# -\*- coding: utf-8 -\*-

"""code\_buildup.ipynb

Automatically generated by Colab.

Original file is located at

https://colab.research.google.com/drive/1rUQdik0xoz5vBoUYLD-zjlb3JWJNnPci

"""

! 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 # Updated imports

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

# Authenticate and authorize (unchanged)

auth.authenticate\_user()

from google.auth import default

creds, \_ = default()

gc = gspread.authorize(creds)

# Logging (unchanged)

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) # 03:11 PM IST, August 04, 2025

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"]):

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"]):

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 intents:

return intents

# Use LLM for complex intent extraction

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()

# Explicit mentions in query

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

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

return True

# Special cases

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()

# Mapping of query terms to possible columns

term\_mapping = {

'vehicle': 'vehicle\_id',

'country': 'country\_code',

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

'driver': 'user\_email',

'vid': 'vehicle\_id'

}

# Check for explicit mentions

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

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

return column

# Fallback logic

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]:

"""Extract entities from query with support for ranking targets and numeric filters"""

query\_lower = query.lower()

is\_ranking = "top" in query\_lower or "bottom" in query\_lower

# Identify ranking target before processing other entities

ranking\_target = None

if is\_ranking:

ranking\_target = self.\_detect\_ranking\_entity(query, self.semantic\_layer["tables"]["efficiency\_table"]["columns"].keys())

# Get all numeric columns from semantic layer

numeric\_columns = [

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

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

]

# Extract numeric conditions

numeric\_conditions = []

for col in numeric\_columns:

# Match patterns like "TKM > 100", "Total\_KM >= 50"

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

matches = re.finditer(pattern, query, re.IGNORECASE)

for match in matches:

operator = match.group(1)

value = float(match.group(2))

numeric\_conditions.append({

"field": col,

"operator": operator,

"value": value,

"condition\_type": "numeric\_filter"

})

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

for metric in metrics:

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

prompt = f"""

Analyze the query and extract the following with high precision:

1. Metrics (e.g., efficiency, c1\_km\_perc) - identify all mentioned metrics

2. Entities - extract with strict typing rules:

- words like countries, vehicles, drivers, user emails, regions, seasons to be correctly mapped to entity.

- Country codes: Any 2-letter uppercase code (like BR, TW, ID) → country\_code

- Full country names → country\_code (map to standard codes)

- Vehicle IDs: Only numeric values with "five" digits → vehicle\_id

- EXCEPTION: In ranking queries (top/bottom N), use 'all' for vehicle\_id

- Regions: Only if explicitly mentioned as "region" or matching region names

3. Numeric conditions:

- Extract conditions like "TKM > 100" as numeric filters

4. Strict mapping rules:

- Never map 2-letter codes to vehicle\_id

- Never map numeric values to country\_code

- TW, BR, ID etc are always country\_code

- Numbers > 10000 are always vehicle\_id

5. Time handling:

- Month+year combinations (Jan 2025) → both drive\_date (2025-01-01) and drive\_month (2025-01)

- Relative dates (last month) → calculate concrete dates

6. Special cases:

- season → season\_name

- hub → driving\_hub\_id

- driver → user\_email

- vid -> vehicle\_id

7. Always keep vehicle\_id as string value.

8. Special handling for comparison queries:

- When seeing "A vs B" or "A or B", extract both as separate entities

- For "which has better X A or B", extract A and B as entities

- who has higher X "A", "B" or "C", extract A, B, C as entities

Return JSON: {{"metrics": [<metric>], "entities": [{{"field": <data\_field>, "value": <value>, "condition\_type": <optional>}}, ...]}}

Current semantic map for reference: {json.dumps(semantic\_map)}

Query: "{query}"

"""

response = self.llm.invoke(prompt)

cleaned\_response = response.content.strip().replace("```json","").replace("```","").strip()

try:

result = json.loads(cleaned\_response)

if not result.get("metrics"):

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

if "entities" not in result:

result["entities"] = []

if "metrics" not in result:

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

if "entities" not in result:

result["entities"] = []

if is\_ranking and ranking\_target:

# Replace any existing entity for the ranking target

result["entities"] = [e for e in result["entities"] if e.get("field") != ranking\_target]

