! pip install --quiet faiss-cpu pyyaml langchain\_openai langgraph

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\_openai import ChatOpenAI, OpenAIEmbeddings

from langgraph.graph import StateGraph, END

from langgraph.graph import START

from dataclasses import dataclass, asdict, field

import json

from datetime import datetime, timedelta

import re

import os

from typing import List, Dict, Any, Optional, Annotated

from enum import Enum

from dateutil.relativedelta import relativedelta

import logging

from operator import add

from statsmodels.tsa.arima.model import ARIMA

from statsmodels.tsa.stattools import acf

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LinearRegression

from IPython.display import display

auth.authenticate\_user()

from google.auth import default

creds, \_ = default()

gc = gspread.authorize(creds)

logging.basicConfig(filename='agentic\_app.log', level=logging.INFO,

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

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

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"

THRESHOLD\_CHECK = "threshold\_check"

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

DIAGNOSE\_METRIC = "diagnose\_metric"

TREND\_ANALYSIS = "trend\_analysis"

SUMMARIZE\_METRIC = "summarize\_metric"

GET\_RECOMMENDATION = "get\_recommendation"

VISUALIZE\_METRIC = "visualize\_metric"

PREDICT\_METRIC = "predict\_metric"

CORRELATE\_METRICS = "correlate\_metrics"

ANOMALY\_DETECTION = "anomaly\_detection"

GROUP\_AGGREGATE = "group\_aggregate"

FILTER\_LIST = "filter\_list"

EXPORT\_DATA = "export\_data"

GENERATE\_REPORT = "generate\_report"

@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

all\_analysis\_results: Annotated[list, add] = field(default\_factory=list)

report: Optional[str] = None

timeframes: Optional[Dict] = None

errors: Optional[List[str]] = field(default\_factory=list)

feedback: Optional[Dict] = None

class VectorStore:

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

self.embeddings = embeddings

self.index = None

self.queries = []

self.query\_history = []

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

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):

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]]:

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],

"intent": self.extract\_intent(self.queries[idx]) if hasattr(self, 'extract\_intent') else [],

"score": float(dist)

})

return results

class TimeExpressionHandler:

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

self.now = datetime(2025, 8, 4, 15, 11)

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.get(month\_name.lower(), 1)

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 for time expression: {lower\_expr}")

return {"start": "1970-01-01", "end": reference\_date.strftime("%Y-%m-%d")}

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": [], "types": []}

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, feedback\_type: str = "programmatic"):

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

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

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

self.save\_feedback()

logging.info(f"Added {feedback\_type} feedback for query: {query}")

def collect\_user\_feedback(self, query: str) -> str:

print(

"\nPlease provide your feedback on the response (e.g., 'Accurate report' or 'Missing trend details'):"

)

user\_feedback = input().strip()

if user\_feedback:

self.add\_feedback(query, user\_feedback, "user")

return user\_feedback

return ""

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

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

return {}

feedback\_list: List[str] = self.feedback\_data["feedback"]

common\_issues = max(set(feedback\_list), key=feedback\_list.count) if feedback\_list else ""

positive = sum(

1

for f in feedback\_list

if any(word in f.lower() for word in ["good", "accurate", "helpful"])

)

negative = sum(

1

for f in feedback\_list

if any(word in f.lower() for word in ["bad", "error", "inaccurate", "missing"])

)

neutral = len(feedback\_list) - positive - negative

recent\_negative = [

f

for f in feedback\_list[-5:]

if any(word in f.lower() for word in ["error", "wrong", "inaccurate", "missing"])

]

return {

"common\_issues": common\_issues,

"sentiment": {"positive": positive, "negative": negative, "neutral": neutral},

"recent\_negative": recent\_negative,

}

def learn\_from\_feedback(self) -> str:

insights = self.get\_feedback\_insights()

if not insights:

return "No prior feedback available."

summary = (

f"Common issues: {insights['common\_issues']}, Sentiment: {insights['sentiment']}."

)

if insights.get("recent\_negative"):

summary += f" Avoid repeating: {', '.join(insights['recent\_negative'])}."

return summary

"""### Query understanding agent"""

class QueryUnderstandingAgent:

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

self.llm = ChatOpenAI(model="gpt-4.1-mini", temperature=0.2)

self.embeddings = OpenAIEmbeddings(model="text-embedding-3-small")

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 2025?",

"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", "fetch", "get value"]):

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

if any(keyword in query\_lower for keyword in ["compare", "versus", "vs", "comparison", "higher", "better","highest among","among"]):

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

if any(keyword in query\_lower for keyword in ["top", "bottom", "rank", "highest", "lowest","low","high"]):

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

if any(keyword in query\_lower for keyword in ["below", "above", "threshold", "check"]):

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

if any(keyword in query\_lower for keyword in ["list", "which", "what are"]):

intents.append(IntentType.LIST\_ENTITIES\_BY\_CRITERIA.value)

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

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

if any(keyword in query\_lower for keyword in ["trend", "over time", "history", "past month", "past year"]):

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

if any(keyword in query\_lower for keyword in ["summarize", "summary", "average"]):

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

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

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

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

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

if any(keyword in query\_lower for keyword in ["predict", "forecast", "next"]):

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

if any(keyword in query\_lower for keyword in ["depend", "correlate", "relationship", "how does"]):

intents.append(IntentType.CORRELATE\_METRICS.value)

if any(keyword in query\_lower for keyword in ["anomaly", "unusual", "detect"]):

intents.append(IntentType.ANOMALY\_DETECTION.value)

if any(keyword in query\_lower for keyword in ["aggregate", "group", "by"]):

intents.append(IntentType.GROUP\_AGGREGATE.value)

if any(keyword in query\_lower for keyword in ["filter", "which with"]):

intents.append(IntentType.FILTER\_LIST.value)

if any(keyword in query\_lower for keyword in ["export", "download"]):

intents.append(IntentType.EXPORT\_DATA.value)

if any(keyword in query\_lower for keyword in ["report", "generate report"]):

intents.append(IntentType.GENERATE\_REPORT.value)

if IntentType.DIAGNOSE\_METRIC.value in intents and IntentType.RANK\_ENTITIES.value in intents:

# For “why … low/high …” prefer diagnosis; ranking is noisy here

intents = [i for i in intents if i != IntentType.RANK\_ENTITIES.value]

if (IntentType.FETCH\_METRIC.value in intents and IntentType.TREND\_ANALYSIS.value in intents):

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

if (IntentType.COMPARE\_METRIC.value in intents and IntentType.RANK\_ENTITIES.value in intents):

intents = [i for i in intents if i != IntentType.RANK\_ENTITIES.value]

if intents:

return intents

prompt = f"""

Analyze the query and determine all possible intents from these options using chain-of-thought reasoning. Return a JSON array of intent names.

Options: {{[i.value for i in IntentType]}}

Thought Process: [identify keywords, infer intent, resolve overlaps with context]

Follow these rules strictly:

1. Return ONLY JSON, Give additional text or explanations only when direct mapping failed.

2. Use the exact intent names from the provided list

3. If unsure, default to ["fetch\_metric"]

Query: "{query}"

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 content.startswith('```json'):

content = content[7:-3].strip()

return json.loads(content)

except json.JSONDecodeError:

print(f"JSON decode error in intent extraction: {response.content}")

return [IntentType.FETCH\_METRIC.value]

def \_is\_ranking\_target(self, query: str, field: str) -> bool:

"""More precise determination if a field is the ranking target"""

query\_lower = query.lower()

field\_lower = field.lower()

if (field\_lower in query\_lower.replace('\_', ' ') or

any(synonym in query\_lower for synonym in self.\_get\_field\_synonyms(field))):

return True

if field == 'user\_email' and ('driver' in query\_lower or 'drivers' in query\_lower):

return True

return False

def \_get\_field\_synonyms(self, field: str) -> List[str]:

"""Returns query terms that should map to each field"""

return {

'vehicle\_id': ['vehicle', 'vid', 'vehicles'],

'country\_code': ['country', 'countries'],

'driving\_hub\_id': ['hub', 'hubs'],

'user\_email': ['driver', 'drivers']

}.get(field, [])

def \_detect\_ranking\_entity(self, query: str, available\_columns: List[str]) -> Optional[str]:

"""Determines which entity should be marked as 'all' in ranking queries"""

query\_lower = query.lower()

term\_mapping = {

'vehicle': 'vehicle\_id',

'country': 'country\_code',

'hub': 'driving\_hub\_id',

'drivers': 'user\_email',

'vid': 'vehicle\_id',

'region': 'region', # <-- NEW

}

for term, column in term\_mapping.items():

if term in query\_lower and column in available\_columns:

return column

# Fallbacks

if 'vehicle' in query\_lower or 'vid' in query\_lower:

return 'vehicle\_id' if 'vehicle\_id' in available\_columns else None

if 'driver' in query\_lower:

return 'user\_email' if 'user\_email' in available\_columns else None

return None

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

"""

Robust entity + metric extraction with:

- Ranking vs. Comparative disambiguation (highest/lowest vs. higher/lower A or B)

- Default ranking target fallback (country\_code → region → driving\_hub\_id → vehicle\_id → user\_email)

- Region recognized as a ranking target

- Numeric condition extraction (e.g., 'TKM > 1000')

- LLM-backed semantic mapping with safe JSON parsing

- Fallback: capture all 2-letter uppercase tokens (BR, TW, MX, ...) as country\_code for compare flows

- Keep vehicle\_id values as strings

- PRESERVE filter entities when adding ranking targets

- Handle multiple values for same field

- Only use "all" when no specific values mentioned

"""

query\_lower = query.lower()

# --------- 1) Detect ranking vs. comparative intent ----------

ranking\_words = ["top", "bottom", "highest", "lowest", "rank"]

comparative = bool(re.search(r"\b(higher|lower)\b", query\_lower))

is\_ranking = (any(w in query\_lower for w in ranking\_words) and not comparative)

# --------- 2) If ranking, detect target column (includes 'region') ----------

available\_cols = list(self.semantic\_layer["tables"]["efficiency\_table"]["columns"].keys())

ranking\_target = None

if is\_ranking:

ranking\_target = self.\_detect\_ranking\_entity(query, available\_cols)

