# === Notebook setup & utilities ===

from google.colab import auth

import gspread

import yaml

import numpy as np

import pandas as pd

import plotly.graph\_objects as go

from sentence\_transformers import SentenceTransformer

import faiss

from langchain\_google\_genai import GoogleGenerativeAIEmbeddings, ChatGoogleGenerativeAI

from langgraph.graph import StateGraph, END

from langgraph.graph import START

from dataclasses import dataclass, asdict

import json

from datetime import datetime, timedelta

import re

import os

from typing import List, Dict, Any, Optional

from enum import Enum

from dateutil.relativedelta import relativedelta

from google.colab import files

import logging

# Authenticate with Google

auth.authenticate\_user()

from google.auth import default

creds, \_ = default()

gc = gspread.authorize(creds)

# Setup logging

logging.basicConfig(filename='analysis\_agent.log', level=logging.INFO,

format='%(asctime)s - %(levelname)s - %(message)s')

def load\_config(file\_path: str = 'config.yaml') -> dict:

"""Load the configuration from a YAML file."""

with open(file\_path, 'r') as file:

return yaml.safe\_load(file)

def load\_semantic\_layer(file\_path: str = 'semantic\_layer.yaml') -> dict:

"""Load the semantic-layer definition from a YAML file."""

with open(file\_path, 'r') as file:

return yaml.safe\_load(file)

class IntentType(Enum):

FETCH\_METRIC = "fetch\_metric"

COMPARE\_METRIC = "compare\_metric"

RANK\_ENTITIES = "rank\_entities"

DIAGNOSE\_METRIC = "diagnose\_metric"

TREND\_ANALYSIS = "trend\_analysis"

THRESHOLD\_CHECK = "threshold\_check"

LIST\_ENTITIES\_BY\_CRITERIA = "list\_entities\_by\_criteria"

SUMMARIZE\_METRIC = "summarize\_metric"

GET\_RECOMMENDATION = "get\_recommendation"

VISUALIZE\_METRIC = "visualize\_metric"

@dataclass

class AgentState:

user\_query: str

intent: Optional[List[str]] = None

extracted\_context: Optional[Dict] = None

similar\_contexts: Optional[List] = None

data: Optional[pd.DataFrame] = None

analysis\_results: Optional[Dict] = None

report: Optional[str] = None

timeframes: Optional[Dict] = None

errors: Optional[List[str]] = None

feedback: Optional[Dict] = None

# === VectorStore with Learning ===

class VectorStore:

def \_\_init\_\_(self, embeddings):

self.embeddings = embeddings

self.index = None

self.queries = []

self.query\_history = [] # For learning and follow-up suggestions

def add\_queries(self, queries: List[str]):

"""Embed and index a batch of example queries."""

self.queries.extend(queries)

query\_embeddings = self.embeddings.embed\_documents(queries)

if self.index is None:

dim = len(query\_embeddings[0])

self.index = faiss.IndexHNSWFlat(dim, 32)

self.index.add(np.array(query\_embeddings).astype('float32'))

else:

self.index.add(np.array(query\_embeddings).astype('float32'))

self.query\_history.extend(queries)

def add\_query(self, query: str):

"""Add a new query for learning."""

if query not in self.queries:

self.queries.append(query)

query\_embeddings = self.embeddings.embed\_documents([query])

self.index.add(np.array(query\_embeddings).astype('float32'))

self.query\_history.append(query)

logging.info(f"Learned new query: {query}")

def search(self, query: str, k: int = 3) -> List[Dict[str, Any]]:

"""Return the top-k most similar past queries."""

q\_emb = self.embeddings.embed\_query(query)

distances, indices = self.index.search(np.array([q\_emb]).astype('float32'), k)

results = []

for idx, dist in zip(indices[0], distances[0]):

if idx < len(self.queries):

results.append({

"query": self.queries[idx],

"score": float(dist)

})

return results

# === TimeExpressionHandler ===

class TimeExpressionHandler:

def \_\_init\_\_(self):

self.now = datetime.now() # Dynamic time for enterprise use

def parse\_time\_expression(self, expression: str, reference\_date: Optional[datetime] = None) -> Optional[Dict[str, str]]:

if not reference\_date:

reference\_date = self.now

lower\_expr = expression.lower().strip()

special\_cases = {

"this week": lambda ref: {"start": (ref - timedelta(days=ref.weekday())).strftime("%Y-%m-%d"), "end": (ref + timedelta(days=(6 - ref.weekday()))).strftime("%Y-%m-%d")},

"last week": lambda ref: {"start": (ref - timedelta(days=ref.weekday() + 7)).strftime("%Y-%m-%d"), "end": (ref - timedelta(days=ref.weekday() + 1)).strftime("%Y-%m-%d")},

"this month": lambda ref: {"start": ref.replace(day=1).strftime("%Y-%m-%d"), "end": ((ref.replace(day=28) + timedelta(days=4)).replace(day=1) - timedelta(days=1)).strftime("%Y-%m-%d")},