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

# Add the numeric conditions we extracted via regex

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

return result

except json.JSONDecodeError:

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

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

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()

# order matters where synonyms overlap

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"

# default for questions like "what is the <metric> ..."

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)

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

# Initialize allowed fields with standard entities

self.allowed\_fields\_for\_filtering = [

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

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

]

# Add numeric columns from semantic layer

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

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

"""Add numeric columns to allowed fields for filtering"""

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):

"""Enforce data types based on semantic layer"""

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:

"""Fetch and filter data based on extracted context"""

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()

# Enforce data types

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

# Add date-related columns

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'])

# Process entities and conditions

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")

# Handle both regular entities and numeric conditions

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")

# Build filter conditions

entity\_conditions\_by\_field = {}

for entity in entities\_to\_process:

field = entity["field"]

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

# Handle numeric conditions (TKM > 100)

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':

# Skip 'all' values (used in ranking)

continue

else:

# Handle regular entity filters

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)

# Combine conditions with OR for same field, AND between different fields

entity\_combined\_conditions = []

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

if conditions\_list:

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

# Apply entity filters

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'])

# Initialize intent column

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

# Apply timeframe filters

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:

# For forecasting we NEED history; ignore future windows.

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}")

# Tag rows with intents based on timeframes

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)}")

# Fallback: tag all rows with intent

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), # sample std

}

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

"""Parse metric formula into numerator and denominator components"""

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 at appropriate granularity for each intent"""

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"]

# Filter data for this intent's timeframe

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)]

# Process each metric for this intent

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 metric (e.g., efficiency) – keep your existing logic

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:

# per-entity calc (sum(num)/sum(den))

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}")

# ✅ RAW COLUMN (e.g., Total\_KM, TKM, etc.) – apply aggregation

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:

"""Main processing pipeline"""

try:

# Step 1: Query Understanding

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

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

print(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")

print(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"] # fallback

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

state.data,

metrics=all\_metrics,

context=state.extracted\_context

)

state.analysis\_results = {}

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

# state.data,

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

# context=state.extracted\_context

# )

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

print(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:

"""Handle simple metric fetch requests"""

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:

"""Handle comparison between entities/metrics"""

if data.empty:

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

try:

# Get the entities being compared from the context

entities = [e for e in context.get("entities", []) if e.get("value") != "all"]

if len(entities) < 2:

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

# Filter data for the specific entities being compared

compare\_data = []

for entity in entities[:2]: # Compare first two entities

field, value = entity["field"], entity["value"]

matched = [d for d in data.to\_dict('records') if str(d.get("field\_value")).upper() == str(value).upper()]

if matched:

compare\_data.extend(matched)

if len(compare\_data) < 2:

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

return {

"type": "comparison",

"data": compare\_data,

"metadata": {

"entities": entities[:2],

"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:

"""Handle ranking requests (top/bottom N) with proper sorting"""

if data.empty:

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

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

is\_reverse = False # Default to ascending (for bottom/low)

# Determine sort direction

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

is\_reverse = True

# Extract N value (default to 5 if not specified)

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

try:

# Clean and sort data

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:

"""Handle threshold checks"""

if data.empty:

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

try:

# Extract threshold from query

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

threshold = self.\_extract\_threshold(query)

# Count items above/below threshold

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:

"""Handle listing entities by criteria"""

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:

"""Handle trend analysis requests"""

if data.empty:

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

try:

# Simple trend calculation

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:

"""Handle summary requests"""

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:

"""Generate visualization data"""

if data.empty:

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

try:

# Prepare data for visualization

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:

# """Handle prediction requests"""

# if data.empty:

# return {"error": "No data available for prediction"}

# try:

# # Simple prediction logic

# last\_value = data.iloc[-1]["value"]

# prediction = last\_value \* 1.05 # Simple 5% increase

# return {

# "type": "prediction",

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

# "metadata": {

# "predicted\_value": prediction,

# "confidence": 0.8, # Placeholder

# "time\_period": "next period",

# "method": "Simple projection (5% increase)"

# }

# }

# except Exception as e:

# return {

# "error": f"Prediction 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:

"""

Hybrid forecast:

- Build monthly efficiency ts for requested entity (e.g., country\_code='BR')

