\s+(?:OR\s+REPLACE\s+)?TABLE\s+"

        # Match the table name, optionally surrounded by backticks.

        r"(?:`)?(?:P<table_name>[\w\d\-\_\.\s])?(?:`)?\s*"

        # Match the column name as everything between the first and last

        # parentheses followed by a semicolon.

        r"\(((?P<all_columns>.*))\);$"

    )

```

```

split_match = regex.search(

    splitter_pattern,

    ddl_statement,

    flags=re.DOTALL | re.VERBOSE | re.MULTILINE,

)

if not split_match:

    return None, None


table_name = split_match.group("table_name")

all_columns = split_match.group("all_columns").strip()

if not table_name or not all_columns:

    return None, None


# Extract the columns from the DDL statement.

# Match the following pattern:

# <column_name> <column_type> [<ignored_text>]

# [, <column_name> <column_type> [<ignored_text>]]*

# Ignore any comments. Ignore any INSERT INTO statements. Ignore any

# lines beginning with a parenthesis (these are example values).

column_pattern = (

    # Ignore any comments.

    r"\s*--.*(*SKIP) (*FAIL) "

    # Ignore any INSERT INTO statements.

    r"|\s*INSERT\s+INTO.*(*SKIP) (*FAIL) "

    # Ignore any lines beginning with a parenthesis.

    r"|\s*\(. (*SKIP) (*FAIL) "

    # Match the column name and type, optionally with backticks.

    r"|\s*(?:`)?\s*(?P<column_name>\w+)(?:`)?\s+(?P<column_type>\w+).*"

```

```

    ) # (?:,)?

    columns = regex.findall(column_pattern, all_columns, flags=re.VERBOSE)

    return table_name, columns

@classmethod
def extract_schema_from_ddls(cls, ddls: str) -> DDLSchemaType:

    """Extracts the schema from multiple DDL statements."""

    ddl_statements = ddls.split(";\\n")

    ddl_statements = [ddl.strip() for ddl in ddl_statements if
ddl.strip()]

    schema = []

    for ddl_statement in ddl_statements:

        if ddl_statement:

            ddl_statement = ddl_statement.strip() + ";" # Add the
semicolon back.

            table_name, columns = cls._extract_schema_from_ddl_statement(

                ddl_statement

            )

            if table_name and columns:

                schema.append((table_name, columns))

    return schema

@classmethod
def _get_schema_from_bird_sample(

    cls, sample: BirdSampleType

) -> dict[str, dict[str, str]]:

    """Returns the schema from the Bird dataset example."""

    col_types_map: dict[str, str] = {

        "text": "TEXT",

```

```

        "number": "FLOAT",

        "date": "DATE",

        "datetime": "DATETIME",

        "time": "TIME",

        "timestamp": "TIMESTAMP",

        "bool": "BOOL",

    }

    tables = sample["db_table_names"]

    table_ids = sample["db_column_names"]["table_id"][1:]

    column_names = sample["db_column_names"]["column_name"][1:]

    column_types = sample["db_column_types"][1:]

    column_types = [col_types_map[col_type] for col_type in column_types]

    assert len(column_names) == len(column_types)

    cols_and_types: list[tuple[str, str]] = list(zip(column_names,
column_types))

    tables_to_columns: dict[str, dict[str, str]] = {}

    for id_pos, table_id in enumerate(table_ids):

        if tables[table_id] in tables_to_columns.keys():

            tables_to_columns[tables[table_id]].update(

                dict([cols_and_types[id_pos]])

            )

        else:

            tables_to_columns[tables[table_id]] =
dict([cols_and_types[id_pos]])

    return tables_to_columns

    @classmethod

    def _get_table_parts(cls, table_name: str) -> tuple[str | None, str |
None, str]:

        """Returns the table parts from the table name."""

```



```

        table_parts = table_name.split(".")

        if len(table_parts) == 3:

            return table_parts

        elif len(table_parts) == 2:

            return None, *table_parts

        elif len(table_parts) == 1:

            return None, None, *table_parts

        else:

            raise ValueError(f"Invalid table name: {table_name}")

    @classmethod

    def format_schema(cls, schema: DDLSchemaType) -> SQLGlotSchemaType:

        """Formats the DDL schema for use in SQLGlot."""

        schema_dict = {}

        catalog, db = None, None

        for table_name, columns in schema:

            catalog, db, table_name = cls._get_table_parts(table_name)

            schema_dict[table_name] = {}

            for column_name, column_type in columns:

                schema_dict[table_name][column_name] = column_type

        if db:

            schema_dict = {db: schema_dict}

        if catalog:

            schema_dict = {catalog: schema_dict}

        return schema_dict

    @classmethod

    def rewrite_schema_for_sqlglot(

        cls, schema: str | SQLGlotSchemaType | BirdSampleType

```

```

) -> SQLGlotSchemaType:

    """Rewrites the schema for use in SQLGlot."""

    schema_dict = None

    if schema:

        if isinstance(schema, str):

            schema = cls.extract_schema_from_ddls(schema)

            schema_dict = cls.format_schema(schema)

        elif _isinstance_sqlglot_schema_type(schema):

            schema_dict = schema

        elif _isinstance_bird_sample_type(schema):

            schema_dict = cls._get_schema_from_bird_sample(schema)

        elif _isinstance_ddl_schema_type(schema):

            schema_dict = cls.format_schema(schema)

        else:

            raise TypeError(f"Unsupported schema type: {type(schema)}")

    return schema_dict


@classmethod
def _check_for_errors(

    cls,

    sql_query: str,

    sql_dialect: str,

    db: str | None = None,

    catalog: str | None = None,

    schema_dict: SQLGlotSchemaType | None = None,

) -> tuple[str | None, str]:

    """Checks for errors in the SQL query.
```

Args:

sql\_query: The SQL query to check for errors.

sql\_dialect: The SQL dialect of the SQL query.

db: The database to use for the translation. This field is optional.

catalog: The catalog to use for the translation. `catalog` is the SQLGlot

term for the project ID. This field is optional.

schema\_dict: The DDL schema to use for the translation. The DDL format is

in the SQLGlot format. This field is optional.

Returns:

tuple of the errors in the SQL query, or None if there are no errors, and

the SQL query after optimization.

"""

try:

# First, try to parse the SQL query into a SQLGlot AST.

sql\_query\_ast = sqlglot.parse\_one(

sql=sql\_query,

read=sql\_dialect.lower(),

error\_level=sqlglot.ErrorLevel.IMMEDIATE,

)

# Then add the database and catalog information for each table to the AST.

for table in sql\_query\_ast.find\_all(sqlglot.exp.Table):

table.set("catalog", sqlglot.exp.Identifier(this=catalog, quoted=True))

table.set("db", sqlglot.exp.Identifier(this=db, quoted=True))

# Then, try to optimize the SQL query.

sql\_query\_ast = sqlglot.optimizer.optimize(

```

        sql_query_ast,

        dialect=sql_dialect.lower(),

        schema=schema_dict,

        db=db,

        catalog=catalog,

        error_level=sqlglot.ErrorLevel.IMMEDIATE,

    )

    sql_query = sql_query_ast.sql(sql_dialect.lower())

except sqlglot.errors.SqlglotError as e:

    return str(e), sql_query

return None, sql_query


def _fix_errors(

    self,

    sql_query: str,

    sql_dialect: str,

    apply_heuristics: bool,

    db: str | None = None,

    catalog: str | None = None,

    ddl_schema: str | SQLGlotSchemaType | BirdSampleType | None = None,

    number_of_candidates: int = 1,

) -> str:

    """Fixes errors in the SQL query.