# --------- 3) Pre-compute numeric filter conditions ----------

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

numeric\_cols = [c for c, info in columns\_info.items() if info.get("type") in ("float", "int")]

numeric\_conditions = []

for col in numeric\_cols:

pattern = rf"{col}\s\*([<>]=?|==?)\s\*(\d+(?:\.\d+)?)"

for m in re.finditer(pattern, query, re.IGNORECASE):

numeric\_conditions.append({

"field": col,

"operator": m.group(1),

"value": float(m.group(2)),

"condition\_type": "numeric\_filter"

})

# --------- 4) Build semantic map for LLM hinting ----------

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

semantic\_map = {}

for col, info in columns\_info.items():

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

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

for metric in metrics:

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

# --------- 5) Enhanced LLM prompt for multiple entity extraction ----------

prompt = f"""

Analyze the query and extract ALL entities mentioned, including:

- Countries (e.g., BR, TW, ID, US) → country\_code

- Drivers (e.g., driver names, emails) → user\_email

- Vehicles (e.g., VID 12345) → vehicle\_id

- Regions (e.g., APAC, LATAM, NA, EMEA) → region

- Hubs (e.g., hub names) → driving\_hub\_id

- Season names (which represents a set of data like swarm, 2025) → season\_name

Return JSON with:

1) "metrics": list of metric names

2) "entities": list of objects with "field" and "value"

CRITICAL RULES:

- Extract ALL entities mentioned, not just the main one

- For ranking queries ("top/bottom N"), include BOTH the ranking target AND any filter entities

- Preserve exact values (BR, not brazil; user@email.com, not "the driver")

- Return entities even if they appear to be constraints/filters

- For "country BR", return: {{"field": "country\_code", "value": "BR"}}

- For "drivers", return: {{"field": "user\_email", "value": "all"}} ONLY if no specific driver is mentioned

- If entity has specific values in query, use those values

- If only entity type is mentioned without specific values, use "all"

Examples:

- "List bottom 5 drivers in country BR" → entities: [{{"field": "country\_code", "value": "BR"}}, {{"field": "user\_email", "value": "all"}}]

- "Show top 3 drivers: john@email.com, mary@email.com in US" → entities: [{{"field": "user\_email", "value": "john@email.com"}}, {{"field": "user\_email", "value": "mary@email.com"}}, {{"field": "country\_code", "value": "US"}}]

- "Rank vehicles by efficiency" → entities: [{{"field": "vehicle\_id", "value": "all"}}]

Semantic hints: {json.dumps(semantic\_map)}

Query: "{query}"

"""

result = {"metrics": ["efficiency"], "entities": []}

try:

resp = self.llm.invoke(prompt)

cleaned = (resp.content or "").strip()

cleaned = cleaned.replace("```json", "").replace("```", "").strip()

parsed = json.loads(cleaned)

# Ensure structure

if isinstance(parsed, dict):

if parsed.get("metrics"):

result["metrics"] = parsed["metrics"]

if parsed.get("entities"):

result["entities"] = parsed["entities"]

except Exception as e:

print(f"LLM parsing failed: {e}")

# Fall back to defaults if LLM parsing fails

pass

# Default metric if LLM didn't provide

if not result.get("metrics"):

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

if "entities" not in result or not isinstance(result["entities"], list):

result["entities"] = []

# --------- 6) Enhanced fallback: detect multiple entity types and values ----------

# Extract country codes (handle multiple)

country\_codes = re.findall(r"\b[A-Z]{2}\b", query)

existing\_countries = {e.get("value") for e in result["entities"] if e.get("field") == "country\_code"}

for code in country\_codes:

if code not in existing\_countries:

result["entities"].append({"field": "country\_code", "value": code})

# Extract driver references (only add "all" if no specific drivers mentioned)

driver\_keywords = ["driver", "drivers", "user", "email", "user\_email"]

if any(keyword in query\_lower for keyword in driver\_keywords):

has\_specific\_driver = any(

e.get("field") == "user\_email" and

e.get("value") not in (None, "all")

for e in result["entities"]

)

if not has\_specific\_driver:

# Check if we should add user\_email: all for ranking

has\_all\_driver = any(e.get("field") == "user\_email" and e.get("value") == "all" for e in result["entities"])

if not has\_all\_driver:

result["entities"].append({"field": "user\_email", "value": "all"})

# Extract vehicle references (only add "all" if no specific vehicles mentioned)

vehicle\_keywords = ["vehicle", "vehicles", "vid", "vids", "car", "cars"]

if any(keyword in query\_lower for keyword in vehicle\_keywords):

has\_specific\_vehicle = any(

e.get("field") == "vehicle\_id" and

e.get("value") not in (None, "all")

for e in result["entities"]

)

if not has\_specific\_vehicle:

has\_all\_vehicle = any(e.get("field") == "vehicle\_id" and e.get("value") == "all" for e in result["entities"])

if not has\_all\_vehicle:

result["entities"].append({"field": "vehicle\_id", "value": "all"})

# --------- 7) Enhanced ranking logic: only add "all" if no specific values exist ----------

if is\_ranking:

# Use the existing ranking\_target detection if available, otherwise detect from query

if not ranking\_target:

ranking\_target = self.\_detect\_ranking\_entity(query, available\_cols)

# If still not found, use enhanced detection based on query content

if not ranking\_target:

if any(word in query\_lower for word in ["driver", "user", "email"]):

ranking\_target = "user\_email"

elif any(word in query\_lower for word in ["vehicle", "vid", "car"]):

ranking\_target = "vehicle\_id"

elif any(word in query\_lower for word in ["country", "region", "hub"]):

ranking\_target = "country\_code"

else:

# Original fallback order (maintains backward compatibility)

for c in ["country\_code", "region", "driving\_hub\_id", "vehicle\_id", "user\_email"]:

if c in available\_cols:

ranking\_target = c

break

if ranking\_target:

# Check if we already have specific values for the ranking target field

has\_specific\_values = any(

e.get("field") == ranking\_target and

e.get("value") not in (None, "all") and

e.get("condition\_type") != "numeric\_filter"

for e in result["entities"]

)

# Only add "all" if no specific values exist for the ranking target

if not has\_specific\_values:

# Remove any existing "all" values for the same field to avoid duplicates

result["entities"] = [

e for e in result["entities"]

if not (e.get("field") == ranking\_target and e.get("value") == "all")

]

# Add the 'all' marker so AnalysisAgent will group by this field

result["entities"].append({"field": ranking\_target, "value": "all"})

# --------- 8) Merge numeric conditions ----------

if numeric\_conditions:

result["entities"].extend(numeric\_conditions)

# --------- 9) Enforce vehicle\_id as string ----------

for e in result["entities"]:

if e.get("field") == "vehicle\_id" and "value" in e and e["value"] is not None:

e["value"] = str(e["value"])

return result

def extract\_timeframe(self, query: str, intents: List[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", "full year"]

reference\_date = self.time\_handler.now

for sentence in sentences:

if not sentence.strip():

continue

sentence\_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.lower()))

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 and "start" in parsed\_time and parsed\_time["start"]:

for intent in sentence\_intents:

timeframes.append({"period\_in\_query": time\_expr, "start": parsed\_time["start"], "end": parsed\_time["end"], "intent\_ref": intent})

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 and "start" in parsed\_time and parsed\_time["start"]:

for intent in sentence\_intents:

timeframes.append({"period\_in\_query": time\_expr, "start": parsed\_time["start"], "end": parsed\_time["end"], "intent\_ref": intent})

for expr in time\_exprs:

if expr in sentence.lower():

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

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

for intent in sentence\_intents:

timeframes.append({"period\_in\_query": expr, "start": parsed\_time["start"], "end": parsed\_time["end"], "intent\_ref": intent})

if not timeframes:

timeframes = [{"period\_in\_query": "all data", "start": "1970-01-01", "end": reference\_date.strftime("%Y-%m-%d"), "intent\_ref": intent} for intent in intents]

return timeframes

def \_detect\_aggregation(self, query: str) -> str:

q = query.lower()

if any(w in q for w in ["standard deviation", "std dev", "stddev", "stdev", "std", "sd"]):

return "std"

if any(w in q for w in ["average", "avg", "mean"]):

return "mean"

if any(w in q for w in ["sum", "total", "total of"]):

return "sum"

if any(w in q for w in ["minimum", "min", "lowest"]):

return "min"

if any(w in q for w in ["maximum", "max", "highest"]):

return "max"

return "mean"

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

intents = self.extract\_intent(query)

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

timeframes = self.extract\_timeframe(query, intents)

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

agg = self.\_detect\_aggregation(query)

query\_analysis\_list = []

for intent in intents:

intent\_timeframes = [t for t in timeframes if t["intent\_ref"] == intent]

query\_analysis\_list.append({

"user\_query": query,

"intent": intent,

"metrics": entities\_metrics.get("metrics", ["efficiency"]),

"entities": entities\_metrics.get("entities", []),

"timeframes": intent\_timeframes,

"aggregation": agg

})

output = {

"query\_analysis": query\_analysis\_list,

"similar\_contexts": similar\_contexts

}

self.vector\_store.add\_query(query)

print(output)

return output

"""### Data Retrieval Agent"""

class DataRetrievalAgent:

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

self.gsheet\_url = gsheet\_url

self.semantic\_layer = load\_semantic\_layer("semantic\_layer.yaml")

self.\_last\_full\_df = None

self.allowed\_fields\_for\_filtering = [

"season\_name", "region", "vehicle\_id", "country\_code",

"user\_email", "driving\_hub\_id"

]

self.\_add\_numeric\_columns\_to\_filtering()

def \_add\_numeric\_columns\_to\_filtering(self):

numeric\_columns = [

col for col, info in self.semantic\_layer["tables"]["efficiency\_table"]["columns"].items()

if info.get("type") in ["float", "int"]

]

self.allowed\_fields\_for\_filtering.extend(numeric\_columns)

def \_enforce\_dtypes(self, df: pd.DataFrame):

dtype\_map = {

"string": "string",

"float": "float64",

"int": "int64",

"date": "datetime64[ns]"

}

columns\_info = self.semantic\_layer.get("tables", {}).get("efficiency\_table", {}).get("columns", {})

for col\_name, col\_info in columns\_info.items():

col\_type = col\_info.get("type")

if col\_name in df.columns and col\_type in dtype\_map:

try:

if col\_type == "date":

df[col\_name] = pd.to\_datetime(df[col\_name], errors='coerce')

else:

if col\_type in ["float", "int"]:

df[col\_name] = (df[col\_name]