- Forecast next 2 months via ARIMA(1,1,1)

- Use LLM to generate a concise human narrative

"""

print("in \_handle\_predict")

try:

print("in \_handle\_predict 2")

# 1) Identify entity (default dimension = country\_code)

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'."}

print(f"entity\_field: {entity\_field} , {entity\_value}")

# 2) Build monthly efficiency ts for that entity

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}."}

print("in \_monthly\_efficiency\_ts")

# 3) Fit ARIMA (robust defaults)

# - enforce\_stationarity=False avoids convergence errors on many small business series

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)}"}

print("in ARIMA")

# 4) Forecast next 2 months

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"

))

# Bound to [0,1] because it's a ratio; clip to avoid silly values

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

# 5) Package historical tail for context (last 6 months)

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()

})

# 6) Ask LLM for a concise narrative (no hallucinated numbers:

# we pass numbers explicitly and instruct not to invent).

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 (e.g., “Expect slight improvement if TKM continues catching up to Total\_KM.”).

"""

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

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

# 7) Return structured result

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), # 0..1

"pretty\_value": f"{v\*100:.2f}%", # formatted for display

})

meh = {"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,

# add a simple tabular view so the report doesn't say "No matching data found."

"data": [

{

"field": entity\_field,

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

"metric": "efficiency",

"value": rec["value"], # 0..1 ratio

"support\_numbers": ""

}

for rec in result\_records

]}

print(f"final return from handle {meh}")

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,

# add a simple tabular view so the report doesn't say "No matching data found."

"data": [

{

"field": entity\_field,

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

"metric": "efficiency",

"value": rec["value"], # 0..1 ratio

"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:

"""Diagnosis with numerator/denominator, peer medians, and trend context."""

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., 'BR'); keep robust fallback ----

entity\_value = None

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

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

entity\_value = e["value"]

break

if not entity\_value:

# Fallback: try to recover a 2-letter uppercase code from the raw query

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

if m:

entity\_value = m.group(0)

# ---- Format the metric value ----

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

entity\_field = "country\_code" # adjust if diagnosing other entity types later

df\_ref = getattr(self, "\_raw\_df", None).copy() # <-- unfiltered copy

print(f'def\_ref: {df\_ref.shape}')

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)

print(f'start\_ts: {start\_ts}, end\_ts: {end\_ts}')

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()

print(f"df\_time: {df\_time.shape}")

# Aggregate per entity over the timeframe

peer\_agg = (

df\_time.groupby(entity\_field)

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

.reset\_index()

)

print(f"peer\_agg: {peer\_agg.shape}")

# Exclude current entity and compute medians

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

print(f"peers: {peers.shape}")

if not peers.empty:

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

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

# ---- Build the combined narrative (as requested) ----

# Example:

# "The efficiency for BR is 29.33% because TKM is 900.96 (all other country median 1500.00)

# while Total\_KM is 3071.36 (all other country median 1200.00). Recent trend: ..."

print(f"num\_med {num\_med}, den\_med {den\_med}")

if numerator is not None and denominator is not None:

if num\_med is not None and den\_med is not None:

narrative = (

f"The {metric\_label} for {entity\_value} is {value\_str} because "

f"{num\_key} is {numerator:,.2f} (all other country median {num\_med:,.2f}) "

f"while {den\_key} is {denominator:,.2f} (all other country median {den\_med:,.2f})."

)

else:

narrative = (

f"The {metric\_label} for {entity\_value} is {value\_str} because "

f"{num\_key} is {numerator:,.2f} while {den\_key} is {denominator:,.2f}."

)

else:

narrative = (

f"The {metric\_label} for {entity\_value or 'selected entity'} is {value\_str}, "

f"but numerator/denominator details were unavailable."

)

# ---- Append trend context (if available) ----

try:

if entity\_value and hasattr(self, "\_trend\_context\_sentence"):

trend\_sentence = self.\_trend\_context\_sentence(

entity\_value=entity\_value, entity\_field=entity\_field, periods=3

)

if trend\_sentence:

narrative = f"{narrative} {trend\_sentence}"

except Exception:

# Non-fatal; keep the base narrative

pass

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,

}

except Exception as e:

return {

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

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

}

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

"""Fallback handler for unhandled intents"""

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:

"""

Accepts either the full extracted\_context (with 'query\_analysis') or a single intent\_ctx.

