# Install required packages

!pip install google-generativeai pandas gcsfs

!gcloud auth login

import pandas as pd

import google.generativeai as genai

from typing import Dict, Any, List

import json

# Configure Gemini

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

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 QueryAgent:

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

self.model = genai.GenerativeModel('gemini-pro')

def execute(self, query: str, context: Dict) -> Dict:

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

prompt = f"""

Analyze this business query and extract key components:

Query: "{query}"

Return JSON with:

- intent: "utilization\_analysis", "nbl\_analysis", "trend\_analysis", "comparison", "performance\_review"

- entities: list of entities mentioned like ["EDL", "project", "manager", "location"]

- timeframe: "current", "historical", "trend"

- level: "overall", "EDL", "project", "manager", "location"

- metrics: list of metrics like ["utilization", "nbl", "shrinkage"]

Example: "Show me utilization by EDL" → {{"intent": "utilization\_analysis", "entities": ["EDL"], "timeframe": "current", "level": "EDL", "metrics": ["utilization"]}}

"""

try:

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

return json.loads(response.text)

except:

# Fallback for common queries

query\_lower = query.lower()

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

return {"intent": "utilization\_analysis", "entities": ["EDL"], "timeframe": "current", "level": "EDL", "metrics": ["utilization"]}

elif 'nbl' in query\_lower:

return {"intent": "nbl\_analysis", "entities": [], "timeframe": "current", "level": "overall", "metrics": ["nbl"]}

else:

return {"intent": "analysis", "entities": [], "timeframe": "current", "level": "overall", "metrics": ["utilization", "nbl"]}

class DataAgent:

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

self.data\_manager = data\_manager

self.df = None

def execute(self, query: str, context: Dict) -> Dict:

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

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

# Basic data cleaning

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

query\_info = context['query\_understanding']

# Apply basic filters based on query

if query\_info['level'] != 'overall':

# In future: filter by specific EDL/project/manager

pass

return {

'data': self.df,

'record\_count': len(self.df),

'data\_columns': list(self.df.columns)

}

class MetricAgent:

def execute(self, query: str, context: Dict) -> Dict:

"""Calculate business metrics"""

df = context['data\_retrieval']['data']

query\_info = context['query\_understanding']

# Core metrics

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

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

# Shrinkage calculation

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

# Level-specific metrics

level\_metrics = {}

if query\_info['level'] == 'EDL':

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

'utilization\_percentage': 'mean',

'billed\_fte': 'sum',

'unbilled\_fte': 'sum'

}).to\_dict()

level\_metrics['edl\_breakdown'] = edl\_metrics

return {

'overall\_utilization': round(overall\_util, 1),

'nbl\_percentage': round(nbl\_percentage, 1),

'shrinkage\_percentage': round(shrinkage, 1),

'total\_billed\_fte': round(total\_billed\_fte, 1),

'total\_unbilled\_fte': round(total\_unbilled\_fte, 1),

'level\_metrics': level\_metrics

}

class AnalysisAgent:

def execute(self, query: str, context: Dict) -> Dict:

"""Analyze patterns and generate insights"""

df = context['data\_retrieval']['data']

metrics = context['metric\_calculation']

# Top performers (utilization > 95%)

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

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

# Areas needing attention (utilization < 90%)

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

# Trend analysis (simple version)

monthly\_trend = df.groupby('month')['utilization\_percentage'].mean()

if len(monthly\_trend) > 1:

trend = "improving" if monthly\_trend.iloc[-1] > monthly\_trend.iloc[-2] else "declining"

trend\_magnitude = abs(monthly\_trend.iloc[-1] - monthly\_trend.iloc[-2]) \* 100

else:

trend = "stable"

trend\_magnitude = 0

# Generate recommendations

recommendations = []

if len(needs\_attention) > 0:

worst\_performer = needs\_attention.index[0]

recommendations.append(f"Focus on performance improvement for {worst\_performer} team")

if metrics['nbl\_percentage'] > 7:

recommendations.append("Consider optimizing bench strength to reduce NBL percentage")

elif metrics['nbl\_percentage'] < 3:

recommendations.append("Maintain current bench levels for operational flexibility")

return {

'top\_performers': top\_performers.to\_dict(),

'needs\_attention': needs\_attention.to\_dict(),

'performance\_trend': trend,

'trend\_magnitude': round(trend\_magnitude, 1),

'recommendations': recommendations

}

class ReportAgent:

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

self.model = genai.GenerativeModel('gemini-pro')

def execute(self, query: str, context: Dict) -> Dict:

"""Generate eloquent business report"""

metrics = context['metric\_calculation']

analysis = context['analysis\_insights']

query\_info = context['query\_understanding']

prompt = f"""

Create a professional business executive summary based on the following analysis:

Original Query: {query}

Key Metrics:

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

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

- Shrinkage: {metrics['shrinkage\_percentage']}%

Performance Analysis:

- Top Performers: {list(analysis['top\_performers'].keys())}

- Areas Needing Attention: {list(analysis['needs\_attention'].keys())}

- Performance Trend: {analysis['performance\_trend']}

- Recommendations: {analysis['recommendations']}

Requirements:

- Use professional business language

- No symbols or emojis

- Structure: Executive Summary → Key Findings → Recommendations

- Keep it concise but insightful

- Focus on actionable insights

"""

try:

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

report = response.text

except:

# Fallback report

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

return {'business\_report': report}

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

"""Fallback report if Gemini fails"""

return f"""

Executive Performance Summary

Based on analysis of operational data, our organization demonstrates strong performance with an overall utilization rate of {metrics['overall\_utilization']} percent.

Key operational metrics show a healthy balance with {metrics['nbl\_percentage']} percent of resources in non-billable roles, supporting organizational flexibility.

Top performing teams include {', '.join(list(analysis['top\_performers'].keys()))} with utilization rates above 95 percent.

Areas requiring management attention include {', '.join(list(analysis['needs\_attention'].keys()))} where utilization falls below 90 percent.

We recommend {', '.join(analysis['recommendations'])} to maintain and improve operational efficiency.

"""

class SimpleOrchestrator:

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

self.data\_manager = DataSourceManager()

self.agents = {

'query\_understanding': QueryAgent(),

'data\_retrieval': DataAgent(self.data\_manager),

'metric\_calculation': MetricAgent(),

'analysis\_insights': AnalysisAgent(),

'report\_generation': ReportAgent()

}

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

"""Main processing pipeline"""

print(f"Processing query: {query}")

context = {}

try:

# Execute agents in sequence

for agent\_name, agent in self.agents.items():

print(f"Executing {agent\_name}...")

context[agent\_name] = agent.execute(query, context)

return context['report\_generation']['business\_report']

except Exception as e:

return f"Analysis completed with partial results. Error: {str(e)}"

# Initialize the system

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

orchestrator = SimpleOrchestrator()

print("System ready!")

# Test the system

test\_queries = [

"What is our current utilization percentage?",

"Show me utilization by EDL",

"What is our NBL percentage?",

"Which teams need attention?"

]

for query in test\_queries:

print(f"\n{'='\*50}")

print(f"QUERY: {query}")

print(f"{'='\*50}")

result = orchestrator.process\_query(query)

print(result)

print(f"{'='\*50}")