.astype(str)

.str.replace(r'[^\d\.-]', '', regex=True)

.replace('', 'NaN'))

df[col\_name] = df[col\_name].astype(dtype\_map[col\_type])

logging.info(f"Converted {col\_name} to {col\_type} ({dtype\_map[col\_type]})")

except Exception as e:

logging.error(f"Error converting {col\_name} to {col\_type} ({dtype\_map[col\_type]}): {e}")

elif col\_name not in df.columns:

logging.warning(f"Column {col\_name} not found in DataFrame. Skipping conversion.")

return df

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

try:

# Load data from Google Sheet

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

worksheet = sheet.get\_worksheet(0)

data = worksheet.get\_all\_records()

df\_processed = pd.DataFrame(data)

df\_processed.columns = df\_processed.columns.str.strip()

# print(df\_processed['country\_code'].value\_counts())

except Exception as e:

print(f"Sheet access error: {str(e)}")

return pd.DataFrame()

df\_processed = self.\_enforce\_dtypes(df\_processed)

if 'drive\_date' in df\_processed.columns:

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

df\_processed["drive\_month"] = pd.to\_datetime(

df\_processed["drive\_date"].dt.to\_period('M').dt.to\_timestamp(),

errors='coerce'

)

self.\_last\_full\_df = df\_processed.copy()

query\_analysis\_list = extracted\_context.get("query\_analysis", [])

is\_prediction\_query = any(

(ctx.get("intent") == IntentType.PREDICT\_METRIC.value) for ctx in query\_analysis\_list

)

if not query\_analysis\_list:

print("query\_analysis\_list is empty. Cannot perform filtering.")

return pd.DataFrame(columns=list(df\_processed.columns) + ['intent'])

entities\_to\_process = []

for ctx in query\_analysis\_list:

for entity in ctx.get("entities", []):

field = entity.get("field")

value = entity.get("value")

condition\_type = entity.get("condition\_type")

if field and field in self.allowed\_fields\_for\_filtering and field in df\_processed.columns:

if condition\_type == "numeric\_filter" or value is not None:

entities\_to\_process.append(entity)

elif field:

print(f"Ignoring entity field '{field}' - not in allowed fields or DataFrame columns")

entity\_conditions\_by\_field = {}

for entity in entities\_to\_process:

field = entity["field"]

if entity.get("condition\_type") == "numeric\_filter":

operator = entity["operator"]

value = entity["value"]

if field not in entity\_conditions\_by\_field:

entity\_conditions\_by\_field[field] = []

entity\_conditions\_by\_field[field].append(f"({field} {operator} {value})")

elif entity["value"] == 'all':

continue

else:

value = entity["value"]

if pd.api.types.is\_numeric\_dtype(df\_processed[field]):

try:

num\_value = pd.to\_numeric(value)

condition = f"({field} == {num\_value})"

except (ValueError, TypeError):

print(f"Could not convert value '{value}' to numeric for column '{field}'")

continue

else:

condition = f"({field} == {repr(value)})"

if field not in entity\_conditions\_by\_field:

entity\_conditions\_by\_field[field] = []

entity\_conditions\_by\_field[field].append(condition)

entity\_combined\_conditions = []

for field, conditions\_list in entity\_conditions\_by\_field.items():

if conditions\_list:

entity\_combined\_conditions.append("( " + " or ".join(conditions\_list) + " )")

combined\_entity\_filter\_condition = " and ".join(entity\_combined\_conditions)

df\_filtered\_entity = df\_processed.copy()

if combined\_entity\_filter\_condition:

try:

df\_filtered\_entity = df\_processed.query(combined\_entity\_filter\_condition, engine='python').copy()

except Exception as e:

print(f"Error applying entity filters: {str(e)}")

return pd.DataFrame(columns=list(df\_processed.columns) + ['intent'])

df\_filtered\_entity['intent'] = [[] for \_ in range(len(df\_filtered\_entity))]

all\_starts = [pd.to\_datetime(t["start"]) for ctx in query\_analysis\_list for t in ctx.get("timeframes", []) if t.get("start")]

all\_ends = [pd.to\_datetime(t["end"]) for ctx in query\_analysis\_list for t in ctx.get("timeframes", []) if t.get("end")]

start\_date = min(all\_starts) if all\_starts else pd.to\_datetime("1970-01-01")

end\_date = max(all\_ends) if all\_ends else datetime.now()

# print(f"condition filtered data shape: {df\_filtered\_entity.shape}")

if is\_prediction\_query:

start\_date = pd.to\_datetime("1970-01-01")

end\_date = pd.to\_datetime(pd.Timestamp.now().date())

now\_date = pd.to\_datetime(pd.Timestamp.now().date())

if end\_date > now\_date:

end\_date = now\_date

if start\_date > end\_date:

start\_date = pd.to\_datetime("1970-01-01")

# print(f"start\_date: {start\_date}, end\_date: {end\_date}")

date\_filt\_data = df\_filtered\_entity[

(df\_filtered\_entity['drive\_date'] >= start\_date) &

(df\_filtered\_entity['drive\_date'] <= end\_date)

].copy()

# print(f"date filtered data shape: {date\_filt\_data.shape}")

for ctx in query\_analysis\_list:

intent\_ref = ctx.get("intent")

timeframes = ctx.get("timeframes", [])

if not intent\_ref:

continue

intent\_time\_conditions = []

for tf in timeframes:

tf\_start = pd.to\_datetime(tf.get("start"))

tf\_end = pd.to\_datetime(tf.get("end"))

if tf\_start and tf\_end:

intent\_time\_conditions.append(

f"(drive\_date >= '{tf\_start.strftime('%Y-%m-%d')}' and drive\_date <= '{tf\_end.strftime('%Y-%m-%d')}')"

)

if intent\_time\_conditions:

combined\_time\_condition = "( " + " or ".join(intent\_time\_conditions) + " )"

# print(f"Time conditions present: {combined\_time\_condition}")

try:

mask = date\_filt\_data.query(combined\_time\_condition, engine='python').index

for idx in mask:

if idx in date\_filt\_data.index and intent\_ref not in date\_filt\_data.loc[idx, 'intent']:

date\_filt\_data.loc[idx, 'intent'].append(intent\_ref)

except Exception as e:

logging.error(f"Error applying timeframe filter: {str(e)}")

for idx in date\_filt\_data.index:

if intent\_ref not in date\_filt\_data.loc[idx, 'intent']:

date\_filt\_data.loc[idx, 'intent'].append(intent\_ref)

print(f"final data shape : {date\_filt\_data.shape}")

return date\_filt\_data

"""### analysis agent"""

class AnalysisAgent:

print("coming here")

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

self.semantic\_layer = semantic\_layer

self.\_agg\_map = {

"mean": lambda s: s.mean(),

"sum": lambda s: s.sum(),

"min": lambda s: s.min(),

"max": lambda s: s.max(),

"std": lambda s: s.std(ddof=1),

}

def \_parse\_formula(self, formula: str) -> tuple:

parts = formula.split('/')

num = parts[0].replace('SUM(', '').replace(')', '').strip()

den = parts[1].replace('SUM(', '').replace(')', '').strip() if len(parts) > 1 else '1'

return num, den

def compute\_metrics(self, df: pd.DataFrame, metrics: List[str], context: Dict) -> Dict[str, pd.DataFrame]:

"""

Compute metrics for each intent in the extracted context.

Enhancements:

- For COMPARE\_METRIC, dynamically groups by the entity field being compared

(e.g., country\_code, region, vehicle\_id, driving\_hub\_id), and—if specific

values are present in the query—filters to just those values in the output.

"""

if df.empty:

print("No data available for analysis!")

return {}

query\_analysis = context.get('query\_analysis', [])

if not query\_analysis:

print("No query analysis available!")

return {}

results = {}

for intent\_ctx in query\_analysis:

intent = intent\_ctx.get('intent')

if not intent:

continue

entities = intent\_ctx.get('entities', [])

timeframes = intent\_ctx.get('timeframes', [])

intent\_metrics = intent\_ctx.get('metrics', metrics)

agg\_name = intent\_ctx.get('aggregation', 'mean')

agg\_fn = self.\_agg\_map.get(agg\_name, self.\_agg\_map['mean'])

if not intent\_metrics:

intent\_metrics = ["efficiency"]

# Time-window filtering (if present)

intent\_df = df.copy()

if timeframes and 'drive\_date' in intent\_df.columns:

start = pd.to\_datetime(timeframes[0].get('start', '1970-01-01'))

end = pd.to\_datetime(timeframes[0].get('end', '9999-12-31'))

intent\_df = intent\_df[(intent\_df['drive\_date'] >= start) &

(intent\_df['drive\_date'] <= end)]

metric\_results = []

# ---- Helper to discover compare field & selected values (if any) ----

def \_discover\_compare\_field\_and\_values(\_entities: List[dict]) -> tuple:

# Prefer entities that are not 'all' and not numeric filters

explicit = [e for e in \_entities

if isinstance(e, dict)

and e.get("condition\_type") != "numeric\_filter"

and e.get("value") not in (None, "all")]

# Fall back to any non-numeric-filter entity to find the field

any\_entity = [e for e in \_entities

if isinstance(e, dict)

and e.get("condition\_type") != "numeric\_filter"]

if explicit:

field = explicit[0].get("field")

values = [str(e.get("value")) for e in explicit if e.get("field") == field]

return field, values

elif any\_entity:

field = any\_entity[0].get("field")

return field, [] # no explicit values provided

return None, []

for metric in intent\_metrics:

if metric in self.semantic\_layer['metrics']:

# Metric defined in semantic layer by formula (e.g., efficiency)

formula = self.semantic\_layer['metrics'][metric]['formula']

num\_expr, den\_expr = self.\_parse\_formula(formula)

# ---------- SPECIAL PATH: COMPARE METRIC ----------

if intent == IntentType.COMPARE\_METRIC.value:

compare\_field, selected\_values = \_discover\_compare\_field\_and\_values(entities)

if compare\_field and compare\_field in intent\_df.columns:

grouped = (intent\_df

.groupby(compare\_field, dropna=False)