Returns (entity\_field, entity\_value).

"""

entity\_field = default\_field

entity\_value = None

try:

# If 'query\_analysis' exists, use its first entry; otherwise assume this IS an intent\_ctx

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

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

else:

q = context or {}

# Prefer non-'all' entity with no numeric\_filter

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")

# Fallback: recover a 2‑letter uppercase code (e.g., BR) from the raw query

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:

"""

Returns a monthly pd.Series of efficiency (TKM/Total\_KM) for the given entity.

Uses the working dataframe produced by DataRetrievalAgent (self.\_working\_df).

Index is month (Timestamp at month start).

"""

print(" in \_monthly\_efficiency\_ts")

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

if df is None or df.empty:

return pd.Series(dtype=float)

# Filter to entity if provided

temp = df.copy()

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

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

# Need monthly granularity and numerators/denominators

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

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

return pd.Series(dtype=float)

# Aggregate per month, compute ratio

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()

print(f"\_monthly\_efficiency\_ts: {ts}")

return ts

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

"""Return (start\_ts, end\_ts) for the primary intent; falls back to wide window."""

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:

"""

Parse strings like 'TKM: 900.96/Total\_KM: 3071.36' into {'TKM': 900.96, 'Total\_KM': 3071.36}.

Works even if there are spaces or different ordering.

"""

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:

"""Helper: check if a metric should be shown as %"""

pct\_metrics = {"efficiency", "c1\_km\_perc"} # add more if needed

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

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

"""Extract N value from ranking queries"""

match = re.search(

r'(?:top|bottom|lowest|highest)\s+(\d+)|(\d+)\s+(?:lowest|highest)',

query.lower()

)

return int(match.group(1) or match.group(2)) if match else 5

def \_extract\_threshold(self, query: str) -> float:

"""Extract threshold value from query"""

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:

"""Determine appropriate chart type based on query"""

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:

"""Generate report that precisely matches query context"""

print("in \_generate\_conversational\_report ")

if not state.analysis\_results:

print("No analysis results!")

return "No results found for your query."

print("in \_generate\_conversational\_report 2")

# Get query context

query\_ctx = state.extracted\_context["query\_analysis"][0]

entities = query\_ctx.get("entities", [])

metrics = query\_ctx.get("metrics", [])

intent = query\_ctx["intent"]

# Replace your preferred\_order block with this:

preferred\_order = [

IntentType.PREDICT\_METRIC.value, # make prediction 1st

IntentType.DIAGNOSE\_METRIC.value,

IntentType.TREND\_ANALYSIS.value, # include trend explicitly

IntentType.VISUALIZE\_METRIC.value, # include visualize too

IntentType.RANK\_ENTITIES.value,

IntentType.FETCH\_METRIC.value,

IntentType.SUMMARIZE\_METRIC.value,

IntentType.COMPARE\_METRIC.value,

]

primary\_result = None

for k in preferred\_order:

if k in state.analysis\_results:

primary\_result = state.analysis\_results[k]

break

# As a very last resort, if nothing matched (shouldn't happen, but safe):

if primary\_result is None and state.analysis\_results:

# Try to pick 'prediction' directly if present

primary\_result = state.analysis\_results.get(IntentType.PREDICT\_METRIC.value)

# Else any result

if primary\_result is None:

primary\_result = next(iter(state.analysis\_results.values()))

results\_data = primary\_result.get("data", [])

# If still empty, try harvesting forecast from ANY prediction result

if not results\_data:

pred\_res = state.analysis\_results.get(IntentType.PREDICT\_METRIC.value)

if 'error' in pred\_res:

print(f'pred\_res: {pred\_res}')

return f"Prediction failed: {pred\_res['error']}"

print(f'pred\_res: {pred\_res}')

if pred\_res and pred\_res.get("forecast"):

# Safely derive entity info (present in pred\_res by design)

ent\_field = pred\_res.get("entity", {}).get("field", "entity")

ent\_value = pred\_res.get("entity", {}).get("value", "N/A")

results\_data = [

{

"field": ent\_field,

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

"metric": "efficiency",

"value": rec["value"], # ratio in 0..1

"support\_numbers": ""