"last month": lambda ref: {"start": (ref.replace(day=1) - timedelta(days=1)).replace(day=1).strftime("%Y-%m-%d"), "end": (ref.replace(day=1) - timedelta(days=1)).strftime("%Y-%m-%d")},

"this year": lambda ref: {"start": ref.replace(month=1, day=1).strftime("%Y-%m-%d"), "end": ref.replace(month=12, day=31).strftime("%Y-%m-%d")},

"last year": lambda ref: {"start": ref.replace(year=ref.year - 1, month=1, day=1).strftime("%Y-%m-%d"), "end": ref.replace(year=ref.year - 1, month=12, day=31).strftime("%Y-%m-%d")},

"this quarter": lambda ref: {"start": ref.replace(month=((ref.month - 1) // 3 \* 3 + 1), day=1).strftime("%Y-%m-%d"), "end": (ref.replace(month=((ref.month - 1) // 3 \* 3 + 4), day=1) - timedelta(days=1)).strftime("%Y-%m-%d")},

"last quarter": lambda ref: {"start": (ref.replace(month=((ref.month - 1) // 3 \* 3 + 1), day=1) - timedelta(days=92)).replace(day=1).strftime("%Y-%m-%d"), "end": (ref.replace(month=((ref.month - 1) // 3 \* 3 + 1), day=1) - timedelta(days=1)).strftime("%Y-%m-%d")},

"till now": lambda ref: {"start": "1970-01-01", "end": ref.strftime("%Y-%m-%d")},

"as of yesterday": lambda ref: {"start": "1970-01-01", "end": (ref - timedelta(days=1)).strftime("%Y-%m-%d")}

}

for expr, fn in special\_cases.items():

if expr in lower\_expr:

return fn(reference\_date)

range\_match = re.search(r'(?:last|past)\s+(\d+)\s+(month|months|year|years)', lower\_expr)

if range\_match:

num, unit = int(range\_match.group(1)), range\_match.group(2)

if unit.startswith("month"):

start = (reference\_date.replace(day=1) - relativedelta(months=num)).replace(day=1)

end = ((reference\_date.replace(day=28) + timedelta(days=4)).replace(day=1) - timedelta(days=1))

else:

start = reference\_date.replace(year=reference\_date.year - num, month=1, day=1)

end = reference\_date.replace(month=12, day=31)

return {"start": start.strftime("%Y-%m-%d"), "end": end.strftime("%Y-%m-%d")}

month\_match = re.search(r'(january|jan|february|feb|march|mar|april|apr|may|june|jun|july|jul|august|aug|september|sep|october|oct|november|nov|december|dec)\s+(\d{4})', lower\_expr)

if month\_match:

month\_name, yr = month\_match.group(1), int(month\_match.group(2))

month\_map = {"jan":1, "january":1, "feb":2, "february":2, "mar":3, "march":3, "apr":4, "april":4, "may":5, "jun":6, "june":6, "jul":7, "july":7, "aug":8, "august":8, "sep":9, "september":9, "oct":10, "october":10, "nov":11, "november":11, "dec":12, "december":12}

mnum = month\_map[month\_name]

start = datetime(yr, mnum, 1)

end = (start.replace(month=mnum % 12 + 1, day=1) - timedelta(days=1))

return {"start": start.strftime("%Y-%m-%d"), "end": end.strftime("%Y-%m-%d")}

logging.warning(f"No match found for time expression: {lower\_expr}")

return {"start": None, "end": None}

# === FeedbackCollector ===

class FeedbackCollector:

def \_\_init\_\_(self, feedback\_file: str = 'feedback.json'):

self.feedback\_file = feedback\_file

self.feedback\_data = self.\_load\_feedback()

def \_load\_feedback(self) -> Dict:

try:

with open(self.feedback\_file, 'r') as f:

return json.load(f)

except FileNotFoundError:

return {"queries": [], "feedback": []}

def save\_feedback(self):

with open(self.feedback\_file, 'w') as f:

json.dump(self.feedback\_data, f, indent=2)

def add\_feedback(self, query: str, feedback: str):

self.feedback\_data["queries"].append(query)

self.feedback\_data["feedback"].append(feedback)

self.save\_feedback()

logging.info(f"Added feedback for query: {query}, Feedback: {feedback}")

def get\_feedback\_insights(self) -> Dict:

if not self.feedback\_data["feedback"]:

return {}

return {"common\_issues": max(set(self.feedback\_data["feedback"]), key=self.feedback\_data["feedback"].count)}