    Args:

        sql_query: The SQL query to fix.

        sql_dialect: The input SQL dialect.

        apply_heuristics: True if the heuristics should be applied.

        db: The database to use for the translation. This field is optional.

```

catalog: The catalog to use for the translation. `catalog` is the SQLGlot term for the project ID. This field is optional.

ddl\_schema: The DDL schema to use for the translation. The DDL format can be the SQLGlot format, the DDL schema format, a Bird dataset example, or a string containing multiple DDL statements. This field is optional.

number\_of\_candidates: The number of candidates to generate, default is 1.

Returns:

str: The fixed SQL query.

"""

if apply\_heuristics:

sql\_query = self.\_apply\_heuristics(sql\_query)

# Reformat the schema if provided. This will remove any comments and

# `INSERT INTO` statements.

schema\_dict = self.rewrite\_schema\_for\_sqlglot(ddl\_schema)

errors\_and\_sql: tuple[str | None, str] = self.\_check\_for\_errors(

sql\_query=sql\_query,

sql\_dialect=self.OUTPUT\_DIALECT,

db=db,

catalog=catalog,

schema\_dict=schema\_dict,

)

errors, sql\_query = errors\_and\_sql

responses = sql\_query # Default to the input SQL query after error check.

if errors:

```

print("Processing input errors")

if schema_dict:

    # If the schema is provided, then insert it into the prompt.

    schema_insert = f"\nThe database schema is:\n{schema_dict}\n"

else:

    schema_insert = "\n"

prompt: str = CORRECTION_PROMPT_TEMPLATE_V1_0.format(

    sql_dialect=sql_dialect.lower(),

    errors=errors,

    sql_query=sql_query,

    schema_insert=schema_insert,

)

requests: list[str] = [prompt for _ in
range(number_of_candidates)]

responses: list[str] = self._model.call_parallel(

    requests, parser_func=self._parse_response

)

if responses:

    # We only use the first response. Therefore the
`number_of_candidates`

    # parameter is not used.

    # pylint: disable=g-bad-todo

    # pylint: enable=g-bad-todo

    # First, find the first non-None response.

    responses = [r for r in responses if r is not None]

    if responses:

        # Then, return the first non-None response.

        responses = responses[0]

return responses

```

```

def translate(
    self,

    sql_query: str,

    db: str | None = None,

    catalog: str | None = None,

    ddl_schema: str | SQLGlotSchemaType | BirdSampleType | None = None,

) -> str:

    """Translates the SQL query to the output SQL dialect.

    Args:

        sql_query: The SQL query to translate.

        db: The database to use for the translation. This field is optional.

        catalog: The catalog to use for the translation. `catalog` is the
SQLGlot

        term for the project ID. This field is optional.

        ddl_schema: The DDL schema to use for the translation. The DDL
format can

        be the SQLGlot format or the DDL schema format. This field is
optional.

    Returns:

        The translated SQL query.

    """

    print("***** sql_query at translator entry:", sql_query)

    if self._process_input_errors:

        sql_query = self._fix_errors(

            sql_query,

            db=db,

            catalog=catalog,

            sql_dialect=self.OUTPUT_DIALECT,

```

```

        ddl_schema=ddl_schema,

        apply_heuristics=True,

    )

print("***** sql_query after fix_errors:", sql_query)

sql_query = sqlglot.transpile(

    sql=sql_query,

    read=self.INPUT_DIALECT,

    write=self.OUTPUT_DIALECT,

    error_level=sqlglot.ErrorLevel.IMMEDIATE,

)[

    0

] # Transpile returns a list of strings.

print("***** sql_query after transpile:", sql_query)

if self._tool_output_errors:

    sql_query = self._fix_errors(

        sql_query,

        db=db,

        catalog=catalog,

        sql_dialect=self.OUTPUT_DIALECT,

        ddl_schema=ddl_schema,

        apply_heuristics=True,

    )

sql_query = sql_query.strip().replace("'", "`")

sql_query = self._apply_heuristics(sql_query)

return sql_query

```



=====

File: data\_science/sub\_agents/bqml/\_\_init\_\_.py

=====

# Copyright 2025 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");

# you may not use this file except in compliance with the License.

# You may obtain a copy of the License at

#

# <http://www.apache.org/licenses/LICENSE-2.0>

#

# Unless required by applicable law or agreed to in writing, software

# distributed under the License is distributed on an "AS IS" BASIS,

# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

# See the License for the specific language governing permissions and

# limitations under the License.

=====

File: data\_science/sub\_agents/bqml/agent.py

=====

# Copyright 2025 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");

# you may not use this file except in compliance with the License.

```

# You may obtain a copy of the License at

#

#     http://www.apache.org/licenses/LICENSE-2.0

#

# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""Data Science Agent V2: generate nl2py and use code interpreter to run the
code."""

import os

from google.adk.agents import Agent

from google.adk.tools import ToolContext

from google.adk.tools.agent_tool import AgentTool

from google.adk.agents.callback_context import CallbackContext


from data_science.sub_agents.bqml.tools import (

    check_bq_models,

    execute_bqml_code,

    rag_response,

)

from .prompts import return_instructions_bqml


from data_science.sub_agents.bigquery.agent import database_agent as
bq_db_agent

```

```

from data_science.sub_agents.bigquery.tools import (

    get_database_settings as get_bq_database_settings,

)

def setup_before_agent_call(callback_context: CallbackContext):

    """Setup the agent."""

    # setting up database settings in session.state

    if "database_settings" not in callback_context.state:

        db_settings = dict()

        db_settings["use_database"] = "BigQuery"

        callback_context.state["all_db_settings"] = db_settings

    # setting up schema in instruction

    if callback_context.state["all_db_settings"]["use_database"] ==
    "BigQuery":

        callback_context.state["database_settings"] =
        get_bq_database_settings()

        schema = callback_context.state["database_settings"]["bq_ddl_schema"]

        callback_context._invocation_context.agent.instruction = (

            return_instructions_bqml()

            + f"""

</BQML Reference for this query>

<The BigQuery schema of the relevant data with a few sample rows>

{schema}

</The BigQuery schema of the relevant data with a few sample rows>

```

```

"""

)

async def call_db_agent(

    question: str,

    tool_context: ToolContext,

):

    """Tool to call database (nl2sql) agent."""

    print(

        "\n call_db_agent.use_database:"

        f' {tool_context.state["all_db_settings"]["use_database"]}'

    )

    database_agent = (

        bq_db_agent

        if tool_context.state["all_db_settings"]["use_database"] == "BigQuery"

