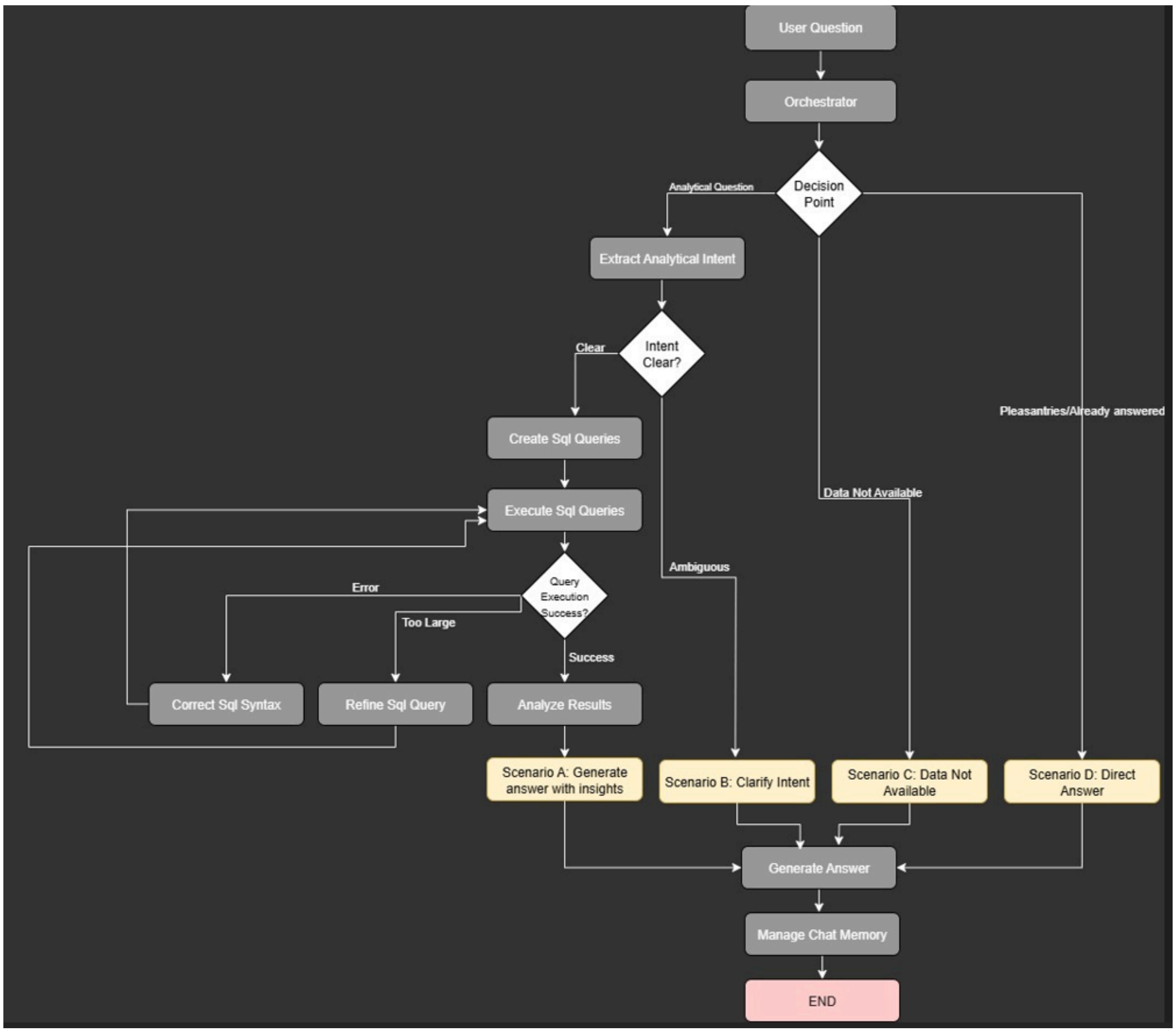
GenAI Database Copilot - Technical Documentation

Overview

This is a conversational AI agent built with LangChain and LangGraph that enables natural language querying of enterprise databases. The agent processes user questions, extracts analytical intent, generates SQL queries, executes them, and provides intelligent business insights.