# === Query Understanding Agent ===

class QueryUnderstandingAgent:

def \_\_init\_\_(self, gsheet\_url: str, semantic\_layer\_path: str):

self.sheet = gc.open\_by\_url(gsheet\_url)

self.llm = ChatGoogleGenerativeAI(model="gemini-1.5-flash", temperature=0.2)

self.embeddings = GoogleGenerativeAIEmbeddings(model="models/embedding-001")

self.vector\_store = VectorStore(self.embeddings)

self.semantic\_layer = load\_semantic\_layer(semantic\_layer\_path)

self.time\_handler = TimeExpressionHandler()

self.feedback\_collector = FeedbackCollector()

example\_queries = [

"What is the efficiency for JP in the week of 6th Jan 25?",

"Compare the efficiency for VID 12345 in April and May 2025",

"which VIDs have efficiency below 60% this month?",

"why is the efficiency low for VID 12345?",

"Show the trend of efficiency for JP over the past year",

"Is efficiency below 70% for any VID this month?",

"Summarize the efficiency for all VIDs in May 2025",

"How can we improve efficiency for VID 12345?"

]

self.vector\_store.add\_queries(example\_queries)

def extract\_intent(self, query: str) -> List[str]:

query\_lower = query.lower()

intents = []

if any(keyword in query\_lower for keyword in ["what is"]):

intents.append(IntentType.FETCH\_METRIC.value)

if any(keyword in query\_lower for keyword in ["top", "bottom", "lowest", "highest", "best", "worst", "what are"]):

intents.append(IntentType.RANK\_ENTITIES.value)

if any(keyword in query\_lower for keyword in ["why", "reason", "cause", "explain", "how", "work", "diagnose", "root cause"]):

intents.append(IntentType.DIAGNOSE\_METRIC.value)

if any(keyword in query\_lower for keyword in ["compare", "versus", "vs", "comparison", "compared"]):

intents.append(IntentType.COMPARE\_METRIC.value)

if any(keyword in query\_lower for keyword in ["trend", "over time", "history", "previous", "last", "quarterly", "half yearly", "yearly"]):

intents.append(IntentType.TREND\_ANALYSIS.value)

if any(keyword in query\_lower for keyword in ["below", "above", "threshold", "check"]): # Updated with "check" from feedback

intents.append(IntentType.THRESHOLD\_CHECK.value)

if any(keyword in query\_lower for keyword in ["summarize", "summarise", "summary", "conclusion", "finally", "conclude", "average", "overall"]):

intents.append(IntentType.SUMMARIZE\_METRIC.value)

if any(keyword in query\_lower for keyword in ["recommend", "improve", "predict", "how to", "optimize", "optimise"]):

intents.append(IntentType.GET\_RECOMMENDATION.value)

if any(keyword in query\_lower for keyword in ["plot", "draw", "visualise", "visualize", "show"]):

intents.append(IntentType.VISUALIZE\_METRIC.value)

if intents:

return intents

prompt = f"""

Analyze the following user query and determine all possible intents from these options using chain-of-thought reasoning. Break down your thought process step by step, then return a JSON array of intent names.

Options:

1. fetch\_metric: Retrieve a specific metric value

2. compare\_metric: Compare metric values

3. rank\_entities: Rank entities by a metric

4. diagnose\_metric: Diagnose or explain a metric's value

5. trend\_analysis: Analyze metric trends over time

6. threshold\_check: Check if a metric meets a threshold

7. list\_entities\_by\_criteria: List entities meeting criteria

8. summarize\_metric: Provide summary statistics

9. get\_recommendation: Suggest improvements

10. visualize\_metric: Generate visualizations and plots

Query: "{query}"

Thought Process: [Your step-by-step reasoning here]

Intents:

"""

response = self.llm.invoke(prompt)

try:

content = response.content.strip()

intents\_start = content.find("Intents:") + len("Intents:")

intents\_json = content[intents\_start:].strip()

if intents\_json.startswith("[") and intents\_json.endswith("]"):

intents = json.loads(intents\_json)

return [IntentType(intents\_str.lower()).value for intents\_str in intents if intents\_str.lower() in IntentType.\_value2member\_map\_]

return [IntentType.FETCH\_METRIC.value]

except json.JSONDecodeError:

logging.error(f"JSON decode error for intent extraction: {response.content}")

return [IntentType.FETCH\_METRIC.value]

except ValueError:

logging.error(f"Value error for intent extraction: {response.content}")

return [IntentType.FETCH\_METRIC.value]

def extract\_entities(self, query: str) -> Dict[str, Any]:

columns = self.semantic\_layer["tables"]["efficiency\_table"]["columns"]

metrics = list(self.semantic\_layer["metrics"].keys())

semantic\_map = {}

for col, info in columns.items():

for term in info.get("semantic", []):

semantic\_map[term.lower()] = col

semantic\_map[col.lower()] = col

for metric in metrics:

semantic\_map[metric.lower()] = metric

prompt = f"""

Analyze the query and extract:

1. The metric (e.g., efficiency, c1\_km\_perc, SOH)

2. Entities (e.g., country, region, VID) and their filter values (e.g., 'JP', 'NA', 12345)

3. Entity type (country\_code, region, vehicle\_id, other)

4. Date field is drive\_date in the data. If any time-related term appears, map entity\_in\_data to drive\_date.

5. If a country name is detected, map entity\_in\_data to country\_code with a two-letter code (e.g., 'Japan' -> 'JP').

6. Map region names: Asia Pacific -> APAC, Europe Africa -> EMEA, Latin America -> LATAM, North America -> NA.

7. For queries like 'all countries in a region', map entity\_in\_data to country\_code and filter\_value to all relevant country codes.