.agg({num\_expr: "sum", den\_expr: "sum"})

.reset\_index())

# Compute ratio safely

grouped["\_\_value\_\_"] = grouped.apply(

lambda r: (r[num\_expr] / r[den\_expr]) if r[den\_expr] != 0 else np.nan, axis=1

)

# If the user explicitly listed values (e.g., BR, TW), keep only those

if selected\_values:

sv\_upper = set(v.upper() for v in selected\_values)

grouped = grouped[grouped[compare\_field].astype(str).str.upper().isin(sv\_upper)]

for \_, row in grouped.iterrows():

metric\_results.append({

'field': compare\_field,

'field\_value': row[compare\_field],

'metric': metric,

'value': float(row["\_\_value\_\_"]) if pd.notnull(row["\_\_value\_\_"]) else np.nan,

'support\_numbers': f"{num\_expr}: {row[num\_expr]:.2f}/{den\_expr}: {row[den\_expr]:.2f}"

})

# Done for this metric

continue

# ---------- SPECIAL PATH: TREND\_ANALYSIS ----------

if intent == IntentType.TREND\_ANALYSIS.value and metric in self.semantic\_layer['metrics']:

formula = self.semantic\_layer['metrics'][metric]['formula']

num\_expr, den\_expr = self.\_parse\_formula(formula)

# Group by drive\_month for trend

if "drive\_month" in intent\_df.columns:

grouped = (intent\_df

.groupby("drive\_month", as\_index=False)

.agg({num\_expr: "sum", den\_expr: "sum"}))

grouped["value"] = grouped.apply(

lambda r: (r[num\_expr] / r[den\_expr]) if r[den\_expr] != 0 else np.nan, axis=1

)

metric\_results.extend([

{

"field": "drive\_month",

"field\_value": row["drive\_month"].strftime("%Y-%m"),

"metric": metric,

"value": float(row["value"]),

"support\_numbers": f"{num\_expr}: {row[num\_expr]:.2f}/"

f"{den\_expr}: {row[den\_expr]:.2f}"

}

for \_, row in grouped.iterrows()

])

results[intent] = pd.DataFrame(metric\_results)

continue

# ---------- DEFAULT PATH (non-compare, or compare fallback) ----------

# If any 'all' entity exists, return per-group values for that field

all\_entities = [e for e in entities if isinstance(e, dict) and e.get('value') == 'all']

if all\_entities:

for entity in all\_entities:

field = entity['field']

if field not in intent\_df.columns:

continue

for name, group in intent\_df.groupby(field, dropna=False):

if pd.isna(name) or (isinstance(name, str) and name.strip() == ""):

continue # skip NaN groups

total\_num = group[num\_expr].sum()

total\_den = group[den\_expr].sum()

value = (total\_num / total\_den) if total\_den != 0 else np.nan

metric\_results.append({

'field': field,

'field\_value': name,

'metric': metric,

'value': value,

'support\_numbers': f"{num\_expr}: {total\_num:.2f}/{den\_expr}: {total\_den:.2f}"

})

else:

total\_num = intent\_df[num\_expr].sum()

total\_den = intent\_df[den\_expr].sum()

value = (total\_num / total\_den) if total\_den != 0 else np.nan

metric\_results.append({

'field': 'overall',

'field\_value': 'all',

'metric': metric,

'value': value,

'support\_numbers': f"{num\_expr}: {total\_num:.2f}/{den\_expr}: {total\_den:.2f}"

})

elif metric in intent\_df.columns:

# Raw numeric column aggregation (non-formula metric)

if intent == IntentType.COMPARE\_METRIC.value:

compare\_field, selected\_values = \_discover\_compare\_field\_and\_values(entities)

if compare\_field and compare\_field in intent\_df.columns:

grp = intent\_df.groupby(compare\_field, dropna=False)[metric].apply(agg\_fn).reset\_index()

if selected\_values:

sv\_upper = set(v.upper() for v in selected\_values)

grp = grp[grp[compare\_field].astype(str).str.upper().isin(sv\_upper)]

for \_, row in grp.iterrows():

metric\_results.append({

'field': compare\_field,

'field\_value': row[compare\_field],

'metric': f"{agg\_name}({metric})",

'value': float(row[metric]) if pd.notnull(row[metric]) else np.nan,

'support\_numbers': f"{agg\_name}({metric}) over "

f"{len(intent\_df[intent\_df[compare\_field] == row[compare\_field]])} rows"

})

continue # handled

# Non-compare, or compare fallback

all\_entities = [e for e in entities if isinstance(e, dict) and e.get('value') == 'all']

if all\_entities:

for entity in all\_entities:

field = entity['field']

if field not in intent\_df.columns:

continue

for name, group in intent\_df.groupby(field, dropna=False):

val = agg\_fn(group[metric])

metric\_results.append({

'field': field,

'field\_value': name,

'metric': f"{agg\_name}({metric})",

'value': float(val) if pd.notnull(val) else np.nan,

'support\_numbers': f"{agg\_name}({metric}) over {len(group)} rows"

})

else:

val = agg\_fn(intent\_df[metric])

metric\_results.append({

'field': 'overall',

'field\_value': 'all',

'metric': f"{agg\_name}({metric})",

'value': float(val) if pd.notnull(val) else np.nan,

'support\_numbers': f"{agg\_name}({metric}) over {len(intent\_df)} rows"

})

else:

# Unknown metric name

print(f"Skipping unknown metric: {metric}")

if metric\_results:

results[intent] = pd.DataFrame(metric\_results)

return results

# def compute\_metrics(self, df: pd.DataFrame, metrics: List[str], context: Dict) -> Dict[str, pd.DataFrame]:

# if df.empty:

# print("No data available for analysis!")

# return {}

# query\_analysis = context.get('query\_analysis', [])

# if not query\_analysis:

# print("No query analysis available!")

# return {}

# results = {}

# # print(f"query\_analysis: {query\_analysis}")

# # print(f"metrics: {metrics}")

# for intent\_ctx in query\_analysis:

# intent = intent\_ctx.get('intent')

# print(f"intent: {intent}")

# if not intent:

# continue

# entities = intent\_ctx.get('entities', [])

# timeframes = intent\_ctx.get('timeframes', [])

# intent\_metrics = intent\_ctx.get('metrics', metrics)

# agg\_name = intent\_ctx.get('aggregation', 'mean')

# agg\_fn = self.\_agg\_map.get(agg\_name, self.\_agg\_map['mean'])

# if not intent\_metrics:

# intent\_metrics = ["efficiency"]

# intent\_df = df.copy()

# if timeframes and 'drive\_date' in intent\_df.columns:

# start = pd.to\_datetime(timeframes[0].get('start', '1970-01-01'))

# end = pd.to\_datetime(timeframes[0].get('end', '9999-12-31'))

# intent\_df = intent\_df[(intent\_df['drive\_date'] >= start) &

# (intent\_df['drive\_date'] <= end)]

# metric\_results = []

# print(f"intent\_metrics: {intent\_metrics}")

# for metric in intent\_metrics:

# print(f"intent\_metrics metric: {metric}")

# if metric in self.semantic\_layer['metrics']:

# formula = self.semantic\_layer['metrics'][metric]['formula']

# num\_expr, den\_expr = self.\_parse\_formula(formula)

# all\_entities = [e for e in entities if isinstance(e, dict) and e.get('value') == 'all']

# if all\_entities:

# for entity in all\_entities:

# field = entity['field']

# if field not in intent\_df.columns:

# continue

# for name, group in intent\_df.groupby(field):

# total\_num = group[num\_expr].sum()

# total\_den = group[den\_expr].sum()

# value = (total\_num / total\_den) if total\_den != 0 else np.nan

# metric\_results.append({

# 'field': field,

# 'field\_value': name,

# 'metric': metric,

# 'value': value,

# 'support\_numbers': f"{num\_expr}: {total\_num:.2f}/{den\_expr}: {total\_den:.2f}"

# })

# else:

# total\_num = intent\_df[num\_expr].sum()

# total\_den = intent\_df[den\_expr].sum()

# value = (total\_num / total\_den) if total\_den != 0 else np.nan

# metric\_results.append({

# 'field': 'overall',

# 'field\_value': 'all',

# 'metric': metric,

# 'value': value,

# 'support\_numbers': f"{num\_expr}: {total\_num:.2f}/{den\_expr}: {total\_den:.2f}"

# })

# elif metric in intent\_df.columns:

# # print(f"intent\_df, metric: {metric}")

# all\_entities = [e for e in entities if isinstance(e, dict) and e.get('value') == 'all']

# if all\_entities:

# for entity in all\_entities:

# field = entity['field']

# if field not in intent\_df.columns:

# continue

# for name, group in intent\_df.groupby(field):

# val = agg\_fn(group[metric])

# metric\_results.append({

# 'field': field,

# 'field\_value': name,

# 'metric': f"{agg\_name}({metric})",

# 'value': float(val) if pd.notnull(val) else np.nan,

# 'support\_numbers': f"{agg\_name}({metric}) over {len(group)} rows"

# })

# else:

# print("metric going to agg\_fn")

# val = agg\_fn(intent\_df[metric])

# metric\_results.append({

# 'field': 'overall',

# 'field\_value': 'all',

# 'metric': f"{agg\_name}({metric})",

# 'value': float(val) if pd.notnull(val) else np.nan,

# 'support\_numbers': f"{agg\_name}({metric}) over {len(intent\_df)} rows"

# })

# else:

# print(f"Skipping unknown metric: {metric}")

# if metric\_results:

# results[intent] = pd.DataFrame(metric\_results)

# return results

"""### Orchestrator Agent"""

class OrchestratorAgent:

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

self.query\_agent = QueryUnderstandingAgent(gsheet\_url, semantic\_layer\_path)

self.data\_agent = DataRetrievalAgent(gsheet\_url)

self.analysis\_agent = AnalysisAgent(load\_semantic\_layer(semantic\_layer\_path))

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

self.narrator\_llm = ChatOpenAI(model="gpt-4o-mini", temperature=0.2)