Architecture



Core Components

The system is built using a **LangGraph StateGraph** architecture with the following key nodes:

1. **reset_state** - Initializes state for new conversations
2. **orchestrator** - Decision-making hub that routes to appropriate tools
3. **extract_analytical_intent** - Processes user questions to extract analytical requirements
4. **create_sql_query_or_queries** - Generates SQL queries from analytical intents
5. **execute_sql_query** - Executes generated queries and processes results
6. **correct_syntax_sql_query** - Corrects SQL syntax errors with up to 3 attempts
7. **refine_sql_query** - Optimizes queries that produce results exceeding token limits
8. **generate_answer** - Creates final responses with business insights

State Management

The agent maintains state through a *State* TypedDict containing:

```python
class State(TypedDict):
    objects_documentation: str          # Database schema documentation
    database_content: str              # Database content summary
    sql_dialect: str                   # SQL dialect (SQLite)
    messages_log: Sequence[BaseMessage] # Conversation history
    intermediate_steps: list[AgentAction] # Agent execution log
    analytical_intent: list[str]       # Extracted analytical requirements
    current_question: str              # Current user question
    current_sql_queries: list[dict]    # Generated queries and results
    generate_answer_details: dict      # Response scenario information
    llm_answer: BaseMessage           # Final AI response
```

LLM Configuration

Primary Models

Smart Model (GPT-4.1)

- Model: gpt-4.1
- Temperature: 0 (deterministic responses)
- Usage: Complex reasoning, analytical intent extraction, SQL generation

Fast Model (GPT-4o)

- Model: gpt-4o
- Temperature: 0 (deterministic responses)

- Usage: Simple tasks, memory summarization, quick analysis

Token Management

- **Maximum SQL result tokens**: 500 tokens
- **Chat history limit**: 1000 tokens (triggers summarization)
- **Token counting**: Uses tiktoken encoding for GPT-4o

Memory Management

Chat History Management

The *manage_memory_chat_history()* function implements intelligent memory management:

1. **Token Monitoring**: Tracks total tokens in conversation history.
2. **Summarization Trigger**: When history exceeds 1000 tokens and >4 message pairs.
3. **Preservation Strategy**: Keeps last 4 message pairs, summarizes older messages.
4. **Summary Generation**: Uses fast LLM to create 400-token summaries.

Memory Persistence

- **Checkpointer**: Uses LangGraph's *MemorySaver* for conversation persistence
- **Thread Management**: Each conversation gets unique thread_id for session tracking
- **State Persistence**: Maintains conversation context across multiple queries

Orchestrator

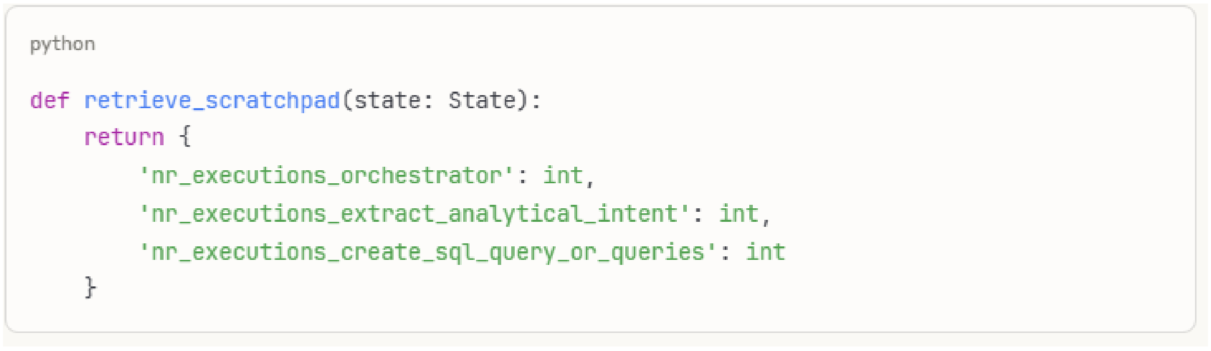
Core Functionality

The orchestrator serves as the central decision-making component that:

1. **Analyzes incoming questions** for analytical complexity
2. **Routes to appropriate tools** based on question type
3. **Maintains execution flow** through scratchpad tracking
4. **Handles edge cases** like pleasantries and missing data

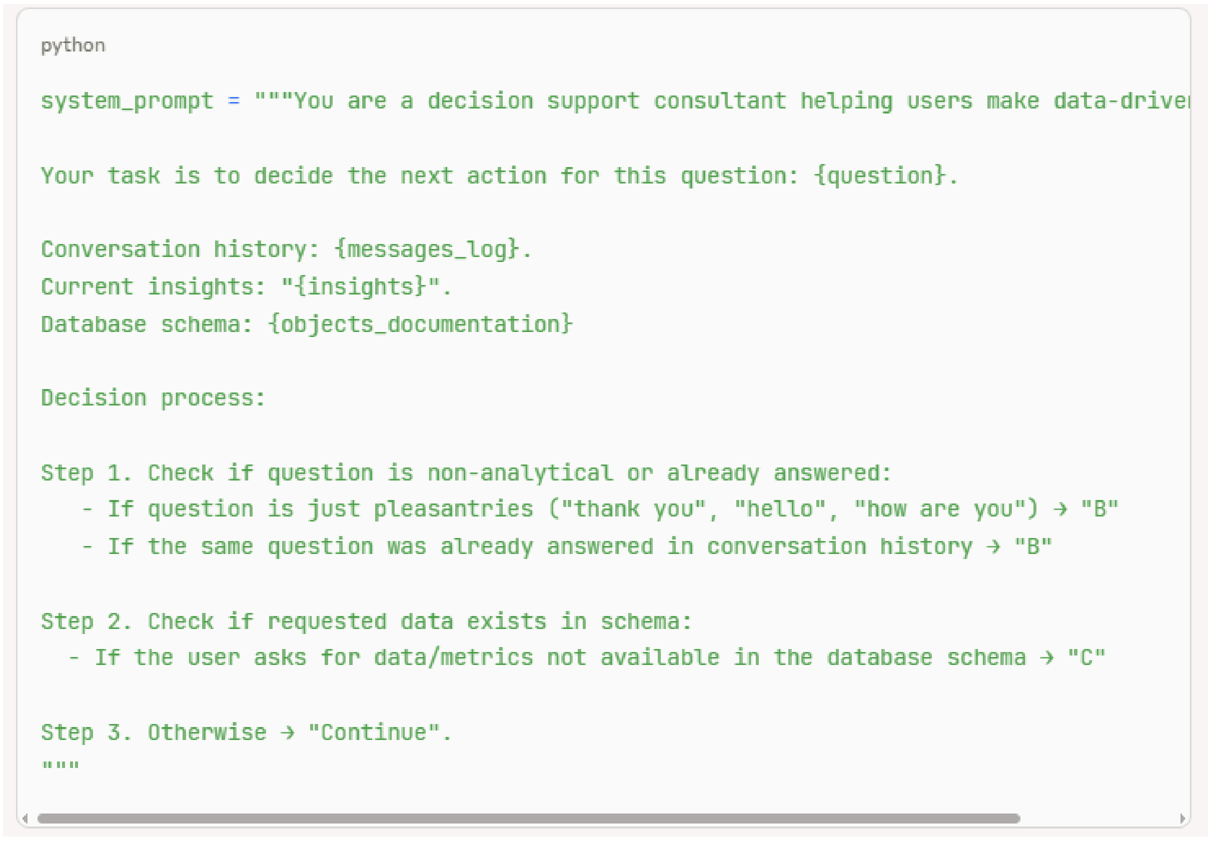
Scratchpad System

The scratchpad tracks tool executions to maintain control flow:


python

```python
def retrieve_scratchpad(state: State):
    return {
        'nr_executions_orchestrator': int,
        'nr_executions_extract_analytical_intent': int,
        'nr_executions_create_sql_query_or_queries': int
    }
```