        # else pg_db_agent

        else None

    )

    agent_tool = AgentTool(agent=database_agent)

    db_agent_output = await agent_tool.run_async(

        args={"request": question}, tool_context=tool_context

    )

    tool_context.state["db_agent_output"] = db_agent_output

    return db_agent_output


root_agent = Agent(

```

```
model=os.getenv("BQML_AGENT_MODEL"),

name="bq_ml_agent",

instruction=return_instructions_bqml(),

before_agent_callback=setup_before_agent_call,

tools=[execute_bqml_code, check_bq_models, call_db_agent, rag_response],

)
```

```
=====
```

```
File: data_science/sub_agents/bqml/prompts.py
```

```
=====
```

```
# Copyright 2025 Google LLC
```

```
#
```

```
# Licensed under the Apache License, Version 2.0 (the "License");
```

```
# you may not use this file except in compliance with the License.
```

```
# You may obtain a copy of the License at
```

```
#
```

```
# http://www.apache.org/licenses/LICENSE-2.0
```

```
#
```

```
# Unless required by applicable law or agreed to in writing, software
```

```
# distributed under the License is distributed on an "AS IS" BASIS,
```

```
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
```

```
# See the License for the specific language governing permissions and
```

```
# limitations under the License.
```

```
"""Module for storing and retrieving agent instructions.
```

```
This module defines functions that return instruction prompts for the
```

bqml\_agent.

These instructions guide the agent's behavior, workflow, and tool usage.

"""

```
def return_instructions_bqml() -> str:
```

```
    instruction_prompt_bqml_v2 = """
```

```
    <CONTEXT>
```

```
    <TASK>
```

```
        You are a BigQuery ML (BQML) expert agent. Your primary role is to
        assist users with BQML tasks, including model creation, training, and
        inspection. You also support data exploration using SQL.
```

```
        **Workflow:**
```

```
            1.  **Initial Information Retrieval:** ALWAYS start by using the
            `rag_response` tool to query the BQML Reference Guide. Use a precise query to
            retrieve relevant information. This information can help you answer user
            questions and guide your actions.
```

```
            2.  **Check for Existing Models:** If the user asks about existing
            BQML models, immediately use the `check_bq_models` tool. Use the `dataset_id`
            provided in the session context for this.
```

```
            3.  **BQML Code Generation and Execution:** If the user requests a
            task requiring BQML syntax (e.g., creating a model, training a model), follow
            these steps:
```

```
                a.  Query the BQML Reference Guide using the `rag_response`
            tool.
```

```
                b.  Generate the complete BQML code.
```

```
                c.  **CRITICAL:** Before executing, present the generated BQML
            code to the user for verification and approval.
```

```
                d.  Populate the BQML code with the correct `dataset_id` and
            `project_id` from the session context.
```

```
                e.  If the user approves, execute the BQML code using the
            `execute_bqml_code` tool. If the user requests changes, revise the code and
            repeat steps b-d.
```

f. **\*\*Inform the user:\*\*** Before executing the BQML code, inform the user that some BQML operations, especially model training, can take a significant amount of time to complete, potentially several minutes or even hours.

4. **\*\*Data Exploration:\*\*** If the user asks for data exploration or analysis, use the ``call_db_agent`` tool to execute SQL queries against BigQuery.

**\*\*Tool Usage:\*\***

\* ``rag_response``: Use this tool to get information from the BQML Reference Guide. Formulate your query carefully to get the most relevant results.

\* ``check_bq_models``: Use this tool to list existing BQML models in the specified dataset.

\* ``execute_bqml_code``: Use this tool to run BQML code. **\*\*Only use this tool AFTER the user has approved the code.\*\***

\* ``call_db_agent``: Use this tool to execute SQL queries for data exploration and analysis.

**\*\*IMPORTANT:\*\***

\* **\*\*User Verification is Mandatory:\*\*** NEVER use ``execute_bqml_code`` without explicit user approval of the generated BQML code.

\* **\*\*Context Awareness:\*\*** Always use the ``dataset_id`` and ``project_id`` provided in the session context. Do not hardcode these values.

\* **\*\*Efficiency:\*\*** Be mindful of token limits. Write efficient BQML code.

\* **\*\*No Parent Agent Routing:\*\*** Do not route back to the parent agent unless the user explicitly requests it.

\* **\*\*Prioritize `rag\_response`:\*\*** Always use ``rag_response`` first to gather information.

\* **\*\*Long Run Times:\*\*** Be aware that certain BQML operations, such as model training, can take a significant amount of time to complete. Inform the user about this possibility before executing such operations.

\* **\*\*No "process is running":** Never use the phrase "process is running" or similar, as your response indicates that the process has finished.

</TASK>

</CONTEXT>

"""

instruction\_prompt\_bqml\_v1 = """

<CONTEXT>

<TASK>

You are an agent that supports with BigQuery ML Workloads.

**\*\*Workflow\*\***

0. Always fetch information from the BQML Reference Guide first using the `rag\_response` tool. For this, make sure you are using a proper query to retrieve relevant information. (You can use this to answer questions, too)

1. If the user asks for existing models, call the `check\_bq\_models` tool. Use the dataset\_ID from the session context.

2. If the user asks for a task that needs BQ ML syntax:

2a. Generate the BQML and the code, populate the correct dataset ID and project ID from the session context. The user needs to validate and approve before you continue.

2b. If the user confirms, run the `execute\_bqml\_code` tool with the BQ ML you created, or correct your plan if necessary.

**\*\*Execute BQ Tool (`execute\_bqml\_code` - if applicable):\*\*** With the response from 2, properly formulate the returned BQ ML Code, add the dataset and project IDs stored in context, and run the execute\_bqml\_code tool.

**\*\*Check BQ ML Models Tool (`check\_bq\_models` - if applicable):\*\*** If the user asks for existing models in BQ ML, use this tool to check for it. Provide the dataset ID you have access to from the session context.

Below you will find documentation and examples of BigQuery ML.

3. If the user asks for data exploration, use the tool `call\_db\_agent`.

</TASK>

Do the following:



- You can use the `rag\_response` tool to retrieve information from the BQML Reference Guide.

- If the user asks for existing bqml models, run the `check\_bq\_models` tool.

- If the user asks for a task that needs BQ ML syntax, generate the BQML and return it for the user to verify. If verified, run the `execute\_bqml\_code` tool.

- If you need to execute SQL against BigQuery, e.g. for data understanding, use the tool `call\_db\_agent`.

- If the user asks for data exploration, use the tool `call\_db\_agent`.

**\*\*IMPORTANT:\*\***

\* Only run the execute\_bqml\_code tool once the user verified the code. NEVER USE `execute\_bqml\_code` BEFORE VERIFYING WITH THE USER!!

\* Make sure you use the database and project ID that is provided to you in the context!!

\* Be efficient. You have a output token limit, so make sure your BQML Code is efficient enough to stay in that limit.

\* Note: never route back to the parent agent, except when the user explicitly prompts for it.

</CONTEXT>

"""

instruction\_prompt\_bqml\_v0 = """

<TASK>

You are an agent that supports with BigQuery ML Workloads.

**\*\*Workflow\*\***

1. If the user asks for a existing models, call the `check\_bq\_models` tool.

2. If the user asks for a task that needs BQ ML syntax, generate the BQML, then **\*\*Execute BQ Tool (`execute\_bqml\_code` - if applicable):\*\*** With the

response from 2, properly formulate the returned BQ ML Code, add the dataset and project IDs stored in context, and run the `execute_bqml_code` tool.

**\*\*Check BQ ML Models Tool (``check_bq_models`` - if applicable):\*\*** If the user asks for existing models in BQ ML, use this tool to check for it. Provide the dataset ID you have access to from the session context.

Below you will find documentation and examples of BigQuery ML.

</TASK>

Do the following:

- If the user asks for existing bqml models, run the ``check_bq_models`` tool.

- If the user asks for a task that needs BQ ML syntax, generate the BQML and run the ``execute_bqml_code`` tool.

<EXAMPLE: CREATE LOGISTIC REGRESSION>

**\*\*BQ ML SYNTAX:\*\***