Return as JSON: {{'metrics': [<metric1>, <metric2>, ...], 'entities': [{{'entity\_type': <type>, 'query\_term': <term>, 'data\_field': <column>, 'value': <value>, 'value\_type': <type>}}, ...]}}

Metrics: {', '.join(metrics)}

Dimensions: {', '.join(k for k, v in columns.items() if v.get("dimension", False))}

Query: "{query}"

"""

response = self.llm.invoke(prompt)

cleaned\_response = response.content.strip()

if cleaned\_response.startswith("```json"):

cleaned\_response = cleaned\_response[7:-3].strip()

elif cleaned\_response.startswith("```"):

cleaned\_response = cleaned\_response[3:-3].strip()

try:

result = json.loads(cleaned\_response)

if "metrics" not in result:

result["metrics"] = ["efficiency"]

if "entities" not in result:

result["entities"] = []

return result

except json.JSONDecodeError:

logging.error(f"JSON decode error for entity extraction: {response.content}")

return {"metrics": ["efficiency"], "entities": []}

def extract\_timeframe(self, query: str) -> List[Dict[str, Any]]:

sentences = re.split(r'[.?]', query)

timeframes = []

time\_exprs = ["this week", "last week", "this month", "last month", "this year", "last year", "this quarter", "last quarter", "last 6 months", "last 2 months", "till now", "as of yesterday"]

reference\_date = self.time\_handler.now

for sentence in sentences:

if not sentence.strip():

continue

intents = self.extract\_intent(sentence)

month\_matches = list(re.finditer(r"(january|jan|february|feb|march|mar|april|apr|may|june|jun|july|jul|august|aug|september|sep|october|oct|november|nov|december|dec)\s+(\d{4})", sentence))

for month\_match in month\_matches:

time\_expr = month\_match.group(0)

parsed\_time = self.time\_handler.parse\_time\_expression(time\_expr, reference\_date)

if parsed\_time["start"] and parsed\_time["end"]:

timeframes.append({"timeframe\_in\_query": time\_expr, "timeframe\_in\_data": parsed\_time, "intent": intents})

range\_matches = list(re.finditer(r"(?:last|past)\s+(\d+)\s+(month|months|year|years)", sentence.lower()))

for range\_match in range\_matches:

time\_expr = range\_match.group(0)

parsed\_time = self.time\_handler.parse\_time\_expression(time\_expr, reference\_date)

if parsed\_time["start"] and parsed\_time["end"]:

timeframes.append({"timeframe\_in\_query": time\_expr, "timeframe\_in\_data": parsed\_time, "intent": intents})

for expr in time\_exprs:

if expr in sentence.lower():

parsed\_time = self.time\_handler.parse\_time\_expression(expr, reference\_date)

if parsed\_time["start"] and parsed\_time["end"]:

timeframes.append({"timeframe\_in\_query": expr, "timeframe\_in\_data": parsed\_time, "intent": intents})

if not timeframes:

return [{"timeframe\_in\_query": None, "timeframe\_in\_data": {"start": None, "end": None}, "intent": intents}]

return timeframes

def process(self, query: str) -> Dict[str, Any]:

intents = self.extract\_intent(query)

entities\_metrics = self.extract\_entities(query)

timeframes = self.extract\_timeframe(query)

similar\_contexts = self.vector\_store.search(query)

feedback\_insights = self.feedback\_collector.get\_feedback\_insights()

if feedback\_insights:

for issue in feedback\_insights.get("common\_issues", []):

if "intent misclassification" in issue.lower():

intents = self.\_refine\_intents\_with\_feedback(intents, query)

extracted\_context = {

"entity\_in\_query": [info["query\_term"] for info in entities\_metrics["entities"]] if entities\_metrics.get("entities") else ["countries"],

"entity\_in\_data": [info["data\_field"] for info in entities\_metrics["entities"]] if entities\_metrics.get("entities") else ["country\_code"],

"filter\_value": [info["value"] for info in entities\_metrics["entities"]] if entities\_metrics.get("entities") else ["NA"],

"metrics": {intent: entities\_metrics["metrics"][0] if entities\_metrics.get("metrics") else "efficiency" for intent in intents},

"timeframes": {t["timeframe\_in\_query"]: {"start": t["timeframe\_in\_data"]["start"], "end": t["timeframe\_in\_data"]["end"], "intent": t["intent"]} for t in timeframes}