# Configure intent handlers

self.intent\_handlers = {

IntentType.FETCH\_METRIC.value: self.\_handle\_fetch\_metric,

IntentType.COMPARE\_METRIC.value: self.\_handle\_compare\_metric,

IntentType.RANK\_ENTITIES.value: self.\_handle\_rank\_entities,

IntentType.THRESHOLD\_CHECK.value: self.\_handle\_threshold\_check,

IntentType.LIST\_ENTITIES\_BY\_CRITERIA.value: self.\_handle\_list\_entities,

IntentType.TREND\_ANALYSIS.value: self.\_handle\_trend\_analysis,

IntentType.SUMMARIZE\_METRIC.value: self.\_handle\_summarize,

IntentType.VISUALIZE\_METRIC.value: self.\_handle\_visualize,

IntentType.PREDICT\_METRIC.value: self.\_handle\_predict,

IntentType.DIAGNOSE\_METRIC.value: self.\_handle\_diagnose,

}

def process(self, state: AgentState) -> AgentState:

try:

# Step 1: Query Understanding

state.extracted\_context = self.query\_agent.process(state.user\_query)

logging.info(f"Extracted context: {state.extracted\_context}")

# Step 2: Data Retrieval

state.data = self.data\_agent.fetch\_data(state.extracted\_context)

logging.info(f"Retrieved {len(state.data)} records")

self.\_working\_df = state.data

self.\_raw\_df = getattr(self.data\_agent, "\_last\_full\_df", None)

# Step 3: Analysis

all\_metrics = [m for ctx in state.extracted\_context['query\_analysis'] for m in ctx.get('metrics', [])]

if not all\_metrics:

all\_metrics = ["efficiency"]

metric\_results = self.analysis\_agent.compute\_metrics(

state.data,

metrics=all\_metrics,

context=state.extracted\_context

)

state.analysis\_results = {}

logging.info(f"Analysis results: {metric\_results}")

# Step 4: Intent-Specific Processing

for intent\_ctx in state.extracted\_context["query\_analysis"]:

intent = intent\_ctx["intent"]

intent\_data = metric\_results.get(intent, pd.DataFrame())

if intent in self.intent\_handlers:

handler\_result = self.intent\_handlers[intent](intent\_data, intent\_ctx)

state.analysis\_results[intent] = handler\_result

else:

state.analysis\_results[intent] = self.\_default\_handler(intent\_data, intent\_ctx)

# Step 5: Generate Final Report

state.report = self.\_generate\_conversational\_report(state)

return state

except Exception as e:

state.errors.append(f"Processing error: {str(e)}")

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

return state

# --- Intent Handlers ---

def \_handle\_fetch\_metric(self, data: pd.DataFrame, context: Dict) -> Dict:

if data.empty:

return {"error": "No data found for specified criteria"}

return {

"type": "metric\_value",

"data": data.to\_dict(orient='records'),

"support\_numbers": data.iloc[0].get("support\_numbers", "") if not data.empty else ""

}

def \_handle\_compare\_metric(self, data: pd.DataFrame, context: Dict) -> Dict:

if data.empty:

return {"error": "No data available for comparison"}

try:

ents = [e for e in context.get("entities", []) if e.get("value") not in (None, "all")]

raw = (context.get("user\_query") or "")

tokens = re.findall(r"\b[A-Z]{2}\b", raw)

for t in tokens:

if all(str(e["value"]).upper() != t for e in ents):

ents.append({"field": "country\_code", "value": t})

wanted = {str(e["value"]).upper() for e in ents}

if len(wanted) < 2:

return {"error": "Need at least two entities to compare"}

records = data.to\_dict("records")

compare\_rows = [r for r in records if str(r.get("field\_value")).upper() in wanted]

if len(compare\_rows) < 2:

return {"error": "Could not find data for all requested entities"}

return {

"type": "comparison",

"data": compare\_rows,

"metadata": {

"entities": [{"field": "country\_code", "value": v} for v in sorted(wanted)],

"metric": data.iloc[0]["metric"] if not data.empty else "unknown"

}

}

except Exception as e:

return {

"error": f"Comparison failed: {str(e)}",

"original\_data": data.to\_dict(orient='records') if not data.empty else []

}

def \_handle\_rank\_entities(self, data: pd.DataFrame, context: Dict) -> Dict:

if data.empty:

return {"error": "No data available for ranking"}

query = context.get("user\_query", "").lower()

is\_reverse = False

if any(word in query for word in ["top", "high", "highest"]):

is\_reverse = True

n = self.\_extract\_ranking\_n(query)

try:

clean\_data = data.dropna(subset=['value'])

sorted\_data = clean\_data.sort\_values('value', ascending=not is\_reverse)

ranked = sorted\_data.head(n)

return {

"type": "ranking",

"data": ranked.to\_dict(orient='records'),

"metadata": {

"sort\_order": "descending" if is\_reverse else "ascending",

"count": len(ranked)

}

}

except Exception as e:

return {

"error": f"Ranking failed: {str(e)}",

"original\_data": data.to\_dict(orient='records') if not data.empty else []

}

def \_handle\_threshold\_check(self, data: pd.DataFrame, context: Dict) -> Dict:

if data.empty:

return {"error": "No data available for threshold check"}

try:

query = context.get("user\_query", "").lower()

threshold = self.\_extract\_threshold(query)

above\_threshold = data[data['value'] > threshold]

below\_threshold = data[data['value'] <= threshold]

return {

"type": "threshold\_check",

"data": data.to\_dict(orient='records'),

"metadata": {

"threshold": threshold,

"count\_above": len(above\_threshold),

"count\_below": len(below\_threshold),

"count\_total": len(data)

}

}

except Exception as e:

return {

"error": f"Threshold check failed: {str(e)}",

"original\_data": data.to\_dict(orient='records') if not data.empty else []

}

def \_handle\_list\_entities(self, data: pd.DataFrame, context: Dict) -> Dict:

if data.empty:

return {"error": "No entities match the criteria"}

return {

"type": "entity\_list",

"data": data.to\_dict(orient='records'),

"metadata": {

"count": len(data),

"criteria": context.get("user\_query", "")

}

}

def \_handle\_trend\_analysis(self, data: pd.DataFrame, context: Dict) -> Dict:

if data.empty:

return {"error": "No data available for trend analysis"}

try:

sorted\_data = data.sort\_values('field\_value')

values = sorted\_data['value'].tolist()

trend = "increasing" if values[-1] > values[0] else "decreasing" if values[-1] < values[0] else "stable"

return {

"type": "trend\_analysis",

"data": data.to\_dict(orient='records'),

"metadata": {

"trend": trend,

"time\_period": context.get("timeframes", [{}])[0].get("period\_in\_query", "unknown")

}

}

except Exception as e:

return {

"error": f"Trend analysis failed: {str(e)}",

"original\_data": data.to\_dict(orient='records') if not data.empty else []

}

def \_handle\_summarize(self, data: pd.DataFrame, context: Dict) -> Dict:

if data.empty:

return {"error": "No data available to summarize"}

try:

avg = data['value'].mean()

count = len(data)

std\_dev = data['value'].std()

return {

"type": "summary",

"data": data.to\_dict(orient='records'),

"metadata": {

"average": avg,

"count": count,

"standard\_deviation": std\_dev,

"metric": data.iloc[0]["metric"] if not data.empty else "unknown"

}

}

except Exception as e:

return {

"error": f"Summary failed: {str(e)}",

"original\_data": data.to\_dict(orient='records') if not data.empty else []

}

def \_handle\_visualize(self, data: pd.DataFrame, context: Dict) -> Dict:

if data.empty:

return {"error": "No data available for visualization"}

try:

viz\_data = []

for \_, row in data.iterrows():

viz\_data.append({

"x": row["field\_value"],

"y": row["value"],

"label": f"{row['field\_value']}: {row['value']:.2f}",

"support": row.get("support\_numbers", "")

})

return {

"type": "visualization",

"data": viz\_data,

"metadata": {

"chart\_type": self.\_determine\_chart\_type(context),

"metric": data.iloc[0]["metric"] if not data.empty else "unknown"

}

}

except Exception as e:

return {

"error": f"Visualization failed: {str(e)}",

"original\_data": data.to\_dict(orient='records') if not data.empty else []

}

def \_handle\_predict(self, data: pd.DataFrame, context: Dict) -> Dict:

try:

entity\_field, entity\_value = self.\_get\_primary\_entity(context, default\_field="country\_code")

if not entity\_value:

return {"error": "No entity provided (e.g., country code). Please specify like 'BR'."}

ts = self.\_monthly\_efficiency\_ts(entity\_field, entity\_value)

if ts.empty or len(ts.dropna()) < 3:

return {"error": f"Not enough history to forecast for {entity\_field}={entity\_value}."}

try:

model = ARIMA(ts, order=(1, 1, 1), enforce\_stationarity=False, enforce\_invertibility=False)

fitted = model.fit()

except Exception as e:

return {"error": f"ARIMA fitting failed: {str(e)}"}

n\_steps = 2

fcast = fitted.get\_forecast(steps=n\_steps)

f\_vals = pd.Series(fcast.predicted\_mean, index=pd.date\_range(

start=ts.index[-1] + pd.offsets.MonthBegin(1),

periods=n\_steps, freq="MS"

))

f\_vals = f\_vals.clip(lower=0.0, upper=1.0)

hist\_tail = ts.tail(6).copy()

hist\_df = pd.DataFrame({

"month": [d.strftime("%b %Y") for d in hist\_tail.index],

"efficiency": (hist\_tail \* 100).round(2).tolist()

})

f\_df = pd.DataFrame({

"month": [d.strftime("%b %Y") for d in f\_vals.index],

"forecast\_efficiency": (f\_vals \* 100).round(2).tolist()

})

prompt = f"""

You are generating a short, precise business trend explanation for vehicle efficiency.

Context:

- Entity: {entity\_field} = {entity\_value}

- Historical monthly efficiency (last {len(hist\_df)} rows, in %):

{hist\_df.to\_string(index=False)}

- Forecast for next {len(f\_df)} months (in %, numeric and clipped to [0, 100]):

{f\_df.to\_string(index=False)}

Instructions:

1) In 3–5 crisp sentences, summarize the direction (rising/falling/stable), magnitude (approximate pp change),

and any month-over-month nuance visible.

2) Do NOT invent extra numbers. Only refer to the values above.

3) End with a one-sentence implication.