```
CREATE OR REPLACE MODEL `your_project_id.your_dataset_id.sample_model`  
  
OPTIONS(model_type='logistic_reg') AS  
  
SELECT  
  
IF(totals.transactions IS NULL, 0, 1) AS label,  
  
IFNULL(device.operatingSystem, "") AS os,  
  
device.isMobile AS is_mobile,  
  
IFNULL(geoNetwork.country, "") AS country,  
  
IFNULL(totals.pageviews, 0) AS pageviews  
  
FROM  
  
`your_project_id.your_dataset_id.ga_sessions_*`  
  
WHERE  
  
_TABLE_SUFFIX BETWEEN '20160801' AND '20170630'
```

## **\*\*QUERY DETAILS\*\***

The CREATE MODEL statement creates the model and then trains the model using the data retrieved by your query's SELECT statement.

The OPTIONS(model\_type='logistic\_reg') clause creates a logistic regression model. A logistic regression model splits input data into two classes, and then estimates the probability that the data is in one of the classes. What you are trying to detect, such as whether an email is spam, is represented by 1 and other values are represented by 0. The likelihood of a given value belonging to the class you are trying to detect is indicated by a value between 0 and 1. For example, if an email receives a probability estimate of 0.9, then there is a 90% probability that the email is spam.

This query's SELECT statement retrieves the following columns that are used by the model to predict the probability that a customer will complete a transaction:

totals.transactions: the total number of ecommerce transactions within the session. If the number of transactions is NULL, the value in the label column is set to 0. Otherwise, it is set to 1. These values represent the possible outcomes. Creating an alias named label is an alternative to setting the input\_label\_cols= option in the CREATE MODEL statement.

device.operatingSystem: the operating system of the visitor's device.

device.isMobile – Indicates whether the visitor's device is a mobile device.

geoNetwork.country: the country from which the sessions originated, based on the IP address.

totals.pageviews: the total number of page views within the session.

The FROM clause – causes the query to train the model by using the bigquery-public-data.google\_analytics\_sample.ga\_sessions sample tables. These tables are sharded by date, so you aggregate them by using a wildcard in the table name: google\_analytics\_sample.ga\_sessions\_\*.

The WHERE clause – \_TABLE\_SUFFIX BETWEEN '20160801' AND '20170630' – limits the number of tables scanned by the query. The date range scanned is August 1, 2016 to June 30, 2017.

</EXAMPLE: CREATE LOGISTIC REGRESSION>

<EXAMPLE: RETRIEVE TRAINING INFO>

SELECT

iteration,

loss,

eval\_metric

FROM

ML.TRAINING\_INFO(MODEL `my\_dataset.my\_model`)

ORDER BY

iteration;

</EXAMPLE: RETRIEVE TRAINING INFO>"""

return instruction\_prompt\_bqml\_v2

=====

File: data\_science/sub\_agents/bqml/tools.py

=====

# Copyright 2025 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");

# you may not use this file except in compliance with the License.

# You may obtain a copy of the License at

#

# <http://www.apache.org/licenses/LICENSE-2.0>

#

# Unless required by applicable law or agreed to in writing, software

```

# distributed under the License is distributed on an "AS IS" BASIS,

# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

# See the License for the specific language governing permissions and

# limitations under the License.


import time

import os

from google.cloud import bigquery

from vertexai import rag


def check_bq_models(dataset_id: str) -> str:

    """Lists models in a BigQuery dataset and returns them as a string.

    Args:

        dataset_id: The ID of the BigQuery dataset (e.g., "project.dataset").

    Returns:

        A string representation of a list of dictionaries, where each
        dictionary

            contains the 'name' and 'type' of a model in the specified dataset.

            Returns an empty string "[]" if no models are found.

    """

    try:

        client = bigquery.Client()

        models = client.list_models(dataset_id)

        model_list = [] # Initialize as a list

```

```

        print(f"Models contained in '{dataset_id}':")

        for model in models:

            model_id = model.model_id

            model_type = model.model_type

            model_list.append({"name": model_id, "type": model_type})

        return str(model_list)

    except Exception as e:

        return f"An error occurred: {str(e)}"

def execute_bqml_code(bqml_code: str, project_id: str, dataset_id: str) ->
str:

    """

    Executes BigQuery ML code.

    """

    # timeout_seconds = 1500

    client = bigquery.Client(project=project_id)

    try:

        query_job = client.query(bqml_code)

        start_time = time.time()

        while not query_job.done():

            elapsed_time = time.time() - start_time

```

```

        # if elapsed_time > timeout_seconds:

        #     return (

        #         "Timeout: BigQuery job did not complete within"

        #         f" {timeout_seconds} seconds. Job ID:
{query_job.job_id}"

        #     )

    print(

        f"Query Job Status: {query_job.state}, Elapsed Time:"

        f" {elapsed_time:.2f} seconds. Job ID: {query_job.job_id}"

    )

    time.sleep(5)

    if query_job.error_result:

        return f"Error executing BigQuery ML code:
{query_job.error_result}"

    if query_job.exception():

        return f"Exception during BigQuery ML execution:
{query_job.exception()}"

    results = query_job.result()

    if results.total_rows > 0:

        result_string = ""

        for row in results:

            result_string += str(dict(row.items())) + "\n"

        return f"BigQuery ML code executed successfully. Results:
\n{result_string}"

    else:

        return "BigQuery ML code executed successfully."

```

```

except Exception as e:

    return f"An error occurred: {str(e)}"


def rag_response(query: str) -> str:

    """Retrieves contextually relevant information from a RAG corpus.

    Args:

        query (str): The query string to search within the corpus.

    Returns:

        vertexai.rag.RagRetrievalQueryResponse: The response containing
retrieved

        information from the corpus.

    """

    corpus_name = os.getenv("BQML_RAG_CORPUS_NAME")

    rag_retrieval_config = rag.RagRetrievalConfig(

        top_k=3, # Optional

        filter=rag.Filter(vector_distance_threshold=0.5), # Optional

    )

    response = rag.retrieval_query(

        rag_resources=[

            rag.RagResource(

                rag_corpus=corpus_name,

            )

        ],

        text=query,

```



```
        rag_retrieval_config=rag_retrieval_config,
    )

    return str(response)
```

```
=====
```

```
File: data_science/utils/create_bq_table.py
```

```
=====
```

```
# Copyright 2025 Google LLC
```

```
#
```

```
# Licensed under the Apache License, Version 2.0 (the "License");
```

```
# you may not use this file except in compliance with the License.
```

```
# You may obtain a copy of the License at
```

```
#
```

```
#     http://www.apache.org/licenses/LICENSE-2.0
```

```
#
```

```
# Unless required by applicable law or agreed to in writing, software
```

```
# distributed under the License is distributed on an "AS IS" BASIS,
```

```
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
```

```
# See the License for the specific language governing permissions and
```

```
# limitations under the License.
```

```
import os
```

```
from google.cloud import bigquery
```

```
from pathlib import Path
```

```
from dotenv import load_dotenv
```

```

# Define the path to the .env file

env_file_path = Path(__file__).parent.parent.parent / ".env"

print(env_file_path)


# Load environment variables from the specified .env file

load_dotenv(dotenv_path=env_file_path)


def load_csv_to_bigquery(project_id, dataset_name, table_name, csv_filepath):

    """Loads a CSV file into a BigQuery table.