}

for rec in pred\_res["forecast"]

]

# Also make sure we display the prediction narrative if the primary wasn't prediction

if not primary\_result.get("narrative") and pred\_res.get("narrative"):

primary\_result["narrative"] = pred\_res["narrative"]

# Fallback for prediction results that only include "forecast"

if not results\_data and primary\_result.get("type") == "prediction" and primary\_result.get("forecast"):

results\_data = [

{

"field": primary\_result["entity"]["field"],

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

"metric": "efficiency",

"value": rec["value"],

"support\_numbers": ""

}

for rec in primary\_result["forecast"]

]

if not results\_data:

print("No matching data found! (no data in primary\_result and no prediction forecast)")

return "No matching data found."

print(f'in \_generate\_conversational\_report 3, {primary\_result.get("narrative", "----")}')

# Prepare display elements

report\_parts = []

# 1. Conversational response (prefer diagnosis narrative if available)

if "narrative" in primary\_result and primary\_result["narrative"]:

response = primary\_result["narrative"]

else:

response = self.\_generate\_conversational\_response(query\_ctx, results\_data)

# response = self.\_generate\_conversational\_response(query\_ctx, results\_data)

print(f"\_generate\_conversational\_report,response {response} ")

report\_parts.append(response)

# 2. Supporting numbers and table

table = self.\_generate\_results\_table(query\_ctx, results\_data)

if table:

report\_parts.append("\n" + table)

# 3. Suggested queries

suggestions = self.\_generate\_suggested\_queries(state.extracted\_context)[:3]

if suggestions:

report\_parts.append("\nSuggested queries:")

report\_parts.extend([f"{i}. {q}" for i, q in enumerate(suggestions, 1)])

print(f'{"\n".join(report\_parts)}')

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])

entities = query\_ctx.get("entities", [])

metric\_value = results\_data[0].get("value", 0)

is\_percent = self.\_is\_percentage\_metric(query\_ctx["metrics"][0])

print(f'\_generate\_conversational\_response is\_percent : {is\_percent}, {query\_ctx["metrics"][0]}')

if is\_percent:

value\_str = f"{metric\_value:.2%}"

else:

value\_str = f"{metric\_value:,.2f}"

if intent == IntentType.RANK\_ENTITIES.value:

ranked\_entities = [r["field\_value"] for r in results\_data]

direction = "lowest" if "bottom" in query\_ctx["user\_query"].lower() else "highest"

print(f'\_generate\_conversational\_response ranked\_entities : {ranked\_entities}')

return f"The {len(ranked\_entities)} {direction} {metric\_name} are: {', '.join(ranked\_entities)}"

elif entities and entities[0]["value"] != "all":

entity\_value = entities[0]["value"]

# print(f'\_generate\_conversational\_response entity\_value : {entity\_value} \n The {metric\_name} for {entity\_value} is {metric\_value.2f} ')

return f"The {metric\_name} for {entity\_value} is {value\_str}"

else:

# print(f'\_generate\_conversational\_response metric\_value : {metric\_value} /n f"The {metric\_name} is {metric\_value:.2% if is\_percent else .2f}"')

return f"The {metric\_name} is {value\_str}"

def \_generate\_results\_table(self, query\_ctx: Dict, results\_data: List[Dict]) -> str:

"""Generate results table that precisely matches query context"""

if not results\_data:

return ""

entity\_field = query\_ctx.get("entities", [{}])[0].get("field", "entity")

entity\_name = self.\_get\_friendly\_name(entity\_field)

table\_data = []

for item in results\_data:

# Use queried entity value if specific entity was requested

entity\_value = (

query\_ctx["entities"][0]["value"]

if query\_ctx.get("entities") and query\_ctx["entities"][0]["value"] != "all"

else item.get("field\_value", "N/A")

)

print(f'\_generate\_results\_table table\_data \_is\_percentage\_metric : {self.\_is\_percentage\_metric(item.get("metric", ""))}, {item.get("metric", "")}')

table\_data.append({

entity\_name: entity\_value,

"Metric": f"{item.get('value', 0):.2%}" if self.\_is\_percentage\_metric(item.get("metric", "")) else f"{item.get('value', 0):.2f}",

"Supporting Numbers": item.get("support\_numbers", "")

})

df = pd.DataFrame(table\_data)

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 \_contribution\_analysis(self, entity\_value: str,

num\_key: str = "TKM",

den\_key: str = "Total\_KM",

entity\_field: str = "country\_code") -> dict:

"""

Return a structured analysis so the caller can craft one combined sentence.

