# Install required packages

!pip install google-generativeai pandas gcsfs langgraph langchain-core

!gcloud auth login

import pandas as pd

import google.generativeai as genai

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

import json

from langgraph.graph import StateGraph, END

from langgraph.graph import add\_messages

# Configure Gemini

genai.configure(api\_key="your-api-key") # Replace with your API key

# Define the State

class AgentState(TypedDict):

user\_query: str

intent: Dict[str, Any]

raw\_data: Optional[pd.DataFrame]

calculated\_metrics: Dict[str, Any]

analysis\_results: Dict[str, Any]

final\_output: str

next\_nodes: List[str]

execution\_path: List[str]

class DataSourceManager:

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

self.sources = {'synthetic\_data': 'gs://basedataadmin/synthetic\_data.csv'}

def get\_data(self, source\_name: str = 'synthetic\_data') -> pd.DataFrame:

"""Load data from GCS"""

return pd.read\_csv(self.sources[source\_name])

class QueryUnderstandingAgent:

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

self.model = genai.GenerativeModel('gemini-1.5-flash')

def \_\_call\_\_(self, state: AgentState) -> AgentState:

"""Understand user query and extract dimensional intent"""

print("Executing QueryUnderstandingAgent...")

query = state["user\_query"]

prompt = f"""

Analyze this business query and extract structured intent components:

Query: "{query}"

Return JSON with:

- primary\_metric: "utilization" | "nbl" | "shrinkage" | "performance"

- scope: "overall" | "breakdown" | "comparison" | "specific"

- breakdown\_dimension: "team" | "edl" | "location" | "project" | "none"

- entities: [] # specific teams/locations mentioned

- filters: [] # "current", "top", "bottom", "trend", "target"

- question\_type: "value" | "breakdown" | "comparison" | "identification"

Focus only on what is explicitly asked. Be precise.

Examples:

"What is NBL percentage?" → {{"primary\_metric": "nbl", "scope": "overall", "breakdown\_dimension": "none", "entities": [], "filters": ["current"], "question\_type": "value"}}

"Show utilization by team for top performers" → {{"primary\_metric": "utilization", "scope": "breakdown", "breakdown\_dimension": "team", "entities": [], "filters": ["top", "current"], "question\_type": "breakdown"}}

"Compare GB and CS team utilization" → {{"primary\_metric": "utilization", "scope": "comparison", "breakdown\_dimension": "team", "entities": ["GB", "CS"], "filters": ["current"], "question\_type": "comparison"}}

"""

try:

response = self.model.generate\_content(prompt)

intent\_data = json.loads(response.text)

# Validate and ensure required fields

required\_fields = ['primary\_metric', 'scope', 'breakdown\_dimension', 'entities', 'filters', 'question\_type']

for field in required\_fields:

if field not in intent\_data:

if field == 'entities':

intent\_data[field] = []

elif field == 'filters':

intent\_data[field] = ['current']

else:

intent\_data[field] = 'overall' if field == 'scope' else 'utilization'

state["intent"] = intent\_data

state["execution\_path"].append("query\_understanding")

except Exception as e:

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

# Enhanced fallback logic

query\_lower = query.lower()

if 'nbl' in query\_lower or 'non-billable' in query\_lower:

state["intent"] = {

"primary\_metric": "nbl",

"scope": "overall",

"breakdown\_dimension": "none",

"entities": [],

"filters": ["current"],

"question\_type": "value"

}

elif 'shrinkage' in query\_lower:

state["intent"] = {

"primary\_metric": "shrinkage",

"scope": "overall",

"breakdown\_dimension": "none",

"entities": [],

"filters": ["current"],

"question\_type": "value"

}

elif 'team' in query\_lower and 'utilization' in query\_lower:

state["intent"] = {

"primary\_metric": "utilization",

"scope": "breakdown",

"breakdown\_dimension": "team",

"entities": [],

"filters": ["current"],

"question\_type": "breakdown"

}

elif 'edl' in query\_lower and 'utilization' in query\_lower:

state["intent"] = {

"primary\_metric": "utilization",

"scope": "breakdown",

"breakdown\_dimension": "edl",

"entities": [],

"filters": ["current"],

"question\_type": "breakdown"

}

else:

state["intent"] = {

"primary\_metric": "utilization",

"scope": "overall",

"breakdown\_dimension": "none",

"entities": [],

"filters": ["current"],

"question\_type": "value"