"""

llm\_resp = self.narrator\_llm.invoke(prompt)

narrative = (llm\_resp.content or "").strip()

result\_records = []

for m, v in f\_vals.items():

result\_records.append({

"field": entity\_field,

"field\_value": entity\_value,

"metric": "efficiency",

"period": m.strftime("%Y-%m"),

"value": float(v),

"pretty\_value": f"{v\*100:.2f}%",

})

return {

"type": "prediction",

"entity": {"field": entity\_field, "value": entity\_value},

"historical": hist\_df.to\_dict(orient="records"),

"forecast": result\_records,

"metadata": {

"model": "ARIMA(1,1,1)",

"horizon\_months": n\_steps

},

"narrative": narrative,

"data": [

{

"field": entity\_field,

"field\_value": f"{entity\_value} — {rec['period']}",

"metric": "efficiency",

"value": rec["value"],

"support\_numbers": ""

}

for rec in result\_records

]

}

except Exception as e:

return {"error": f"Prediction failed: {str(e)}"}

def \_handle\_diagnose(self, data: pd.DataFrame, context: Dict) -> Dict:

if data.empty:

return {"error": "No data available for diagnosis"}

try:

# ---- Pull computed ratio & supports from AnalysisAgent result ----

row = data.iloc[0]

metric\_label = row.get("metric", "metric")

metric\_value = float(row.get("value", float("nan")))

support\_str = row.get("support\_numbers", "")

# ---- Resolve entity (e.g., 'ID'); ----

entity\_value = None

entity\_field = "country\_code" # Default for country diagnosis

for e in context.get("entities", []):

if e.get("value") not in (None, "all"):

entity\_value = e["value"]

entity\_field = e.get("field", "country\_code")

break

if not entity\_value:

m = re.search(r"\b[A-Z]{2}\b", context.get("user\_query", ""))

if m:

entity\_value = m.group(0)

is\_percent = (

self.\_is\_percentage\_metric(metric\_label)

or self.\_is\_percentage\_metric(context.get("metrics", ["efficiency"])[0])

)

value\_str = f"{metric\_value:.2%}" if is\_percent else f"{metric\_value:,.2f}"

# ---- Parse numerator / denominator from support string ----

parsed = self.\_parse\_support\_numbers(support\_str) if hasattr(self, "\_parse\_support\_numbers") else {}

num\_key, den\_key = None, None

if "TKM" in parsed and "Total\_KM" in parsed:

num\_key, den\_key = "TKM", "Total\_KM"

elif len(parsed) >= 2:

keys = list(parsed.keys())

num\_key, den\_key = keys[0], keys[1]

numerator = parsed.get(num\_key) if num\_key else None

denominator = parsed.get(den\_key) if den\_key else None

# ---- Compute peer medians (exclude the current entity) ----

num\_med = den\_med = None

df\_ref = getattr(self, "\_raw\_df", None).copy()

if (

df\_ref is not None and not df\_ref.empty and entity\_value

and num\_key and den\_key

and all(col in df\_ref.columns for col in [entity\_field, num\_key, den\_key, "drive\_date"])

):

# Apply the SAME timeframe, but DO NOT apply entity filters

start\_ts, end\_ts = self.\_get\_time\_window\_from\_context(context)

df\_ref["drive\_date"] = pd.to\_datetime(df\_ref["drive\_date"])

df\_time = df\_ref[(df\_ref["drive\_date"] >= start\_ts) & (df\_ref["drive\_date"] <= end\_ts)].copy()

# Aggregate per entity over the timeframe

peer\_agg = (

df\_time.groupby(entity\_field)

.agg({num\_key: "sum", den\_key: "sum"})

.reset\_index()

)

# Exclude current entity and compute medians

peers = peer\_agg[peer\_agg[entity\_field] != entity\_value]

if not peers.empty:

num\_med = float(peers[num\_key].median())

den\_med = float(peers[den\_key].median())

# ---- Get trend data for recent months ----

trend\_data = self.\_get\_recent\_trend\_data(entity\_field, entity\_value, context)

# ---- Enhanced: Gather additional metrics for LLM analysis ----

additional\_metrics = self.\_gather\_additional\_metrics(entity\_field, entity\_value, context)

# ---- Enhanced: Use LLM for comprehensive analysis ----

diagnosis\_prompt = self.\_build\_diagnosis\_prompt(

basic\_metrics={

"efficiency": metric\_value \* 100, # Convert to percentage

"TKM": numerator,

"Total\_KM": denominator,

"peer\_median\_TKM": num\_med,

"peer\_median\_Total\_KM": den\_med

},

additional\_metrics=additional\_metrics,

entity\_value=entity\_value,

trend\_data=trend\_data

)

# Get LLM analysis

llm\_response = self.narrator\_llm.invoke(diagnosis\_prompt)

narrative = llm\_response.content.strip()

# ---- Enhanced supporting numbers with precise metrics ----

supporting\_numbers = [

f"Efficiency: {value\_str}",

f"TKM: {numerator:,.0f}" if numerator else "",

f"Total\_KM: {denominator:,.0f}" if denominator else "",

]

# Add peer comparisons if available

if num\_med:

supporting\_numbers.append(f"Peer Median TKM: {num\_med:,.0f}")

if den\_med:

supporting\_numbers.append(f"Peer Median Total\_KM: {den\_med:,.0f}")

# Add route composition percentages

c1\_km = additional\_metrics.get('C1\_kms\_Collected', 0)

c2\_km = additional\_metrics.get('C2\_kms\_Collected', 0)

c3\_km = additional\_metrics.get('C3\_kms\_Collected', 0)

total\_km\_context = c1\_km + c2\_km + c3\_km

if total\_km\_context > 0:

c1\_pct = (c1\_km / total\_km\_context) \* 100

c2\_pct = (c2\_km / total\_km\_context) \* 100

c3\_pct = (c3\_km / total\_km\_context) \* 100

supporting\_numbers.append(f"Route Composition: C1: {c1\_pct:.1f}% / C2: {c2\_pct:.1f}% / C3: {c3\_pct:.1f}%")

# Add driver performance metrics

if 'driver\_performance\_spread' in additional\_metrics:

supporting\_numbers.append(f"Driver Performance Spread: {additional\_metrics['driver\_performance\_spread']:.1f}pp")

if 'top\_driver\_efficiency' in additional\_metrics:

supporting\_numbers.append(f"Top Driver Efficiency: {additional\_metrics['top\_driver\_efficiency']:.1f}%")

if 'bottom\_driver\_efficiency' in additional\_metrics:

supporting\_numbers.append(f"Bottom Driver Efficiency: {additional\_metrics['bottom\_driver\_efficiency']:.1f}%")

# Add SOH

if 'SOH' in additional\_metrics:

supporting\_numbers.append(f"SOH: {additional\_metrics['SOH']:,.0f} hours")

# Add trend information

if trend\_data and len(trend\_data) >= 2:

values = [d['efficiency'] for d in trend\_data]

months = [d['month'] for d in trend\_data]

trend\_str = " → ".join(f"{v:.1f}%" for v in values)

trend\_change = values[-1] - values[0]

supporting\_numbers.append(f"Trend: {trend\_str} ({trend\_change:+.1f}pp change)")

supporting\_numbers = [s for s in supporting\_numbers if s]

return {

"type": "diagnosis",

"data": data.to\_dict(orient="records"),

"metadata": {

"metric": metric\_label,

"entity": entity\_value,

"numerator\_key": num\_key,

"denominator\_key": den\_key,

"numerator": numerator,

"denominator": denominator,

"peer\_num\_median": num\_med,

"peer\_den\_median": den\_med,

},

"narrative": narrative,

"supporting\_numbers": supporting\_numbers,

"trend\_data": trend\_data

}

except Exception as e:

return {

"error": f"Diagnosis failed: {str(e)}",

"original\_data": data.to\_dict(orient="records") if not data.empty else [],

}

def \_gather\_additional\_metrics(self, entity\_field: str, entity\_value: str, context: Dict) -> Dict:

"""Gather C1/C2/C3, SOH, and driver metrics from filtered dataframe"""

additional\_metrics = {}

try:

df = getattr(self, "\_working\_df", None)

if df is None or df.empty:

return additional\_metrics

temp = df.copy()

if entity\_value and entity\_field in temp.columns:

temp = temp[temp[entity\_field] == entity\_value]

# Use the correct column names from semantic layer

for chain\_col in ['C1\_kms\_Collected', 'C2\_kms\_Collected', 'C3\_kms\_Collected']:

if chain\_col in temp.columns:

additional\_metrics[chain\_col] = temp[chain\_col].sum()

if 'SOH' in temp.columns:

additional\_metrics['SOH'] = temp['SOH'].sum()

if 'user\_email' in temp.columns and 'efficiency' in temp.columns:

driver\_stats = temp.groupby('user\_email')['efficiency'].agg(['mean', 'std']).reset\_index()

if not driver\_stats.empty and len(driver\_stats) > 1:

additional\_metrics['driver\_performance\_spread'] = driver\_stats['std'].mean()

additional\_metrics['top\_driver\_efficiency'] = driver\_stats['mean'].max()

additional\_metrics['bottom\_driver\_efficiency'] = driver\_stats['mean'].min()

except Exception as e:

print(f"Warning: Could not gather additional metrics: {e}")

return additional\_metrics

def \_build\_diagnosis\_prompt(self, basic\_metrics: Dict, additional\_metrics: Dict,

entity\_value: str, trend\_data: List[Dict]) -> str:

"""Build prompt for precise, data-driven diagnosis with narrative structure"""

# Calculate percentage differences for peer comparison

tkm\_pct\_diff = self.\_calculate\_percentage\_diff(basic\_metrics.get('TKM', 0), basic\_metrics.get('peer\_median\_TKM', 0))

total\_km\_pct\_diff = self.\_calculate\_percentage\_diff(basic\_metrics.get('Total\_KM', 0), basic\_metrics.get('peer\_median\_Total\_KM', 0))

prompt = f"""

Generate a data-driven operational efficiency diagnosis for {entity\_value} using EXACT numbers from the provided metrics.