Returns:

{

"num\_driver": bool, # True if numerator is unusually low

"den\_driver": bool, # True if denominator is unusually high

"num\_med": float | None, # peer median for numerator (excludes this entity)

"den\_med": float | None # peer median for denominator (excludes this entity)

}

"""

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

out = {"num\_driver": False, "den\_driver": False, "num\_med": None, "den\_med": None}

if df is None or df.empty:

return out

if not all(col in df.columns for col in [entity\_field, num\_key, den\_key]):

return out

peer\_df = df.groupby(entity\_field).agg({num\_key: "sum", den\_key: "sum"}).reset\_index()

if entity\_value not in peer\_df[entity\_field].values:

return out

row = peer\_df.loc[peer\_df[entity\_field] == entity\_value].iloc[0]

num\_val, den\_val = float(row[num\_key]), float(row[den\_key])

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

if peers.empty:

return out

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

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

out["num\_med"] = num\_med

out["den\_med"] = den\_med

# Heuristics: 25% bands; tweak if you like

out["num\_driver"] = num\_val < 0.75 \* num\_med

out["den\_driver"] = den\_val > 1.25 \* den\_med

return out

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 \_trend\_context\_sentence(self, entity\_value: str, entity\_field: str = "country\_code", periods: int = 3) -> str:

"""

Build a short sentence describing the recent efficiency trend for an entity.

Uses the filtered working dataframe (self.\_working\_df) and drive\_month.

"""

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]

# Need both TKM and Total\_KM to compute efficiency

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

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

return ""

# Aggregate by month and compute efficiency

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 ""

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

ts = ts[ts["efficiency"].notna()].sort\_index().tail(periods)

if ts.empty or len(ts) < 2:

return ""

vals = (ts["efficiency"] \* 100).round(2).tolist()

months = [idx.strftime("%b %Y") for idx in ts.index]

delta\_pp = round(vals[-1] - vals[0], 2)

direction = "increased" if delta\_pp > 0 else "decreased" if delta\_pp < 0 else "stayed flat"

seq = " → ".join(f"{v}%" for v in vals)

span = f"{months[0]}–{months[-1]}" if months[0] != months[-1] else months[-1]

return f"Recent trend: over the last {len(vals)} months ({span}), efficiency {direction} from {seq} ({delta\_pp:+.2f} pp)."

"""### Main function"""

def main():

# Configuration

os.environ["OPENAI\_API\_KEY"] = "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"

gsheet\_url = "https://docs.google.com/spreadsheets/d/19k-\_OyARAwHfM2wxlZszdrLlmQoAab15V4gJKJMSMtQ/edit?gid=805320532#gid=805320532"

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

# Initialize orchestrator

orchestrator = OrchestratorAgent(gsheet\_url, semantic\_layer\_path, os.environ["OPENAI\_API\_KEY"])

query = "which country has better TKM country ID or BR ?"

# 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("Query Analysis:", json.dumps({"intents": [ctx["intent"] for ctx in result.extracted\_context.get("query\_analysis", [])], "extracted\_context": result.extracted\_context, "similar\_contexts": result.similar\_contexts}, indent=2))

# Display results

# print("\nQuery Analysis:")

# print(json.dumps({

# "intents": [ctx["intent"] for ctx in result.extracted\_context.get("query\_analysis", [])],

# "metrics": list({m for ctx in result.extracted\_context.get("query\_analysis", [])

# for m in ctx.get("metrics", [])})

# }, indent=2))

# print("\nAnalysis Results:")

# for intent, res in result.analysis\_results.items():

# print(f"\n{intent.upper()} Results:")

# print(json.dumps(res, indent=2))

print("\nFinal Report:")

print(result.report)

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

main()

"""# New Section

##archive

"""