}

return state

class DataRetrievalAgent:

def \_\_init\_\_(self, data\_manager: DataSourceManager):

self.data\_manager = data\_manager

def \_\_call\_\_(self, state: AgentState) -> AgentState:

"""Load and filter data based on query context"""

print("Executing DataRetrievalAgent...")

df = self.data\_manager.get\_data()

# Basic data cleaning with updated column names

df = df.dropna(subset=['utilization\_\_percentage'])

query\_info = state["intent"]

# Apply filters based on intent

if query\_info['scope'] == 'specific' and query\_info['entities']:

# Filter by specific entities (teams, EDLs, etc.)

entity\_column = 'edl\_name' if query\_info['breakdown\_dimension'] == 'edl' else 'team\_name'

df = df[df[entity\_column].isin(query\_info['entities'])]

state["raw\_data"] = df

state["execution\_path"].append("data\_retrieval")

return state

class MetricCalculationAgent:

def \_\_call\_\_(self, state: AgentState) -> AgentState:

"""Calculate business metrics based on query intent"""

print("Executing MetricCalculationAgent...")

df = state["raw\_data"]

query\_info = state["intent"]

# Initialize results

results = {

'calculated\_metrics': {},

'level\_metrics': {},

'query\_focus': query\_info

}

# Calculate only requested metrics

primary\_metric = query\_info['primary\_metric']

scope = query\_info['scope']

if primary\_metric in ['utilization', 'performance']:

# Core utilization metrics

overall\_util = df['utilization\_\_percentage'].mean() \* 100

results['calculated\_metrics']['overall\_utilization'] = round(overall\_util, 1)

# Level-specific utilization if breakdown requested

if scope == 'breakdown':

dimension = query\_info['breakdown\_dimension']

if dimension == 'edl':

edl\_metrics = df.groupby('edl\_name').agg({

'utilization\_\_percentage': 'mean',

'billed\_fte': 'sum',

'unbilled\_fte': 'sum'

})

edl\_metrics['utilization\_\_percentage'] = edl\_metrics['utilization\_\_percentage'] \* 100

results['level\_metrics']['edl\_breakdown'] = edl\_metrics.to\_dict()

if primary\_metric in ['nbl', 'performance']:

# NBL calculation

total\_billed\_fte = df['billed\_fte'].sum()

total\_unbilled\_fte = df['unbilled\_fte'].sum()

total\_fte = df['total\_fte'].sum()

nbl\_percentage = (total\_unbilled\_fte / total\_fte) \* 100 if total\_fte > 0 else 0

results['calculated\_metrics']['nbl\_percentage'] = round(nbl\_percentage, 1)

if primary\_metric in ['shrinkage', 'performance']:

# Shrinkage calculation with updated column name

allocated\_ftes = (df['allocation\_\_percentage'] / 100).sum()

actual\_billed\_ftes = df['billed\_fte'].sum()

shrinkage = ((allocated\_ftes - actual\_billed\_ftes) / allocated\_ftes) \* 100 if allocated\_ftes > 0 else 0

results['calculated\_metrics']['shrinkage\_percentage'] = round(shrinkage, 1)

state["calculated\_metrics"] = results

state["execution\_path"].append("metric\_calculation")

return state

class AnalysisAgent:

def \_\_call\_\_(self, state: AgentState) -> AgentState:

"""Analyze patterns based on query intent"""

print("Executing AnalysisAgent...")

df = state["raw\_data"]

metrics = state["calculated\_metrics"]

query\_info = state["intent"]

analysis\_results = {

'insights': [],

'recommendations': [],

'detailed\_analysis': {}

}

primary\_metric = query\_info['primary\_metric']

filters = query\_info['filters']

scope = query\_info['scope']

# Only perform analysis relevant to the query

if primary\_metric == 'utilization':

edl\_performance = df.groupby('edl\_name')['utilization\_\_percentage'].mean() \* 100

# Top performers analysis if requested

if 'top' in filters:

top\_performers = edl\_performance[edl\_performance > 95].sort\_values(ascending=False).head(3)

analysis\_results['detailed\_analysis']['top\_performers'] = top\_performers.to\_dict()

analysis\_results['insights'].append(f"Top performers: {', '.join(list(top\_performers.keys()))}")