    Args:

        project_id: The ID of the Google Cloud project.

        dataset_name: The name of the BigQuery dataset.

        table_name: The name of the BigQuery table.

        csv_filepath: The path to the CSV file.

    """

    client = bigquery.Client(project=project_id)

    dataset_ref = client.dataset(dataset_name)

    table_ref = dataset_ref.table(table_name)

    job_config = bigquery.LoadJobConfig(

        source_format=bigquery.SourceFormat.CSV,

        skip_leading_rows=1, # Skip the header row

        autodetect=True, # Automatically detect the schema

    )

```

```

with open(csv_filepath, "rb") as source_file:

    job = client.load_table_from_file(

        source_file, table_ref, job_config=job_config

    )


job.result() # Wait for the job to complete


print(f"Loaded {job.output_rows} rows into {dataset_name}.{table_name}")


def create_dataset_if_not_exists(project_id, dataset_name):

    """Creates a BigQuery dataset if it does not already exist.

    Args:

        project_id: The ID of the Google Cloud project.

        dataset_name: The name of the BigQuery dataset.

    """

    client = bigquery.Client(project=project_id)

    dataset_id = f"{project_id}.{dataset_name}"

    try:

        client.get_dataset(dataset_id) # Make an API request.

        print(f"Dataset {dataset_id} already exists")

    except Exception:

        dataset = bigquery.Dataset(dataset_id)

        dataset.location = "US" # Set the location (e.g., "US", "EU")

        dataset = client.create_dataset(dataset, timeout=30) # Make an API
request.

```

```

        print(f"Created dataset {dataset_id}")

def main():

    current_directory = os.getcwd()

    print(f"Current working directory: {current_directory}")

    """Main function to load CSV files into BigQuery."""

    project_id = os.getenv("BQ_PROJECT_ID")

    if not project_id:

        raise ValueError("BQ_PROJECT_ID environment variable not set.")

    dataset_name = "forecasting_sticker_sales"

    train_csv_filepath = "data_science/utils/data/train.csv"

    test_csv_filepath = "data_science/utils/data/test.csv"

    # Create the dataset if it doesn't exist

    print("Creating dataset.")

    create_dataset_if_not_exists(project_id, dataset_name)

    # Load the train data

    print("Loading train table.")

    load_csv_to_bigquery(project_id, dataset_name, "train",
train_csv_filepath)

    # Load the test data

    print("Loading test table.")

    load_csv_to_bigquery(project_id, dataset_name, "test", test_csv_filepath)

```

```
if __name__ == "__main__":
```

```
    main()
```

```
=====
```

```
File: data_science/utils/reference_guide_RAG.py
```

```
=====
```

```
# Copyright 2025 Google LLC
```

```
#
```

```
# Licensed under the Apache License, Version 2.0 (the "License");
```

```
# you may not use this file except in compliance with the License.
```

```
# You may obtain a copy of the License at
```

```
#
```

```
# http://www.apache.org/licenses/LICENSE-2.0
```

```
#
```

```
# Unless required by applicable law or agreed to in writing, software
```

```
# distributed under the License is distributed on an "AS IS" BASIS,
```

```
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
```

```
# See the License for the specific language governing permissions and
```

```
# limitations under the License.
```

```
import os
```

```
from pathlib import Path
```

```
from dotenv import load_dotenv, set_key
```

```
import vertexai
```

```
from vertexai import rag
```

```

# Define the path to the .env file

env_file_path = Path(__file__).parent.parent.parent / ".env"

print(env_file_path)


# Load environment variables from the specified .env file

load_dotenv(dotenv_path=env_file_path)


PROJECT_ID = os.getenv("GOOGLE_CLOUD_PROJECT")

corpus_name = os.getenv("BQML_RAG_CORPUS_NAME")


display_name = "bqml_referenceguide_corpus"


paths = [

    "gs://cloud-samples-data/adk-samples/data-science/bqml"

] # Supports Google Cloud Storage and Google Drive Links


# Initialize Vertex AI API once per session

vertexai.init(project=PROJECT_ID, location="us-central1")


def create_RAG_corpus():

    # Create RagCorpus

    # Configure embedding model, for example "text-embedding-005".

    embedding_model_config = rag.RagEmbeddingModelConfig(

        vertex_prediction_endpoint=rag.VertexPredictionEndpoint(

```

```

        publisher_model="publishers/google/models/text-embedding-005"

    )

)

backend_config = rag.RagVectorDbConfig(

    rag_embedding_model_config=embedding_model_config

)

bqml_corpus = rag.create_corpus(

    display_name=display_name,

    backend_config=backend_config,

)

write_to_env(bqml_corpus.name)

return bqml_corpus.name

def ingest_files(corpus_name):

    transformation_config = rag.TransformationConfig(

        chunking_config=rag.ChunkingConfig(

            chunk_size=512,

            chunk_overlap=100,

        ),

    )

    rag.import_files(

        corpus_name,

```

```

        paths,

        transformation_config=transformation_config, # Optional

        max_embedding_requests_per_min=1000, # Optional

    )

    # List the files in the rag corpus

    rag.list_files(corpus_name)


def rag_response(query: str) -> str:

    """Retrieves contextually relevant information from a RAG corpus.

    Args:

        query (str): The query string to search within the corpus.

    Returns:

        vertexai.rag.RagRetrievalQueryResponse: The response containing
retrieved

        information from the corpus.

    """

    corpus_name = os.getenv("BQML_RAG_CORPUS_NAME")

    rag_retrieval_config = rag.RagRetrievalConfig(

        top_k=3, # Optional

        filter=rag.Filter(vector_distance_threshold=0.5), # Optional

    )

    response = rag.retrieval_query(

        rag_resources=[

            rag.RagResource(

```



```

        rag_corpus=corpus_name,

    )

],

    text=query,

    rag_retrieval_config=rag_retrieval_config,

)

return str(response)

```

```
def write_to_env(corpus_name):
```

```
    """Writes the corpus name to the specified .env file.
```

```
    Args:
```

```
        corpus_name: The name of the corpus to write.
```

```
    """
```

```
    load_dotenv(env_file_path) # Load existing variables if any
```

```
    # Set the key-value pair in the .env file
```

```
    set_key(env_file_path, "BQML_RAG_CORPUS_NAME", corpus_name)
```

```
    print(f"BQML_RAG_CORPUS_NAME '{corpus_name}' written to {env_file_path}")
```

```
if __name__ == "__main__":
```

```
    # rag_corpus = rag.list_corpora()
```

```
    corpus_name = os.getenv("BQML_RAG_CORPUS_NAME")
```

```
print("Creating the corpus.")

corpus_name = create_RAG_corpus()

print(f"Corpus name: {corpus_name}")


print(f"Importing files to corpus: {corpus_name}")

ingest_files(corpus_name)

print(f"Files imported to corpus: {corpus_name}")
```

=====

File: data\_science/utils/utils.py

=====

```
# Copyright 2025 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at

#

#     http://www.apache.org/licenses/LICENSE-2.0

#

# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


import json

import os
```

```

from vertexai.preview.extensions import Extension


def list_all_extensions():

    extensions = Extension.list(location='us-central1')

    for extension in extensions:

        print('Name:', extension.gca_resource.name)

        print('Display Name:', extension.gca_resource.display_name)

        print('Description:', extension.gca_resource.description)


def get_env_var(var_name):

    """Retrieves the value of an environment variable.