Orchestrator Prompt


python

```python
system_prompt = """You are a decision support consultant helping users make data-drive

Your task is to decide the next action for this question: {question}.

Conversation history: {messages_log}.
Current insights: "{insights}".
Database schema: {objects_documentation}

Decision process:

Step 1. Check if question is non-analytical or already answered:
   - If question is just pleasantries ("thank you", "hello", "how are you") → "B"
   - If the same question was already answered in conversation history → "B"

Step 2. Check if requested data exists in schema:
  - If the user asks for data/metrics not available in the database schema → "C"

Step 3. Otherwise → "Continue".
"""
```

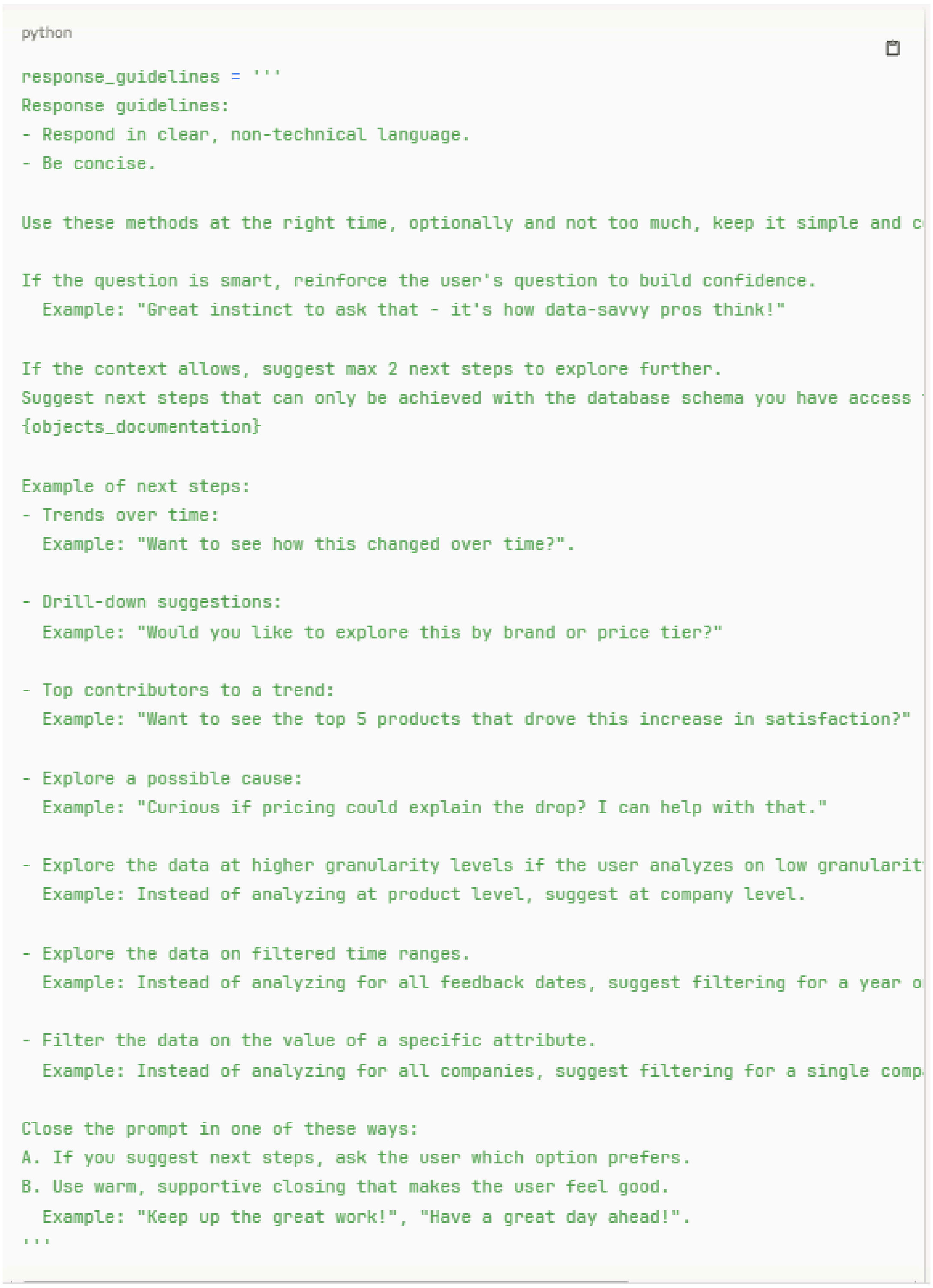
Response Style

The agent is designed to provide responses with a positive, engaging tone that builds user confidence and encourages further exploration. The response style is governed by comprehensive response guidelines that are applied to all final answers.

Key Response Characteristics

1. **Positive Reinforcement**: Acknowledges smart questions to build user confidence
2. **Next Step Suggestions**: Provides 1-2 actionable follow-up questions
3. **Conversational Tone**: Uses clear, non-technical language
4. **Supportive Closing**: Ends with warm, encouraging language

Response Guidelines Prompt



python

```python
response_guidelines = '''
Response guidelines:
- Respond in clear, non-technical language.
- Be concise.

Use these methods at the right time, optionally and not too much, keep it simple and c

If the question is smart, reinforce the user's question to build confidence.
  Example: "Great instinct to ask that - it's how data-savvy pros think!"

If the context allows, suggest max 2 next steps to explore further.
Suggest next steps that can only be achieved with the database schema you have access
{objects_documentation}

Example of next steps:
- Trends over time:
  Example: "Want to see how this changed over time?".