# Areas needing attention if requested

if 'bottom' in filters or 'attention' in query\_info.get('question\_type', ''):

needs\_attention = edl\_performance[edl\_performance < 90].sort\_values(ascending=True).head(3)

analysis\_results['detailed\_analysis']['needs\_attention'] = needs\_attention.to\_dict()

if len(needs\_attention) > 0:

analysis\_results['insights'].append(f"Areas needing attention: {', '.join(list(needs\_attention.keys()))}")

analysis\_results['recommendations'].append(f"Focus on performance improvement for {list(needs\_attention.keys())[0]} team")

elif primary\_metric == 'nbl':

nbl\_value = metrics['calculated\_metrics']['nbl\_percentage']

if nbl\_value > 7:

analysis\_results['recommendations'].append("Consider optimizing bench strength to reduce NBL percentage")

elif nbl\_value < 3:

analysis\_results['recommendations'].append("Maintain current bench levels for operational flexibility")

state["analysis\_results"] = analysis\_results

state["execution\_path"].append("analysis")

return state

class ReportGenerationAgent:

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

self.model = genai.GenerativeModel('gemini-1.5-flash')

def \_\_call\_\_(self, state: AgentState) -> AgentState:

"""Generate focused business report based on query intent"""

print("Executing ReportGenerationAgent...")

metrics = state["calculated\_metrics"]

analysis = state["analysis\_results"]

query\_info = state["intent"]

query = state["user\_query"]

# Build focused prompt based on query intent

prompt = self.\_build\_focused\_prompt(query, metrics, analysis, query\_info)

try:

response = self.model.generate\_content(prompt)

report = response.text

except Exception as e:

print(f"LLM report generation failed: {e}")

report = self.\_generate\_focused\_fallback\_report(metrics, analysis, query\_info, query)

state["final\_output"] = report

state["execution\_path"].append("report\_generation")

return state

def \_build\_focused\_prompt(self, query: str, metrics: Dict, analysis: Dict, query\_info: Dict) -> str:

"""Build a focused prompt based on query intent"""

primary\_metric = query\_info['primary\_metric']

scope = query\_info['scope']

question\_type = query\_info['question\_type']

base\_prompt = f"""

Create a professional business response focused ONLY on what was asked.

Original Query: {query}

Requirements:

- Answer ONLY the specific question asked

- Use professional business language

- No symbols or emojis

- Keep it concise but insightful

- Focus on actionable insights relevant to the query

- Do not include unrelated metrics or analysis

"""

# Metric-specific prompts

if primary\_metric == 'nbl' and question\_type == 'value':

return base\_prompt + f"""

Key Metric:

- NBL Percentage: {metrics['calculated\_metrics']['nbl\_percentage']}%

Provide a brief executive summary focused solely on NBL percentage.

"""

elif primary\_metric == 'utilization' and scope == 'overall':

return base\_prompt + f"""

Key Metric:

- Overall Utilization: {metrics['calculated\_metrics']['overall\_utilization']}%

Provide a concise summary of the overall utilization rate.

"""

elif primary\_metric == 'utilization' and scope == 'breakdown':

dimension = query\_info['breakdown\_dimension']

breakdown\_data = metrics['level\_metrics'].get(f'{dimension}\_breakdown', {})

return base\_prompt + f"""

Breakdown Analysis:

- Dimension: {dimension}

- Data: {breakdown\_data}

Provide a focused breakdown analysis.

"""

else:

# General fallback prompt

return base\_prompt + f"""

Relevant Metrics: {metrics['calculated\_metrics']}

Provide a focused response addressing the specific query.

"""

def \_generate\_focused\_fallback\_report(self, metrics: Dict, analysis: Dict, query\_info: Dict, query: str) -> str:

"""Fallback report generator that respects query focus"""

primary\_metric = query\_info['primary\_metric']

calculated\_metrics = metrics['calculated\_metrics']

if primary\_metric == 'nbl':

return f"""

NBL Analysis

Based on current operational data, the Non-Billable Load percentage is {calculated\_metrics['nbl\_percentage']}%.