}

output = {

"intent": intents,

"user\_query": query,

"extracted\_context": extracted\_context,

"similar\_contexts": [{"intent": self.extract\_intent(q["query"]), "query": q["query"]} for q in similar\_contexts]

}

self.vector\_store.add\_query(query) # Query pattern learning

return output

def \_refine\_intents\_with\_feedback(self, intents: List[str], query: str) -> List[str]:

feedback\_insights = self.feedback\_collector.get\_feedback\_insights()

if feedback\_insights.get("common\_issues"):

prompt = f"""

Refine intents based on feedback. Current intents: {intents}. Query: {query}. Common feedback issues: {feedback\_insights['common\_issues']}. Suggest corrected intents using chain-of-thought reasoning, then return a JSON array.

Thought Process: [Your reasoning here]

Intents:

"""

response = self.llm.invoke(prompt)

try:

content = response.content.strip()

intents\_start = content.find("Intents:") + len("Intents:")

intents\_json = content[intents\_start:].strip()

if intents\_json.startswith("[") and intents\_json.endswith("]"):

refined\_intents = json.loads(intents\_json)

return [IntentType(intent.lower()).value for intent in refined\_intents if intent.lower() in IntentType.\_value2member\_map\_]

except json.JSONDecodeError:

logging.error(f"JSON decode error in intent refinement: {response.content}")

return intents

# === Data Retrieval Agent ===

class DataRetrievalAgent:

def \_\_init\_\_(self, gsheet\_url: str):

self.sheet = gc.open\_by\_url(gsheet\_url)

self.worksheets = {sheet.title: sheet for sheet in self.sheet.worksheets()} # Multi-sheet support

def fetch\_data(self, extracted\_context: Dict[str, Any]) -> pd.DataFrame:

worksheet = self.worksheets.get("efficiency\_data", self.sheet.get\_worksheet(0)) # Default to first sheet

data = worksheet.get\_all\_records()

df = pd.DataFrame(data)

logging.info(f"Initial DataFrame shape: {df.shape}")

if "drive\_date" not in df.columns:

logging.error("drive\_date column missing in data")

return pd.DataFrame()

df["drive\_date"] = pd.to\_datetime(df["drive\_date"], errors='coerce')

start\_dates = [pd.to\_datetime(t["start"]) for t in extracted\_context["timeframes"].values() if t["start"]]

end\_dates = [pd.to\_datetime(t["end"]) for t in extracted\_context["timeframes"].values() if t["end"]]

start\_date = min(start\_dates) if start\_dates else None

end\_date = max(end\_dates) if end\_dates else None

if start\_date and end\_date:

df = df[(df["drive\_date"] >= start\_date) & (df["drive\_date"] <= end\_date)]

entity\_filters = []

for i in range(len(extracted\_context["entity\_in\_data"])):

entity\_in\_data = extracted\_context["entity\_in\_data"][i]

filter\_value = extracted\_context["filter\_value"][i]

if entity\_in\_data in df.columns and filter\_value:

if entity\_in\_data == "country\_code" or entity\_in\_data == "region":

entity\_filters.append(f"{entity\_in\_data} == @filter\_value")

else:

entity\_filters.append(f"{entity\_in\_data} == '{filter\_value}'")

if entity\_filters:

try:

df = df.query(" and ".join(entity\_filters))

except Exception as e:

logging.error(f"Filter query failed: {str(e)}")

return pd.DataFrame()

if not df.empty:

df["intent"] = df["drive\_date"].apply(

lambda x: list({

intent

for timeframe in extracted\_context["timeframes"].values()

for intent in timeframe["intent"]

if pd.to\_datetime(timeframe["start"], errors='coerce') <= x <= pd.to\_datetime(timeframe["end"], errors='coerce')

})

)

return df

# === Analysis Agents ===

class Agent:

def process(self, df: pd.DataFrame, intent: List[str], context: Dict = None) -> Dict[str, Any]:

raise NotImplementedError("Subclasses must implement process()")

class MetricCalculator(Agent):

def \_\_init\_\_(self, semantic\_layer):

self.metrics = semantic\_layer.get("metrics", {})

def process(self, df: pd.DataFrame, intent: List[str], context: Dict = None) -> Dict[str, Any]:

df["drive\_date"] = pd.to\_datetime(df["drive\_date"], errors='coerce')

df["drive\_month"] = pd.to\_datetime(df["drive\_month"], errors='coerce', unit='M', origin='1970-01-01')

for metric, config in self.metrics.items():

formula = config.get("formula")

columns = config.get("columns", [])

if all(col in df.columns for col in columns):

try:

if "sum" in formula.lower():

numerator = df[columns[0]].sum() if len(columns) > 0 else 0

denominator = df[columns[1]].sum() if len(columns) > 1 else 1

df[metric] = numerator / denominator if denominator else 0

elif "mean" in formula.lower():

df[metric] = df[columns[0]].mean()