\*\*CORE METRICS:\*\*

- Current Efficiency: {basic\_metrics.get('efficiency', 0):.2f}%

- TKM: {basic\_metrics.get('TKM', 0):,.0f} ({tkm\_pct\_diff} below peer median)

- Total\_KM: {basic\_metrics.get('Total\_KM', 0):,.0f} ({total\_km\_pct\_diff} below peer median)

\*\*CONTEXT DATA:\*\*"""

# Add topography metrics

c1\_km = additional\_metrics.get('C1\_kms\_Collected', 0)

c2\_km = additional\_metrics.get('C2\_kms\_Collected', 0)

c3\_km = additional\_metrics.get('C3\_kms\_Collected', 0)

total\_km\_context = c1\_km + c2\_km + c3\_km

if total\_km\_context > 0:

c1\_pct = (c1\_km / total\_km\_context) \* 100

c2\_pct = (c2\_km / total\_km\_context) \* 100

c3\_pct = (c3\_km / total\_km\_context) \* 100

prompt += f"\n- Route Composition: C1: {c1\_pct:.1f}% / C2: {c2\_pct:.1f}% / C3: {c3\_pct:.1f}%"

# Add driver metrics

if 'driver\_performance\_spread' in additional\_metrics:

prompt += f"\n- Driver Performance Spread: {additional\_metrics['driver\_performance\_spread']:.1f}pp"

if 'top\_driver\_efficiency' in additional\_metrics and 'bottom\_driver\_efficiency' in additional\_metrics:

prompt += f" (Top: {additional\_metrics['top\_driver\_efficiency']:.1f}%, Bottom: {additional\_metrics['bottom\_driver\_efficiency']:.1f}%)"

# Add SOH

if 'SOH' in additional\_metrics:

soh = additional\_metrics['SOH']

prompt += f"\n- SOH: {soh:,.0f} hours"

# Add trend

if trend\_data and len(trend\_data) >= 2:

values = [d['efficiency'] for d in trend\_data]

months = [d['month'] for d in trend\_data]

trend\_str = " → ".join(f"{v:.1f}%" for v in values)

trend\_change = values[-1] - values[0]

prompt += f"\n- Trend: {trend\_str} ({trend\_change:+.1f}pp over {len(values)} months)"

prompt += f"""

\*\*CRITICAL INSTRUCTIONS:\*\*

1. Use ONLY the exact numbers provided above - no estimations or assumptions

2. ABSOLUTELY NO vague language: no "may", "could", "suggest", "potential", "might"

3. Structure the analysis in this exact format:

\*\*Current Standing:\*\*

[Entity]'s operational efficiency stands at [X]%, placing it [Y] percentage points below the regional average of [Z]%.

\*\*Recent Performance:\*\*

Efficiency trends show [Month1]: [A]% → [Month2]: [B]% → [Month3]: [C]%, representing a [±X] percentage point change over the period.

\*\*Root Cause Analysis:\*\*

- \*\*Operational Effectiveness:\*\* [Driver spread: Xpp] combined with [SOH utilization context] contributes [Y] percentage points to the efficiency gap

- \*\*Route Composition:\*\* [C1: A%] / [C2: B%] / [C3: C%] distribution creates [Z] percentage point efficiency impact

- \*\*Scale Deficiency:\*\* TKM at [Number] ([X]% below peer median) and Total\_KM at [Number] ([Y]% below median) account for [Z] percentage points of the performance gap

\*\*Performance Decomposition:\*\*

- Operational factors: [X]pp impact ([Y]% of total gap)

- Route mix: [X]pp impact ([Y]% of total gap)

- Market scale: [X]pp impact ([Y]% of total gap)

\*\*Conclusion:\*\*

[Primary factor] presents the primary opportunity, addressing [X] percentage points of the efficiency gap. [Secondary factor] offers additional potential for [Y] points. Based on peer performance benchmarks, achieving [Z]% efficiency represents a realistic target.

4. Use only factual, quantitative statements based on the provided numbers

5. For peer comparisons, use relative percentage point differences, not vague descriptions

6. All impacts must be quantified in percentage points with exact numbers

"""

return prompt

def \_calculate\_percentage\_diff(self, value: float, median: float) -> str:

"""Calculate percentage difference for prompt"""

if median == 0 or value is None:

return "N/A"

pct\_diff = ((median - value) / median) \* 100

direction = "above" if pct\_diff < 0 else "below"

return f"{abs(pct\_diff):.0f}% {direction}"

def \_get\_recent\_trend\_data(self, entity\_field: str, entity\_value: str, context: Dict) -> List[Dict]:

"""Get efficiency trend data for the last 3 months"""

try:

df = getattr(self, "\_working\_df", None)

if df is None or df.empty or "drive\_month" not in df.columns:

return []

temp = df.copy()

if entity\_value and entity\_field in temp.columns:

temp = temp[temp[entity\_field] == entity\_value]

# Ensure drive\_month is properly formatted

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

recent\_months = temp["drive\_month"].dropna().unique()

recent\_months.sort()

recent\_months = recent\_months[-3:] # Get last 3 months

trend\_data = []

for month in recent\_months:

month\_data = temp[temp["drive\_month"] == month]

if not month\_data.empty:

tkm\_sum = month\_data["TKM"].sum()

total\_km\_sum = month\_data["Total\_KM"].sum()

efficiency = (tkm\_sum / total\_km\_sum) \* 100 if total\_km\_sum != 0 else 0

trend\_data.append({

"month": month.strftime("%b %Y"),

"efficiency": efficiency,

"TKM": tkm\_sum,

"Total\_KM": total\_km\_sum

})

return trend\_data

except Exception as e:

print(f"Error getting trend data: {e}")

return []

def \_default\_handler(self, data: pd.DataFrame, context: Dict) -> Dict:

return {

"type": "generic\_results",

"data": data.to\_dict(orient='records') if not data.empty else [],

"support\_numbers": data.iloc[0].get("support\_numbers", "") if not data.empty else ""

}

# --- Helper Methods ---

def \_get\_primary\_entity(self, context: Dict, default\_field: str = "country\_code") -> tuple:

entity\_field = default\_field

entity\_value = None

try:

if isinstance(context, dict) and "query\_analysis" in context:

q = context["query\_analysis"][0] if context.get("query\_analysis") else {}

else:

q = context or {}

entities = [

e for e in q.get("entities", [])

if isinstance(e, dict)

and e.get("value") not in (None, "all")

and e.get("condition\_type") != "numeric\_filter"

]

if entities:

entity\_field = entities[0].get("field", default\_field)

entity\_value = entities[0].get("value")

if not entity\_value:

m = re.search(r"\b[A-Z]{2}\b", q.get("user\_query", ""))

if m:

entity\_field = default\_field

entity\_value = m.group(0)

except Exception as e:

logging.error(f"\_get\_primary\_entity error: {e}")

return entity\_field, entity\_value

def \_monthly\_efficiency\_ts(self, entity\_field: str, entity\_value: str) -> pd.Series:

df = getattr(self, "\_working\_df", None)

if df is None or df.empty:

return pd.Series(dtype=float)

temp = df.copy()

if entity\_field in temp.columns and entity\_value:

temp = temp[temp[entity\_field] == entity\_value]

required\_cols = {"drive\_month", "TKM", "Total\_KM"}

if not required\_cols.issubset(set(temp.columns)):

return pd.Series(dtype=float)

ts = (

temp.dropna(subset=["drive\_month"])

.groupby("drive\_month", as\_index=True)

.agg(TKM=("TKM", "sum"), Total\_KM=("Total\_KM", "sum"))

)

if ts.empty:

return pd.Series(dtype=float)

ts["efficiency"] = ts["TKM"] / ts["Total\_KM"]

ts = ts["efficiency"].dropna().sort\_index()

return ts

def \_get\_time\_window\_from\_context(self, context: Dict) -> tuple:

try:

q = context.get("query\_analysis", [])[0]

tfs = q.get("timeframes", [])

if tfs:

start = pd.to\_datetime(tfs[0].get("start"))

end = pd.to\_datetime(tfs[0].get("end"))

return start, end

except Exception:

pass

return pd.to\_datetime("1970-01-01"), pd.Timestamp.now().tz\_localize(None)

def \_parse\_support\_numbers(self, s: str) -> dict:

if not s:

return {}

pairs = re.findall(r'([A-Za-z\_]+)\s\*:\s\*([\-]?\d+(?:\.\d+)?)', s)

return {k: float(v) for k, v in pairs}

def \_is\_percentage\_metric(self, metric\_name: str) -> bool:

pct\_metrics = {"efficiency", "c1\_km\_perc"}

return metric\_name.lower() in pct\_metrics

def \_extract\_ranking\_n(self, query: str) -> int:

q = (query or "").lower()

m = re.search(r'(?:top|bottom|lowest|highest)\s+(\d+)|(\d+)\s+(?:lowest|highest)', q)

if m:

return int(m.group(1) or m.group(2))

if any(w in q for w in ["highest", "lowest"]):

return 1

return 5

def \_extract\_threshold(self, query: str) -> float:

match = re.search(

r'(?:above|below|>|<|>=|<=|==)\s\*(\d+(?:\.\d+)?)',

query.lower()

)

return float(match.group(1)) if match else 0.0

def \_determine\_chart\_type(self, context: Dict) -> str:

intent = context.get("intent", "")

timeframes = context.get("timeframes", [])

if intent == IntentType.TREND\_ANALYSIS.value and timeframes:

return "line\_chart"

elif intent == IntentType.COMPARE\_METRIC.value:

return "bar\_chart"

elif intent == IntentType.RANK\_ENTITIES.value:

return "horizontal\_bar\_chart"

else:

return "bar\_chart"

def \_generate\_conversational\_report(self, state: AgentState) -> str:

if not state.analysis\_results:

return "No results found for your query."

qa\_list = (state.extracted\_context or {}).get("query\_analysis") or []

if not qa\_list:

return "No query analysis available."

query\_ctx = qa\_list[0]

intent = query\_ctx["intent"]

primary\_result = None

if IntentType.DIAGNOSE\_METRIC.value in state.analysis\_results:

primary\_result = state.analysis\_results[IntentType.DIAGNOSE\_METRIC.value]

else:

for k in state.analysis\_results:

primary\_result = state.analysis\_results[k]

break

if not primary\_result or "error" in primary\_result:

return primary\_result.get("error", "No results found for your query.")