{analysis['recommendations'][0] if analysis['recommendations'] else 'This level supports organizational flexibility.'}

"""

elif primary\_metric == 'utilization' and query\_info['scope'] == 'overall':

return f"""

Utilization Summary

Current overall utilization rate: {calculated\_metrics['overall\_utilization']}%

"""

elif primary\_metric == 'utilization' and query\_info['scope'] == 'breakdown':

dimension = query\_info['breakdown\_dimension']

breakdown = metrics['level\_metrics'].get(f'{dimension}\_breakdown', {})

report\_lines = [f"Utilization by {dimension.upper()}:"]

for entity, data in breakdown.items():

if 'utilization\_\_percentage' in data:

util\_rate = data['utilization\_\_percentage']

report\_lines.append(f"- {entity}: {util\_rate:.1f}%")

return "\n".join(report\_lines)

else:

# Minimal general fallback

main\_metric\_value = list(calculated\_metrics.values())[0] if calculated\_metrics else "N/A"

return f"Query: {query}\nResult: {main\_metric\_value}"

# Router Functions for Conditional Logic

def route\_after\_understanding(state: AgentState) -> str:

"""Decide which agents to run based on query intent"""

intent = state["intent"]

# Always need data retrieval and metrics

if intent["primary\_metric"] == "nbl" and intent["scope"] == "overall":

# Simple NBL query - skip analysis

return "data\_retrieval"

elif intent["question\_type"] == "value" and not intent["filters"]:

# Simple value query - skip analysis

return "data\_retrieval"

else:

# Complex query - include analysis

return "data\_retrieval"

def route\_after\_metrics(state: AgentState) -> str:

"""Decide if analysis is needed"""

intent = state["intent"]

if intent["filters"] or intent["question\_type"] in ["comparison", "identification"]:

return "analysis"

else:

return "report\_generation"

def route\_after\_analysis(state: AgentState) -> str:

"""Always go to report after analysis"""

return "report\_generation"

def route\_after\_data(state: AgentState) -> str:

"""Always go to metrics after data retrieval"""

return "metric\_calculation"

# Create the LangGraph Workflow

def create\_workflow():

"""Create and compile the LangGraph workflow"""

# Initialize components

data\_manager = DataSourceManager()

# Create graph

workflow = StateGraph(AgentState)

# Add nodes

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

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

workflow.add\_node("metric\_calculation", MetricCalculationAgent())

workflow.add\_node("analysis", AnalysisAgent())

workflow.add\_node("report\_generation", ReportGenerationAgent())

# Define entry point

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

# Add conditional edges

workflow.add\_conditional\_edges(

"query\_understanding",

route\_after\_understanding,

{

"data\_retrieval": "data\_retrieval",

}

)

workflow.add\_conditional\_edges(

"data\_retrieval",

route\_after\_data,

{

"metric\_calculation": "metric\_calculation",

}

)

workflow.add\_conditional\_edges(

"metric\_calculation",

route\_after\_metrics,

{

"analysis": "analysis",

"report\_generation": "report\_generation"

}

)

workflow.add\_conditional\_edges(

"analysis",

route\_after\_analysis,

{

"report\_generation": "report\_generation"

}

)

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

# Compile the graph

return workflow.compile()

# Initialize the system

print("Initializing LangGraph Agentic Analysis System...")

app = create\_workflow()

print("System ready!")

# Test function

def process\_query(query: str) -> str:

"""Process a query through the LangGraph workflow"""

print(f"\n{'='\*60}")

print(f"PROCESSING QUERY: {query}")

print(f"{'='\*60}")

initial\_state = AgentState(

user\_query=query,

intent={},

raw\_data=None,

calculated\_metrics={},

analysis\_results={},

final\_output="",

next\_nodes=[],

execution\_path=[]

)

try:

# Execute the graph

final\_state = app.invoke(initial\_state)

print(f"Execution Path: {final\_state['execution\_path']}")

print(f"\nRESULT:")

print(f"{'='\*60}")

print(final\_state['final\_output'])

print(f"{'='\*60}")

return final\_state['final\_output']

except Exception as e:

error\_msg = f"Error processing query: {str(e)}"

print(error\_msg)

return error\_msg

# Test the LangGraph system

test\_queries = [

"What is our current utilization percentage?",

"What is our NBL percentage?",

"Show me utilization by EDL",

"Which teams need attention?",

"Compare GB and CS team utilization",

"Show me top performing teams by utilization"

]

print("Starting LangGraph Test Suite...")

for query in test\_queries:

process\_query(query)

print("\nLangGraph System Test Complete!")