elif "min" in formula.lower():

df[metric] = df[columns[0]].min()

elif "max" in formula.lower():

df[metric] = df[columns[0]].max()

except Exception as e:

logging.error(f"Metric calculation failed for {metric}: {str(e)}")

df[metric] = np.nan

return {"data": df}

class InsightGenerator(Agent):

def process(self, df: pd.DataFrame, intent: List[str], context: Dict = None) -> Dict[str, Any]:

metrics = context.get("metrics", {}) if context else {}

try:

for row\_intent in df["intent"].iloc[0] if not df.empty else []:

if row\_intent in intent:

metric = metrics.get(row\_intent, next(iter(metrics.values()), "efficiency"))

if metric in df.columns:

if row\_intent == "fetch\_metric":

return {"summary": f"Value of {metric}: {df[metric].mean():.2f}"}

elif row\_intent == "compare\_metric" and "drive\_date" in df.columns:

comparison = df.groupby(df["drive\_date"].dt.to\_period("M"))[metric].mean().to\_dict()

return {"summary": f"Comparison of {metric}", "data": comparison}

elif row\_intent == "rank\_entities":

ranking = df.groupby("vehicle\_id")[metric].mean().sort\_values(ascending=False).head(5).to\_dict()

return {"summary": f"Top 5 by {metric}", "data": ranking}

elif row\_intent == "threshold\_check":

below\_threshold = df[df[metric] < 0.7]["vehicle\_id"].unique().tolist()

return {"summary": f"Vehicle IDs with {metric} < 70%: {below\_threshold}"}

elif row\_intent == "list\_entities\_by\_criteria":

criteria\_met = df[df[metric] < 0.7]["vehicle\_id"].unique().tolist()

return {"summary": f"Entities with {metric} < 0.7: {criteria\_met}"}

elif row\_intent == "summarize\_metric":

summary = {f"{metric}\_{stat}": df[metric].agg(stat) for stat in ["mean", "min", "max"]}

return {"summary": f"Summary of {metric}", "data": summary}

elif row\_intent == "trend\_analysis" and "drive\_date" in df.columns:

trend = df.groupby(df["drive\_date"].dt.to\_period("M"))[metric].mean().to\_dict()

return {"summary": f"Trend for {metric}", "data": trend}

except Exception as e:

logging.error(f"Insight generation failed: {str(e)}")

return {"error": str(e)}

return {}

class PatternDetector(Agent):

def process(self, df: pd.DataFrame, intent: List[str], context: Dict = None) -> Dict[str, Any]:

metrics = context.get("metrics", {}) if context else {}

llm = ChatGoogleGenerativeAI(model="gemini-pro", temperature=0.1)

try:

for row\_intent in df["intent"].iloc[0] if not df.empty else []:

if row\_intent in intent and row\_intent == "diagnose\_metric":

metric = metrics.get(row\_intent, next(iter(metrics.values()), "efficiency"))

if metric in df.columns:

correlations = {col: df[col].corr(df[metric]) for col in ["C1\_Kms\_Collected", "C2\_Kms\_Collected", "C3\_Kms\_Collected"] if col in df.columns}

prompt = f"""

Perform chain-of-thought reasoning to diagnose why {metric} is low. Data correlations: {correlations}. Step-by-step:

1. Identify the strongest negative correlation.

2. Hypothesize a causal relationship.

3. Suggest a verification step.

Thought Process: [Your reasoning here]

Diagnosis:

"""

response = llm.invoke(prompt)

diagnosis = response.content.strip().split("Diagnosis:")[1].strip() if "Diagnosis:" in response.content else "Unable to determine cause."

return {"hidden\_insights": [diagnosis]}

except Exception as e:

logging.error(f"Pattern detection failed: {str(e)}")

return {"error": str(e)}

return {}

class PlotGenerator(Agent):

def process(self, df: pd.DataFrame, intent: List[str], context: Dict = None) -> Dict[str, Any]:

metrics = context.get("metrics", {}) if context else {}

try:

for row\_intent in df["intent"].iloc[0] if not df.empty else []:

if row\_intent in intent and row\_intent == "visualize\_metric" and "drive\_date" in df.columns:

metric = metrics.get(row\_intent, next(iter(metrics.values()), "efficiency"))

if metric in df.columns:

fig = go.Figure()

fig.add\_trace(go.Scatter(x=df["drive\_date"], y=df[metric], mode="lines"))

fig.update\_layout(title=f"{metric} Trend", xaxis\_title="Date", yaxis\_title=metric)

fig.show()

return {"plot": f"Interactive {metric} plot displayed in canvas"}

except Exception as e:

logging.error(f"Plot generation failed: {str(e)}")

return {"error": str(e)}

return {}

class RecommendationEngine(Agent):

def process(self, df: pd.DataFrame, intent: List[str], context: Dict = None) -> Dict[str, Any]:

metrics = context.get("metrics", {}) if context else {}

try:

for row\_intent in df["intent"].iloc[0] if not df.empty else []:

if row\_intent in intent and row\_intent == "get\_recommendation":

metric = metrics.get(row\_intent, next(iter(metrics.values()), "efficiency"))

if metric in df.columns and df[metric].min() < 0.7:

return {"recommendations": [f"Optimize {metric} for low-performing VIDs by reviewing TKM and Total\_KM.", f"Consider maintenance checks for vehicles with {metric} < 0.7."]}

except Exception as e:

logging.error(f"Recommendation failed: {str(e)}")

return {"error": str(e)}

return {}

# === Analysis Workflow ===

def validate\_intents(state: AgentState) -> Dict[str, Any]:

if state.data is not None and not state.data.empty:

df\_intents = set().union(\*state.data["intent"])

state\_intents = set(state.intent or [])

if df\_intents != state\_intents:

mismatch = list(state\_intents - df\_intents)

return {"errors": [f"Intent mismatch: {mismatch} not found in DataFrame intents {df\_intents}"]}

return {}

def metric\_calculation\_node(state: AgentState) -> Dict[str, Any]:

return {

"data": MetricCalculator(load\_semantic\_layer()).process(state.data, state.intent, state.extracted\_context)["data"],

\*\*validate\_intents(state)

}

def create\_analysis\_nodes(config: dict, llm: ChatGoogleGenerativeAI):

graph = StateGraph(AgentState)

graph.add\_node("metric\_calculation", metric\_calculation\_node)

agents = {

"MetricCalculator": MetricCalculator(load\_semantic\_layer()),

"InsightGenerator": InsightGenerator(),

"PatternDetector": PatternDetector(),

"PlotGenerator": PlotGenerator(),

"RecommendationEngine": RecommendationEngine()

}

def error\_handling\_node(state: AgentState) -> Dict[str, Any]:

if state.errors:

llm\_prompt = f"Handle errors: {json.dumps(state.errors)}. Suggest fixes or fallback actions based on user\_query: {state.user\_query} and intents: {state.intent}. Return JSON with 'fix' and 'message' fields."

response = llm.invoke(llm\_prompt)

try:

fix\_data = json.loads(response.content.strip())

state.feedback = {"query": state.user\_query, "feedback": f"Applied fix: {fix\_data['fix']}"}

state.feedback\_collector.add\_feedback(state.user\_query, fix\_data['message'])

state.report = f"Errors occurred: {state.errors}\nLLM Fix: {fix\_data['message']}\n{state.report or ''}"

state.errors = []

except json.JSONDecodeError:

logging.error(f"JSON decode error in error handling: {response.content}")

state.report = f"Errors occurred: {state.errors}\n{state.report or ''}"

state.errors = []

return state

for agent\_name in agents:

graph.add\_node(agent\_name.lower(), lambda state, agent=agents[agent\_name]: {

\*\*agents[agent\_name].process(state.data, state.intent, state.extracted\_context),

"data": state.data

})

agent\_mapping = config.get("agent\_mapping", {})

for intent in IntentType.\_\_dict\_\_.values():

if intent and isinstance(intent, str):

next\_nodes = agent\_mapping.get(intent, [])

if next\_nodes:

for node in next\_nodes:

graph.add\_conditional\_edges("metric\_calculation", lambda x, intent=intent: node.lower() if intent in x.intent else None, [node.lower()])

graph.add\_edge(node.lower(), "error\_handling")

graph.add\_node("error\_handling", error\_handling\_node)

graph.add\_conditional\_edges("error\_handling", lambda state: "report\_generation" if not state.errors else END, ["report\_generation"])

graph.add\_node("report\_generation", lambda state: {

"report": llm.invoke(f"""

Compile a professional report with: {json.dumps(state.analysis\_results or {})}. Enhance with similar\_contexts: {json.dumps(state.similar\_contexts or [])} and query history: {json.dumps(state.vector\_store.query\_history[-5:] if state.vector\_store.query\_history else [])}.

Suggest follow-up questions based on the analysis, similar contexts, and history using chain-of-thought reasoning. Include the thought process.

Thought Process: [Your reasoning here]

Report:

Follow-up Questions:

""").content.strip().split("Follow-up Questions:")[1].strip() if "Follow-up Questions:" in llm.invoke(f"""

Compile a professional report with: {json.dumps(state.analysis\_results or {})}. Enhance with similar\_contexts: {json.dumps(state.similar\_contexts or [])} and query history: {json.dumps(state.vector\_store.query\_history[-5:] if state.vector\_store.query\_history else [])}.

Suggest follow-up questions based on the analysis, similar contexts, and history using chain-of-thought reasoning. Include the thought process.

Thought Process: [Your reasoning here]

Report:

Follow-up Questions:

""").content else "Report compilation failed." + f"\nFollow-up Questions: Check trends for other regions?"