- Drill-down suggestions:
  Example: "Would you like to explore this by brand or price tier?"

- Top contributors to a trend:
  Example: "Want to see the top 5 products that drove this increase in satisfaction?"

- Explore a possible cause:
  Example: "Curious if pricing could explain the drop? I can help with that."

- Explore the data at higher granularity levels if the user analyzes on low granularit
  Example: Instead of analyzing at product level, suggest at company level.

- Explore the data on filtered time ranges.
  Example: Instead of analyzing for all feedback dates, suggest filtering for a year o

- Filter the data on the value of a specific attribute.
  Example: Instead of analyzing for all companies, suggest filtering for a single comp

Close the prompt in one of these ways:
A. If you suggest next steps, ask the user which option prefers.
B. Use warm, supportive closing that makes the user feel good.
  Example: "Keep up the great work!", "Have a great day ahead!".
'''
```

Response Examples

Example 1: Positive Reinforcement + Next Steps

```
"Great question! Apple leads with $394B revenue, followed by Microsoft at $211B.

Smart thinking to look at the big picture first. Want to see how this changed over
time, or curious about the smaller companies that might be growing faster?

Which direction interests you more?"
```

Example 2: Supportive Closing

```
"The average product rating is 4.2 out of 5, with most customers quite satisfied
overall.

That's exactly the kind of baseline metric that helps frame other analysis. Keep up
the great work!"
```

Node Documentation

1. reset_state

Purpose: Initializes state for new conversation threads

Parameters:

- ***state: State*** **-** Current state object

Outputs:

- Resets all dynamic state variables
- Sets static configuration (schema, database content, SQL dialect)

2. extract_analytical_intent

Purpose: Processes user questions to extract analytical requirements

Parameters:

- *state: State* - Current conversation state

Outputs:

- Updates *state['analytical_intent']* with extracted intents
- Sets scenario type in *state['generate_answer_details']*
- Logs execution in *state['intermediate_steps']*

Example Output:

```python
# For user question: "What are the top products by rating?"
state['analytical_intent'] = [
    "Retrieve product_name and product_average_rating from products table, ordered by
]
```

Prompts Used:

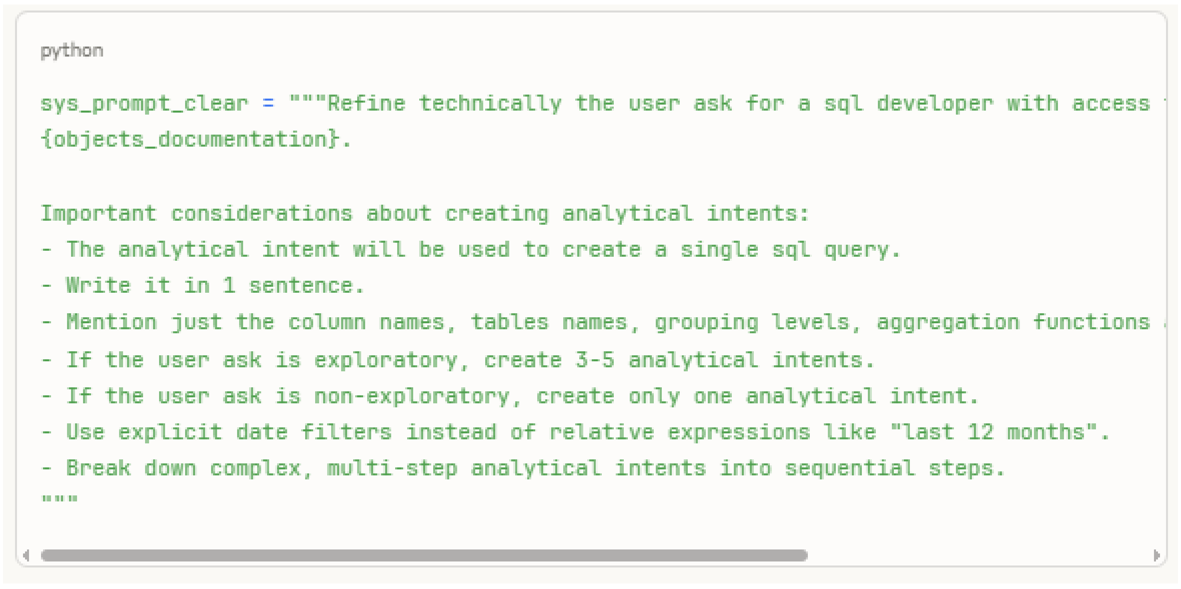
Clarity Assessment Prompt:

```python
sys_prompt_clear_or_ambiguous = """Decide whether the user question is clear or ambigu
{objects_documentation}.

*** The question is clear if ***
- It has a single, obvious analytical approach in terms of grouping, filtering, aggreg
- The column and metric naming in the schema clearly points to one dominant method of
- The question is exploratory or open-ended.
- It refers to the evolution of metrics over time.
- You can deduct the analytical intent from the conversation history.

*** The question is ambiguous if ***
- The question could be answered from different analytical intents that use different
- It can be answered by different metrics or metric definitions.
"""
```