report\_parts = []

if intent == IntentType.DIAGNOSE\_METRIC.value:

if "narrative" in primary\_result:

report\_parts.append(primary\_result["narrative"])

if primary\_result.get("supporting\_numbers"):

report\_parts.append("\n\*\*Supporting Numbers:\*\*")

for num in primary\_result["supporting\_numbers"]:

report\_parts.append(f"- {num}")

else:

results\_data = primary\_result.get("data", [])

if results\_data:

response = self.\_generate\_conversational\_response(query\_ctx, results\_data)

report\_parts.append(response)

table = self.\_generate\_results\_table(query\_ctx, results\_data)

if table:

report\_parts.append("\n" + table)

suggestions = self.\_generate\_suggested\_queries(state.extracted\_context)[:3]

if suggestions:

report\_parts.append("\n\*\*Suggested follow-up queries:\*\*")

report\_parts.extend([f"{i}. {q}" for i, q in enumerate(suggestions, 1)])

return "\n".join(report\_parts)

def \_generate\_conversational\_response(self, query\_ctx: Dict, results\_data: List[Dict]) -> str:

intent = query\_ctx["intent"]

metric\_name = self.\_get\_friendly\_name(query\_ctx["metrics"][0])

if intent == IntentType.RANK\_ENTITIES.value:

rows = [r for r in results\_data if pd.notnull(r.get("value"))]

if not rows:

return "No results found."

q = (query\_ctx.get("user\_query") or "").lower()

want\_highest = any(w in q for w in ["top", "high", "highest"])

best = max(rows, key=lambda r: r["value"]) if want\_highest else min(rows, key=lambda r: r["value"])

metric\_name = self.\_get\_friendly\_name(query\_ctx["metrics"][0]) if query\_ctx.get("metrics") else "Metric"

ent\_label = self.\_get\_friendly\_name(best.get("field", "entity"))

ent\_value = str(best.get("field\_value"))

is\_percent = self.\_is\_percentage\_metric(query\_ctx["metrics"][0]) if query\_ctx.get("metrics") else True

fmt = (lambda v: f"{v:.2%}") if is\_percent else (lambda v: f"{v:,.2f}")

if want\_highest:

return f"{ent\_value} has the highest {metric\_name} at {fmt(best['value'])}."

else:

return f"{ent\_value} has the lowest {metric\_name} at {fmt(best['value'])}."

if intent == IntentType.TREND\_ANALYSIS.value:

try:

return self.\_format\_mom\_trend(query\_ctx, results\_data)

except Exception as e:

print(f"Error in \_format\_mom\_trend: {e}")

if intent == IntentType.COMPARE\_METRIC.value:

rows = [r for r in results\_data if pd.notnull(r.get("value"))]

if len(rows) >= 2:

rows = sorted(rows, key=lambda x: x["value"], reverse=True)

metric\_name = self.\_get\_friendly\_name(query\_ctx["metrics"][0]) if query\_ctx.get("metrics") else "Metric"

is\_percent = self.\_is\_percentage\_metric(query\_ctx["metrics"][0]) if query\_ctx.get("metrics") else True

fmt = (lambda v: f"{v:.2%}") if is\_percent else (lambda v: f"{v:,.2f}")

winner = rows[0]

winner\_name = str(winner["field\_value"])

winner\_val = fmt(winner["value"])

names\_in\_results = [str(r["field\_value"]) for r in rows]

if len(rows) == 2:

loser = rows[1]

return f"{winner\_name} has higher {metric\_name} ({winner\_val}) than {loser['field\_value']} ({fmt(loser['value'])})."

else:

among = ", ".join(names\_in\_results)

return f"{winner\_name} has the highest {metric\_name} at {winner\_val} among {among}."

entities = query\_ctx.get("entities", [])

metric\_value = results\_data[0].get("value", 0)

is\_percent = self.\_is\_percentage\_metric(query\_ctx["metrics"][0])

value\_str = f"{metric\_value:.2%}" if is\_percent else f"{metric\_value:,.2f}"

if entities and entities[0]["value"] != "all":

entity\_value = entities[0]["value"]

return f"The {metric\_name} for {entity\_value} is {value\_str}"

else:

return f"The {metric\_name} is {value\_str}"

def \_generate\_results\_table(self, query\_ctx: Dict, results\_data: List[Dict]) -> str:

if not results\_data:

return ""

entity\_field = query\_ctx.get("entities", [{}])[0].get("field", "entity")

entity\_name = self.\_get\_friendly\_name(entity\_field)

if query\_ctx.get("intent") == IntentType.TREND\_ANALYSIS.value:

entity\_name = "Month"

try:

tmp = pd.DataFrame(results\_data).copy()

tmp["\_m"] = pd.to\_datetime(tmp["field\_value"].ast)

tmp["\_m"] = pd.to\_datetime(tmp["field\_value"].astype(str).str[:7], errors="coerce")

results\_data = tmp.sort\_values("\_m").drop(columns=["\_m"]).to\_dict(orient="records")

except Exception:

pass

distinct\_vals = {str(item.get("field\_value")) for item in results\_data if "field\_value" in item}

multi\_entity = (query\_ctx.get("intent") == IntentType.COMPARE\_METRIC.value) or (len(distinct\_vals) > 1)

rows = []

for item in results\_data:

row\_entity\_value = item.get("field\_value", "N/A") if multi\_entity else (

query\_ctx["entities"][0]["value"]

if query\_ctx.get("entities") and query\_ctx["entities"][0]["value"] != "all"

else item.get("field\_value", "N/A")

)

metric\_str = (

f"{item.get('value', 0):.2%}"

if self.\_is\_percentage\_metric(item.get("metric", ""))

else f"{item.get('value', 0):.2f}"

)

rows.append({entity\_name: row\_entity\_value,

"Metric": metric\_str,

"Supporting Numbers": item.get("support\_numbers", "")})

df = pd.DataFrame(rows)

return "### Detailed Results\n" + df.to\_markdown(index=False, tablefmt="grid")

def \_generate\_suggested\_queries(self, context: Dict) -> List[str]:

"""Generate context-aware suggested queries"""

query\_ctx = context["query\_analysis"][0]

intent = query\_ctx["intent"]

metric = query\_ctx["metrics"][0]

entities = query\_ctx.get("entities", [])

suggestions = []

# Base suggestions for all intents

if entities and entities[0]["value"] != "all":

entity\_value = entities[0]["value"]

suggestions.extend([

f"Show trend of {metric} for {entity\_value} over time",

f"Compare {entity\_value} with similar entities",

f"Breakdown components of {metric} for {entity\_value}"

])

else:

suggestions.extend([

f"Show top 10 entities by {metric}",

f"Compare average {metric} by region",

f"Analyze trend of {metric} over time"

])

# Intent-specific additions

if intent == IntentType.COMPARE\_METRIC.value:

suggestions.append("Show comparison breakdown by component")

return suggestions[:3] # Return only top 3 most relevant

def \_get\_friendly\_name(self, technical\_name: str) -> str:

"""Convert technical names to friendly display names"""

name\_map = {

"country\_code": "Country",

"region": "Region",

"vehicle\_id": "Vehicle",

"efficiency": "Efficiency",

"TKM": "Target Kilometers",

"Total\_KM": "Total Kilometers",

"all": "All"

}

return name\_map.get(technical\_name, technical\_name.replace("\_", " ").title())

def \_format\_mom\_trend(self, query\_ctx: Dict, results\_data: List[Dict]) -> str:

# keep only rows with a value

rows = [r for r in results\_data if pd.notnull(r.get("value"))]

# sort by month-like field\_value (e.g., '2025-01', '2025-02')

def \_parse\_month(s: str):

try:

# works for 'YYYY-MM' and for 'YYYY-MM-DD'; falls back gracefully

return pd.to\_datetime(str(s)[:7], errors="coerce")

except Exception:

return pd.NaT

rows = sorted(rows, key=lambda r: \_parse\_month(r.get("field\_value", "")))

# format values as % (efficiency is percentage)

is\_percent = self.\_is\_percentage\_metric(query\_ctx["metrics"][0]) if query\_ctx.get("metrics") else True

fmt = (lambda v: f"{v:.2%}") if is\_percent else (lambda v: f"{v:,.2f}")

series = " \u2192 ".join(fmt(r["value"]) for r in rows) # \u2192 is → arrow

# friendly header

metric\_name = self.\_get\_friendly\_name(query\_ctx["metrics"][0]) if query\_ctx.get("metrics") else "Metric"

entity\_value = next((e.get("value") for e in (query\_ctx.get("entities") or []) if e.get("value") not in (None, "all")), None)

if entity\_value:

return f"The {metric\_name} trend for {entity\_value} is {series}."

return f"The {metric\_name} trend is {series}."

"""### Main function"""

def main():

# Configuration

os.environ["OPENAI\_API\_KEY"] = "sk-proj-Ck4zxDJufOmdWt5J1rdM03QKKLfy0DicPIFjdlRS8FjU\_GdYPOgGXPurr\_DhlNuMNjWbDTW1YAT3BlbkFJyy5GLDE7cLELkX1mVOa3gQX80ritwaIJQHPKx3kLtPQygTiAMoOTY-M7F6e8yqjVMxdS-T47oA"

gsheet\_url = "https://docs.google.com/spreadsheets/d/1zsMj2wX84cQW7hwRyoBquS8uzHn6qmKUBMB9OGXFZGg/edit?gid=0#gid=0"

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

# Initialize orchestrator

orchestrator = OrchestratorAgent(gsheet\_url, semantic\_layer\_path, os.environ["OPENAI\_API\_KEY"])

query = "List bottom 5 drivers with low TKM"#"Show the trend for efficiency in country BR for the past 3 months" #"which country has better efficiency country ID or BR ?"#" why does ID have low efficiency?" #"why does ID have low efficiency"

# which country has better efficiency country ID or BR ?

# what is the TKM for country TW ?

# List the bottom 5 countires with low efficiency ?

# Why is the efficiency low for BR ?

# Show the trendd for BR next 2 months

# predict efficiency for next two months for BR

# Presentable summary with supporting numbers or table / plot

state = AgentState(user\_query=query)

result = orchestrator.process(state)

print("\nFinal Report:")

print(result.report)

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

main()

# which country has highest efficiency among the countries BR or TW or ID?

#Give bottom 5 drivers with lowest efficiency in country BR and TW

#List bottom 3 countries with lowest efficiency

"""# New Section

##archive

"""