})

graph.add\_edge("report\_generation", END)

graph.set\_entry\_point("metric\_calculation")

return graph.compile()

# === Full Workflow ===

def query\_understanding\_node(state: AgentState) -> Dict[str, Any]:

query\_agent = QueryUnderstandingAgent(

gsheet\_url="https://docs.google.com/spreadsheets/d/1auyXukmp19gTl0p0TrdJXbx18oHM0V8HB1O-a9UwzXE/edit?resourcekey=0-Y7TxpGkjJnTY2bEA3vS-bg&gid=1808",

semantic\_layer\_path="semantic\_layer.yaml"

)

result = query\_agent.process(state.user\_query)

return {

"intent": result["intent"],

"extracted\_context": result["extracted\_context"],

"similar\_contexts": result["similar\_contexts"],

}

def data\_retrieval\_node(state: AgentState) -> Dict[str, Any]:

agent = DataRetrievalAgent(

gsheet\_url="https://docs.google.com/spreadsheets/d/1auyXukmp19gTl0p0TrdJXbx18oHM0V8HB1O-a9UwzXE/edit?resourcekey=0-Y7TxpGkjJnTY2bEA3vS-bg&gid=1808"

)

data = agent.fetch\_data(state.extracted\_context)

return {"data": data}

workflow = StateGraph(AgentState)

workflow.add\_node("query\_understanding", query\_understanding\_node)

workflow.add\_node("data\_retrieval", data\_retrieval\_node)

workflow.add\_node("analysis", lambda state: {

"intent": state.intent,

"extracted\_context": state.extracted\_context,

"data": state.data,

"timeframes": state.timeframes,

"similar\_contexts": state.similar\_contexts,

\*\*create\_analysis\_nodes(load\_config(), ChatGoogleGenerativeAI(model="gemini-pro", temperature=0)).invoke(state)

})

workflow.set\_entry\_point("query\_understanding")

workflow.add\_edge("query\_understanding", "data\_retrieval")

workflow.add\_edge("data\_retrieval", "analysis")

workflow.add\_edge("analysis", END)

app = workflow.compile()

# === Example Usage ===

if \_\_name\_\_ == "\_\_main\_\_":

os.environ["GOOGLE\_API\_KEY"] = "your-google-api-key"

# Semantic Layer

semantic\_layer = {

"tables": {"efficiency\_table": {"columns": {

"region": {"dimension": True, "semantic": ["region", "area"]},

"country\_code": {"dimension": True, "semantic": ["country", "nation"]},

"vehicle\_id": {"dimension": True, "semantic": ["vid", "vehicle"]},

"drive\_date": {"dimension": True, "semantic": ["date", "day"]},

"drive\_month": {"dimension": True, "semantic": ["month"]},

"TKM": {"measure": True},

"Total\_KM": {"measure": True},

"C1\_Kms\_Collected": {"measure": True},

"C2\_Kms\_Collected": {"measure": True},

"C3\_Kms\_Collected": {"measure": True}

}}},

"metrics": {

"efficiency": {"formula": "sum(TKM) / sum(Total\_KM)", "columns": ["TKM", "Total\_KM"]},

"c1\_km\_perc": {"formula": "sum(C1\_Kms\_Collected) / sum(Total\_KM)", "columns": ["C1\_Kms\_Collected", "Total\_KM"]}

}

}

with open("semantic\_layer.yaml", "w") as f:

yaml.dump(semantic\_layer, f)

# Config

config = {

"agent\_mapping": {

"fetch\_metric": ["InsightGenerator"],

"compare\_metric": ["InsightGenerator"],

"rank\_entities": ["InsightGenerator"],

"diagnose\_metric": ["PatternDetector"],

"trend\_analysis": ["InsightGenerator", "PatternDetector"],

"threshold\_check": ["InsightGenerator"],

"list\_entities\_by\_criteria": ["InsightGenerator"],

"summarize\_metric": ["InsightGenerator"],

"get\_recommendation": ["RecommendationEngine"],

"visualize\_metric": ["PlotGenerator"]

}

}

with open("config.yaml", "w") as f:

yaml.dump(config, f)

# Sample Data

sample\_df = pd.read\_csv("sample\_efficiency\_data.csv") # Assume a CSV with the above columns

sample\_df["drive\_date"] = pd.to\_datetime(sample\_df["drive\_date"])

sample\_df["drive\_month"] = pd.to\_datetime(sample\_df["drive\_month"])

state = AgentState(user\_query="What is the c1\_km\_perc for country JP in May 2025? Plot the trend of efficiency in region APAC for past 3 months")

result = app.invoke(state)

result\_to\_print = {k: v for k, v in result.items() if k != "data" and k != "feedback"}

print(json.dumps(result\_to\_print, indent=2))

if result.get("data") is not None:

display(result.get("data"))

if result.get("report"):

print(result["report"])

if result.get("feedback"):

print(f"Feedback applied: {result['feedback']}")