class QueryUnderstandingAgent:

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

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

# Semantic mapping: user terms -> actual column names

self.column\_mapping = {

'team': 'edl\_name',

'edl': 'edl\_name',

'location': 'utilization\_location',

'project': 'project\_name',

'manager': 'manager\_name'

}

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" | "manager" | "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"}}

"Show me utilization by location" → {{"primary\_metric": "utilization", "scope": "breakdown", "breakdown\_dimension": "location", "entities": [], "filters": ["current"], "question\_type": "breakdown"}}

"""

try:

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

response\_text = response.text.strip()

print(f"Raw LLM Response: {response\_text}")

# Extract JSON from response

json\_text = response\_text.replace('```json', '').replace('```', '').strip()

start\_idx = json\_text.find('{')

end\_idx = json\_text.rfind('}') + 1

if start\_idx != -1 and end\_idx != -1:

json\_text = json\_text[start\_idx:end\_idx]

intent\_data = json.loads(json\_text)

print(f"Parsed Intent Data: {intent\_data}")

# Validate 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'

# Map semantic dimension to actual column

dimension = intent\_data['breakdown\_dimension']

intent\_data['actual\_column'] = self.column\_mapping.get(dimension, dimension)

state["intent"] = intent\_data

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

except Exception as e:

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

# Enhanced fallback with semantic mapping

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", "actual\_column": "none"

}

elif 'shrinkage' in query\_lower:

state["intent"] = {

"primary\_metric": "shrinkage", "scope": "overall", "breakdown\_dimension": "none",

"entities": [], "filters": ["current"], "question\_type": "value", "actual\_column": "none"

}

elif any(word in query\_lower for word in ['location', 'city', 'country', 'region']):

state["intent"] = {

"primary\_metric": "utilization", "scope": "breakdown", "breakdown\_dimension": "location",

"entities": [], "filters": ["current"], "question\_type": "breakdown", "actual\_column": "utilization\_location"

}

elif 'team' in query\_lower or 'edl' in query\_lower:

state["intent"] = {

"primary\_metric": "utilization", "scope": "breakdown", "breakdown\_dimension": "team",

"entities": [], "filters": ["current"], "question\_type": "breakdown", "actual\_column": "edl\_name"

}

else:

state["intent"] = {

"primary\_metric": "utilization", "scope": "overall", "breakdown\_dimension": "none",

"entities": [], "filters": ["current"], "question\_type": "value", "actual\_column": "none"

}

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()

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

query\_info = state["intent"]

# Use actual\_column for filtering

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

entity\_column = query\_info.get('actual\_column', 'edl\_name')

if entity\_column != 'none' and entity\_column in df.columns:

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"]

results = {

'calculated\_metrics': {},

'level\_metrics': {},

'query\_focus': query\_info

}

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

scope = query\_info['scope']

actual\_column = query\_info.get('actual\_column', 'none')

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

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

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

# Use actual\_column for breakdown

if scope == 'breakdown' and actual\_column != 'none' and actual\_column in df.columns:

breakdown\_metrics = df.groupby(actual\_column).agg({

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

'billed\_fte': 'sum',

'unbilled\_fte': 'sum'

})

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

results['level\_metrics'][f'{actual\_column}\_breakdown'] = breakdown\_metrics.to\_dict()

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

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']:

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']

actual\_column = query\_info.get('actual\_column', 'edl\_name')

if primary\_metric == 'utilization' and actual\_column != 'none' and actual\_column in df.columns:

# Use actual\_column for analysis

performance\_data = df.groupby(actual\_column)['utilization\_\_percentage'].mean() \* 100

if 'top' in filters:

top\_performers = performance\_data[performance\_data > 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()))}")

if 'bottom' in filters:

needs\_attention = performance\_data[performance\_data < 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]}")

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"]

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:

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

scope = query\_info['scope']

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

actual\_column = query\_info.get('actual\_column', 'none')

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

"""

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':

# Use semantic term in prompt, but actual column for data

semantic\_term = query\_info['breakdown\_dimension']

breakdown\_key = f"{actual\_column}\_breakdown"

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

return base\_prompt + f"""

Breakdown Analysis:

- Dimension: {semantic\_term}

- Data: {breakdown\_data}

Provide a focused breakdown analysis by {semantic\_term}.

"""

else:

return base\_prompt + f"""

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

Provide a focused response addressing the specific query.

"""