    Args:

        var_name: The name of the environment variable.

    Returns:

        The value of the environment variable, or None if it is not set.

    Raises:

        ValueError: If the environment variable is not set.

    """

    try:

        value = os.environ[var_name]

        return value

    except KeyError:

```

```
raise ValueError(f'Missing environment variable: {var_name}')
```

```
def get_image_bytes(filepath):
```

```
    """Reads an image file and returns its bytes.
```

```
    Args:
```

```
        filepath: The path to the image file.
```

```
    Returns:
```

```
        The bytes of the image file, or None if the file does not exist or cannot  
be
```

```
        read.
```

```
    """
```

```
    try:
```

```
        with open(filepath, 'rb') as f: # "rb" mode for reading in binary
```

```
            image_bytes = f.read()
```

```
        return image_bytes
```

```
    except FileNotFoundError:
```

```
        print(f'Error: File not found at {filepath}')
```

```
        return None
```

```
    except Exception as e:
```

```
        print(f'Error reading file: {e}')
```

```
        return None
```

```
def extract_json_from_model_output(model_output):
```

```
    """Extracts JSON object from a string that potentially contains markdown
```

code fences.

Args:

model\_output: A string potentially containing a JSON object wrapped in  
markdown code fences (````json ... ````).

Returns:

A Python dictionary representing the extracted JSON object,  
or None if JSON extraction fails.

"""

try:

```
cleaned_output = (  
    model_output.replace('````json', '').replace('````', '').strip()  
)
```

```
json_object = json.loads(cleaned_output)
```

```
return json_object
```

except json.JSONDecodeError as e:

```
msg = f'Error decoding JSON: {e}'
```

```
print(msg)
```

```
return {'error': msg}
```

```
if __name__ == '__main__':
```

```
    list_all_extensions()
```

=====

File: deployment/\_\_init\_\_.py

=====

# Copyright 2025 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");

# you may not use this file except in compliance with the License.

# You may obtain a copy of the License at

#

# <http://www.apache.org/licenses/LICENSE-2.0>

#

# Unless required by applicable law or agreed to in writing, software

# distributed under the License is distributed on an "AS IS" BASIS,

# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

# See the License for the specific language governing permissions and

# limitations under the License.

=====

File: deployment/deploy.py

=====

# Copyright 2025 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");

# you may not use this file except in compliance with the License.

# You may obtain a copy of the License at

#

```

#      http://www.apache.org/licenses/LICENSE-2.0

#

# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

""""Deployment script for Data Science agent."""

import logging

import os

import vertexai

from absl import app, flags

from data_science.agent import root_agent

from dotenv import load_dotenv

from google.api_core import exceptions as google_exceptions

from google.cloud import storage

from vertexai import agent_engines

from vertexai.preview.reasoning_engines import AdkApp

FLAGS = flags.FLAGS

flags.DEFINE_string("project_id", None, "GCP project ID.")

flags.DEFINE_string("location", None, "GCP location.")

flags.DEFINE_string(
    "bucket", None, "GCP bucket name (without gs:// prefix)."
)

# Changed flag description

flags.DEFINE_string("resource_id", None, "ReasoningEngine resource ID.")

```

```

flags.DEFINE_bool("create", False, "Create a new agent.")

flags.DEFINE_bool("delete", False, "Delete an existing agent.")

flags.mark_bool_flags_as_mutual_exclusive(["create", "delete"])


AGENT_WHL_FILE = "data_science-0.1-py3-none-any.whl"


# Configure logging

logging.basicConfig(level=logging.INFO)

logger = logging.getLogger(__name__)


def setup_staging_bucket(
    project_id: str, location: str, bucket_name: str
) -> str:
    """
    Checks if the staging bucket exists, creates it if not.

    Args:
        project_id: The GCP project ID.

        location: The GCP location for the bucket.

        bucket_name: The desired name for the bucket (without gs:// prefix).

    Returns:
        The full bucket path (gs://<bucket_name>).

    Raises:
        google_exceptions.GoogleCloudError: If bucket creation fails.

```



```

"""

storage_client = storage.Client(project=project_id)

try:

    # Check if the bucket exists

    bucket = storage_client.lookup_bucket(bucket_name)

    if bucket:

        logger.info("Staging bucket gs://%s already exists.", bucket_name)

    else:

        logger.info(

            "Staging bucket gs://%s not found. Creating...", bucket_name

        )

        # Create the bucket if it doesn't exist

        new_bucket = storage_client.create_bucket(

            bucket_name, project=project_id, location=location

        )

        logger.info(

            "Successfully created staging bucket gs://%s in %s.",

            new_bucket.name,

            location,

        )

        # Enable uniform bucket-level access for simplicity

        new_bucket.iam_configuration.uniform_bucket_level_access_enabled =

(

    True

)

        new_bucket.patch()

        logger.info(

            "Enabled uniform bucket-level access for gs://%s.",

            new_bucket.name,

```

```

    )

except google_exceptions.Forbidden as e:

    logger.error(

        (

            "Permission denied error for bucket gs://%s. "

            "Ensure the service account has 'Storage Admin' role. Error:
%s"

        ),

        bucket_name,

        e,

    )

    raise

except google_exceptions.Conflict as e:

    logger.warning(

        (

            "Bucket gs://%s likely already exists but owned by another "

            "project or recently deleted. Error: %s"

        ),

        bucket_name,

        e,

    )

    # Assuming we can proceed if it exists, even with a conflict warning

except google_exceptions.ClientError as e:

    logger.error(

        "Failed to create or access bucket gs://%s. Error: %s",

        bucket_name,

        e,

    )

```

```

        raise

    return f"gs://{bucket_name}"

def create(env_vars: dict[str, str]) -> None:

    """Creates and deploys the agent."""

    adk_app = AdkApp(

        agent=root_agent,

        enable_tracing=False,

    )

    if not os.path.exists(AGENT_WHL_FILE):

        logger.error("Agent wheel file not found at: %s", AGENT_WHL_FILE)

        # Consider adding instructions here on how to build the wheel file

        raise FileNotFoundError(f"Agent wheel file not found:
{AGENT_WHL_FILE}")

    logger.info("Using agent wheel file: %s", AGENT_WHL_FILE)

    remote_agent = agent_engines.create(

        adk_app,

        requirements=[AGENT_WHL_FILE],

        extra_packages=[AGENT_WHL_FILE],

        env_vars=env_vars

    )

    logger.info("Created remote agent: %s", remote_agent.resource_name)

    print(f"\nSuccessfully created agent: {remote_agent.resource_name}")

```

```

def delete(resource_id: str) -> None:

    """Deletes the specified agent."""

    logger.info("Attempting to delete agent: %s", resource_id)

    try:

        remote_agent = agent_engines.get(resource_id)

        remote_agent.delete(force=True)

        logger.info("Successfully deleted remote agent: %s", resource_id)

        print(f"\nSuccessfully deleted agent: {resource_id}")

    except google_exceptions.NotFound:

        logger.error("Agent with resource ID %s not found.", resource_id)

        print(f"\nAgent{resource_id} not found.")

        print(f"\nAgent not found: {resource_id}")

    except Exception as e:

        logger.error(

            "An error occurred while deleting agent %s: %s", resource_id, e

        )

        print(f"\nError deleting agent {resource_id}: {e}")


def main(argv: list[str]) -> None: # pylint: disable=unused-argument

    """Main execution function."""

    load_dotenv()

    env_vars = {}

    project_id = (

        FLAGS.project_id

        if FLAGS.project_id

        else os.getenv("GOOGLE_CLOUD_PROJECT")

```

```

    )

    location = (

        FLAGS.location if FLAGS.location else
os.getenv("GOOGLE_CLOUD_LOCATION")

    )

    # Default bucket name convention if not provided

    default_bucket_name = f"{project_id}-adk-staging" if project_id else None

    bucket_name = (

        FLAGS.bucket

        if FLAGS.bucket

        else os.getenv("GOOGLE_CLOUD_STORAGE_BUCKET", default_bucket_name)

    )

    # Don't set "GOOGLE_CLOUD_PROJECT" or "GOOGLE_CLOUD_LOCATION"

    # when deploying to Agent Engine. Those are set by the backend.

    env_vars["ROOT_AGENT_MODEL"] = os.getenv("ROOT_AGENT_MODEL")

    env_vars["ANALYTICS_AGENT_MODEL"] = os.getenv("ANALYTICS_AGENT_MODEL")

    env_vars["BASELINE_NL2SQL_MODEL"] = os.getenv("BASELINE_NL2SQL_MODEL")

    env_vars["BIGQUERY_AGENT_MODEL"] = os.getenv("BIGQUERY_AGENT_MODEL")

    env_vars["BQML_AGENT_MODEL"] = os.getenv("BQML_AGENT_MODEL")

    env_vars["CHASE_NL2SQL_MODEL"] = os.getenv("CHASE_NL2SQL_MODEL")

    env_vars["BQ_DATASET_ID"] = os.getenv("BQ_DATASET_ID")

    env_vars["BQ_PROJECT_ID"] = os.getenv("BQ_PROJECT_ID")

    env_vars["BQML_RAG_CORPUS_NAME"] = os.getenv("BQML_RAG_CORPUS_NAME")

    env_vars["CODE_INTERPRETER_EXTENSION_NAME"] = os.getenv(

        "CODE_INTERPRETER_EXTENSION_NAME")

    env_vars["NL2SQL_METHOD"] = os.getenv("NL2SQL_METHOD")

    logger.info("Using PROJECT: %s", project_id)

    logger.info("Using LOCATION: %s", location)

```

```
logger.info("Using BUCKET NAME: %s", bucket_name)

# --- Input Validation ---

if not project_id:

    print("\nError: Missing required GCP Project ID.")

    print(

        "Set the GOOGLE_CLOUD_PROJECT environment variable or use --
project_id flag."

    )

    return

if not location:

    print("\nError: Missing required GCP Location.")

    print(

        "Set the GOOGLE_CLOUD_LOCATION environment variable or use --
location flag."

    )

    return

if not bucket_name:

    print("\nError: Missing required GCS Bucket Name.")

    print(

        "Set the GOOGLE_CLOUD_STORAGE_BUCKET environment variable or use
--bucket flag."

    )

    return

if not FLAGS.create and not FLAGS.delete:

    print("\nError: You must specify either --create or --delete flag.")

    return

if FLAGS.delete and not FLAGS.resource_id:

    print(

        "\nError: --resource_id is required when using the --delete flag."
```

```

    )

    return

# --- End Input Validation ---

try:

    # Setup staging bucket

    staging_bucket_uri=None

    if FLAGS.create:

        staging_bucket_uri = setup_staging_bucket(

            project_id, location, bucket_name

        )

    # Initialize Vertex AI *after* bucket setup and validation

    vertexai.init(

        project=project_id,

        location=location,

        staging_bucket=staging_bucket_uri, # Staging bucket is passed
directly to create/update methods now

    )

    if FLAGS.create:

        create(env_vars)

    elif FLAGS.delete:

        delete(FLAGS.resource_id)

except google_exceptions.Forbidden as e:

    print(

        "Permission Error: Ensure the service account/user has necessary "

        "permissions (e.g., Storage Admin, Vertex AI User)."
```

```

        f"\nDetails: {e}"

    )

except FileNotFoundError as e:

    print(f"\nFile Error: {e}")

    print(

        "Please ensure the agent wheel file exists in the 'deployment' "

        "directory and you have run the build script "

        "(e.g., poetry build --format=wheel --output=deployment')."

    )

except Exception as e:

    print(f"\nAn unexpected error occurred: {e}")

    logger.exception(

        "Unhandled exception in main:"

    ) # Log the full traceback


if __name__ == "__main__":

    app.run(main)

```

```

=====

```

```

File: deployment/test_deployment.py

```

```

=====

```

```

# Copyright 2025 Google LLC

```

```

#

```

```

# Licensed under the Apache License, Version 2.0 (the "License");

```



```
# you may not use this file except in compliance with the License.

# You may obtain a copy of the License at

#

#     http://www.apache.org/licenses/LICENSE-2.0

#

# Unless required by applicable law or agreed to in writing, software

# distributed under the License is distributed on an "AS IS" BASIS,

# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

# See the License for the specific language governing permissions and

# limitations under the License.
```

```
"""Test deployment of Data Science Agent to Agent Engine."""
```

```
import asyncio
```

```
import os
```

```
import vertexai
```

```
from absl import app, flags
```

```
from dotenv import load_dotenv
```

```
from google.adk.sessions import VertexAiSessionService
```

```
from vertexai import agent_engines
```

```
FLAGS = flags.FLAGS
```

```
flags.DEFINE_string("project_id", None, "GCP project ID.")
```

```
flags.DEFINE_string("location", None, "GCP location.")
```

```
flags.DEFINE_string("bucket", None, "GCP bucket.")
```

```
flags.DEFINE_string(
```

```
    "resource_id",
```

```

        None,

        "ReasoningEngine resource ID (returned after deploying the agent)",
    )

    flags.DEFINE_string("user_id", None, "User ID (can be any string).")

    flags.mark_flag_as_required("resource_id")

    flags.mark_flag_as_required("user_id")


def main(argv: list[str]) -> None:  # pylint: disable=unused-argument

    load_dotenv()

    project_id = (

        FLAGS.project_id

        if FLAGS.project_id

        else os.getenv("GOOGLE_CLOUD_PROJECT")

    )

    location = (

        FLAGS.location if FLAGS.location else
os.getenv("GOOGLE_CLOUD_LOCATION")

    )

    bucket = (

        FLAGS.bucket

        if FLAGS.bucket

        else os.getenv("GOOGLE_CLOUD_STORAGE_BUCKET")

    )

    project_id = os.getenv("GOOGLE_CLOUD_PROJECT")

    location = os.getenv("GOOGLE_CLOUD_LOCATION")

```

```

bucket = os.getenv("GOOGLE_CLOUD_STORAGE_BUCKET")

if not project_id:

    print("Missing required environment variable: GOOGLE_CLOUD_PROJECT")

    return

elif not location:

    print("Missing required environment variable: GOOGLE_CLOUD_LOCATION")

    return

elif not bucket:

    print(

        "Missing required environment variable:
GOOGLE_CLOUD_STORAGE_BUCKET"

    )

    return


vertexai.init(

    project=project_id,

    location=location,

    staging_bucket=f"gs://{bucket}",

)


session_service = VertexAiSessionService(project_id, location)

session = asyncio.run(session_service.create_session(

    app_name=FLAGS.resource_id,

    user_id=FLAGS.user_id

))


agent = agent_engines.get(FLAGS.resource_id)

print(f"Found agent with resource ID: {FLAGS.resource_id}")

```

```

print(f"Created session for user ID: {FLAGS.user_id}")

print("Type 'quit' to exit.")

while True:

    user_input = input("Input: ")

    if user_input == "quit":

        break

    for event in agent.stream_query(

        user_id=FLAGS.user_id,

        session_id=session.id,

        message=user_input

    ):

        if "content" in event:

            if "parts" in event["content"]:

                parts = event["content"]["parts"]

                for part in parts:

                    if "text" in part:

                        text_part = part["text"]

                        print(f"Response: {text_part}")

    asyncio.run(session_service.delete_session(

        app_name=FLAGS.resource_id,

        user_id=FLAGS.user_id,

        session_id=session.id

    ))

print(f"Deleted session for user ID: {FLAGS.user_id}")

```

```
if __name__ == "__main__":  
    app.run(main)
```

=====

File: eval/\_\_init\_\_.py

=====

# Copyright 2025 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");

# you may not use this file except in compliance with the License.

# You may obtain a copy of the License at

#

# <http://www.apache.org/licenses/LICENSE-2.0>

#

# Unless required by applicable law or agreed to in writing, software

# distributed under the License is distributed on an "AS IS" BASIS,

# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

# See the License for the specific language governing permissions and

# limitations under the License.

=====

File: eval/test\_eval.py

=====

```
# Copyright 2025 Google LLC

#

# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


import os


import pytest

from dotenv import find_dotenv, load_dotenv

from google.adk.evaluation.agent_evaluator import AgentEvaluator


pytest_plugins = ("pytest_asyncio",)


@pytest.fixture(scope="session", autouse=True)

def load_env():

    load_dotenv(find_dotenv(".env"))
```

```

@pytest.mark.asyncio

async def test_eval_simple():

    """Test the agent's basic ability via a session file."""

    await AgentEvaluator.evaluate(

        "data_science",

        os.path.join(os.path.dirname(__file__), "eval_data/simple.test.json"),

        num_runs=1,

    )

```

```

=====

File: eval/eval_data/simple.test.json

=====

```

```

[

  {

    "query": "what data do you have?",

    "expected_tool_use": [],

    "expected_intermediate_agent_responses": [],

    "reference": "I have access to two tables: `train` and `test`. Both tables
contain information about sticker sales, including the date, country, store,
product, and number sold. The `train` table contains the historical sales
data, while the `test` table contains more recent sales data.\n"

  },

  {

    "query": "what countries are in test?",

    "expected_tool_use": [

      {

        "tool_name": "call_db_agent",

        "tool_input": {

```

```

        "question": "What are the distinct countries in the test table?"
    }

}

],

"expected_intermediate_agent_responses": [],

"reference": "***Result:** The distinct countries in the test table are
Canada, Finland, Italy, Kenya, Norway, and Singapore.\n\n**Explanation:** I
queried the `test` table to find all the distinct values in the `country`
column. The query returned the list of countries: Canada, Finland, Italy,
Kenya, Norway, and Singapore.\n"

}

]

```

```
=====
```

File: eval/eval\_data/test\_config.json

```
=====
```

```

{
    "criteria": {
        "tool_trajectory_avg_score": 1.0,
        "response_match_score": 0.1
    }
}

```

```
=====
```

File: tests/test\_agents.py

```
=====
```

# Copyright 2025 Google LLC



```
#

# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at

#
#      http://www.apache.org/licenses/LICENSE-2.0
#

# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""Test cases for the analytics agent and its sub-agents."""

import os

import sys

import pytest

import unittest

sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__),
"..")))

from google.genai import types

from google.adk.artifacts import InMemoryArtifactService

from google.adk.runners import Runner

from google.adk.sessions import InMemorySessionService

from data_science.agent import root_agent
```

```

from data_science.sub_agents.bqml.agent import root_agent as bqml_agent

from data_science.sub_agents.bigquery.agent import database_agent


session_service = InMemorySessionService()

artifact_service = InMemoryArtifactService()


class TestAgents(unittest.TestCase):

    """Test cases for the analytics agent and its sub-agents."""

    def setUp(self):

        """Set up for test methods."""

        self.session = session_service.create_session(

            app_name="DataAgent",

            user_id="test_user",

        )

        self.user_id = "test_user"

        self.session_id = self.session.id


        self.runner = Runner(

            app_name="DataAgent",

            agent=None,

            artifact_service=artifact_service,

            session_service=session_service,

        )


    def _run_agent(self, agent, query):

        """Helper method to run an agent and get the final response."""

```

```

self.runner.agent = agent

content = types.Content(role="user", parts=[types.Part(text=query)])

events = list(

    self.runner.run(

        user_id=self.user_id, session_id=self.session_id,
new_message=content

    )

)

last_event = events[-1]

final_response = "".join(

    [part.text for part in last_event.content.parts if part.text]

)

return final_response

```

```
@pytest.mark.db_agent
```

```

def test_db_agent_can_handle_env_query(self):

    """Test the db_agent with a query from environment variable."""

    query = "what countries exist in the train table?"

    response = self._run_agent(database_agent, query)

    print(response)

    # self.assertIn("Canada", response)

    self.assertIsNotNone(response)

```

```
@pytest.mark.ds_agent
```

```

def test_ds_agent_can_be_called_from_root(self):

    """Test the ds_agent from the root agent."""

    query = "plot the most selling category"

```

```

        response = self._run_agent(root_agent, query)

        print(response)

        self.assertIsNotNone(response)

@pytest.mark.bqml
def test_bqml_agent_can_check_for_models(self):

    """Test that the bqml_agent can check for existing models."""

    query = "Are there any existing models in the dataset?"

    response = self._run_agent(bqml_agent, query)

    print(response)

    self.assertIsNotNone(response)

@pytest.mark.bqml
def test_bqml_agent_can_execute_code(self):

    """Test that the bqml_agent can execute BQML code."""

    query = """

    I want to train a BigQuery ML model on the sales_train_validation data for
    sales prediction.

    Please show me an execution plan.

    """

    response = self._run_agent(bqml_agent, query)

    print(response)

    self.assertIsNotNone(response)

if __name__ == "__main__":

    unittest.main()

# testagent = TestAgents

```

```
# testagent.setUp(testagent)

# testagent.test_root_agent_can_list_tools(testagent)

# testagent.test_db_agent_can_handle_env_query(testagent)
```