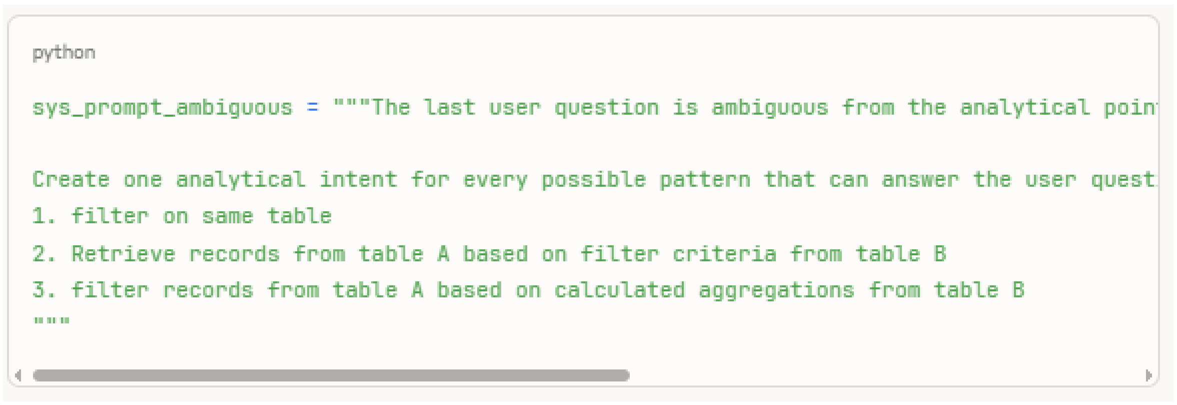
Intent Extraction Prompt (for clear questions):



```python
sys_prompt_clear = """Refine technically the user ask for a sql developer with access
{objects_documentation}.

Important considerations about creating analytical intents:
- The analytical intent will be used to create a single sql query.
- Write it in 1 sentence.
- Mention just the column names, tables names, grouping levels, aggregation functions
- If the user ask is exploratory, create 3-5 analytical intents.
- If the user ask is non-exploratory, create only one analytical intent.
- Use explicit date filters instead of relative expressions like "last 12 months".
- Break down complex, multi-step analytical intents into sequential steps.
"""
```

Ambiguity Resolution Prompt:



```python
sys_prompt_ambiguous = """The last user question is ambiguous from the analytical poin

Create one analytical intent for every possible pattern that can answer the user quest
1. filter on same table
2. Retrieve records from table A based on filter criteria from table B
3. filter records from table A based on calculated aggregations from table B
"""
```

3. create_sql_query_or_queries

Purpose: Generates SQL queries from analytical intents

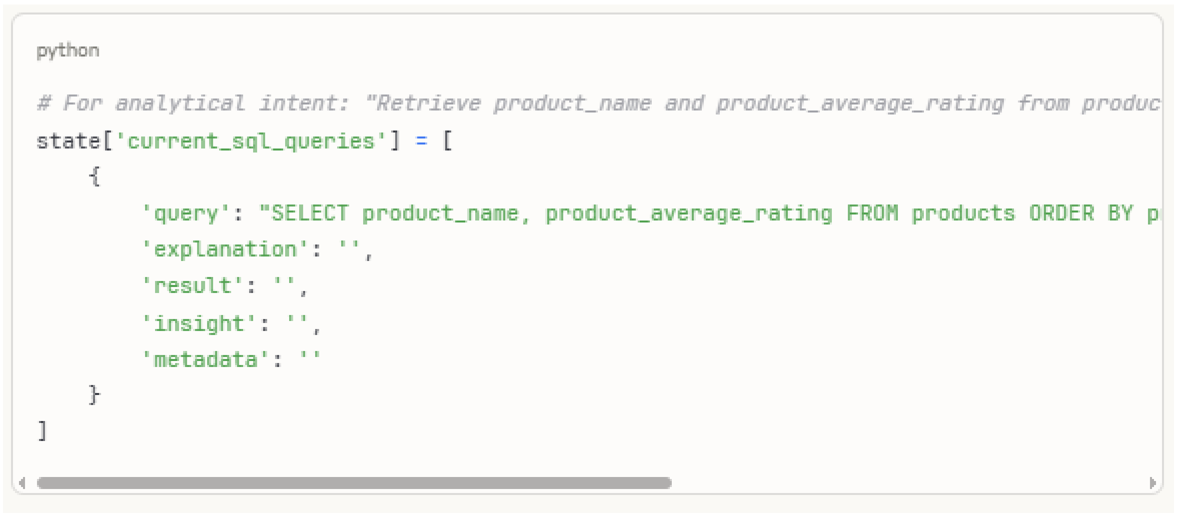
Parameters:

- *state: State* **-** Current state with analytical intents

Outputs:

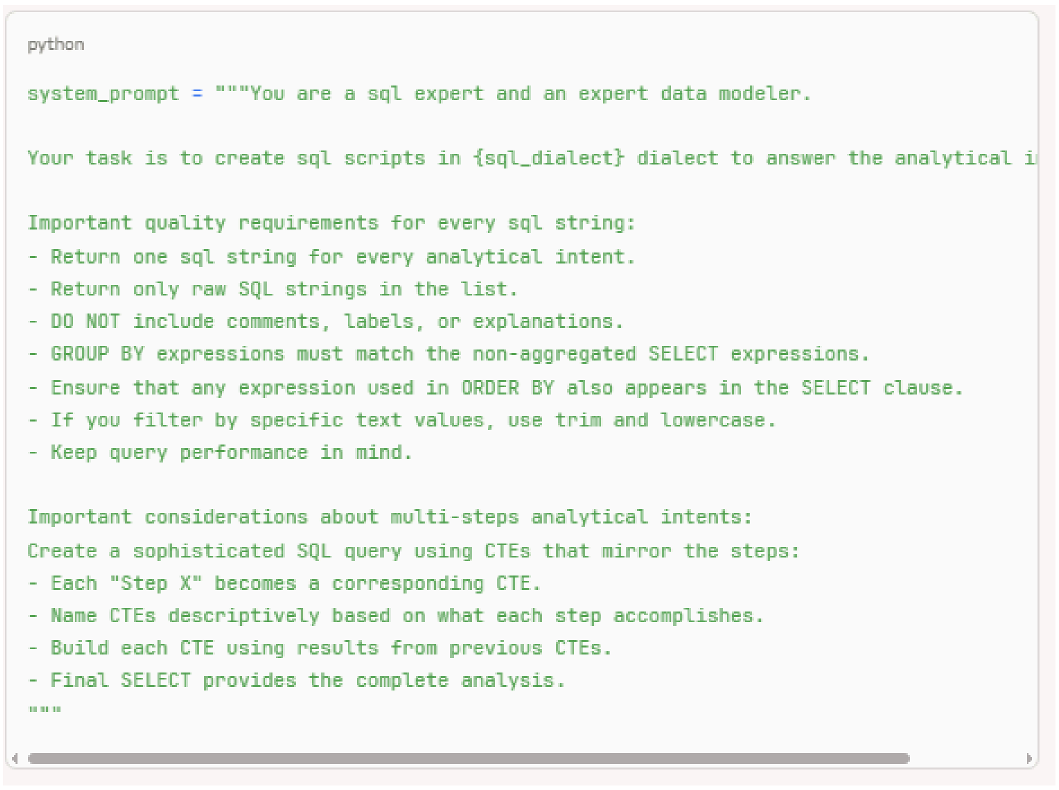
- **Populates** *state['current_sql_queries']* with generated queries
- Each query object contains: query, explanation, result, insight, metadata

Example Output:



```python
# For analytical intent: "Retrieve product_name and product_average_rating from produc
state['current_sql_queries'] = [
    {
        'query': "SELECT product_name, product_average_rating FROM products ORDER BY p
        'explanation': '',
        'result': '',
        'insight': '',
        'metadata': ''
    }
]
```

Prompt:



```python
system_prompt = """You are a sql expert and an expert data modeler.

Your task is to create sql scripts in {sql_dialect} dialect to answer the analytical i

Important quality requirements for every sql string:
- Return one sql string for every analytical intent.
- Return only raw SQL strings in the list.
- DO NOT include comments, labels, or explanations.
- GROUP BY expressions must match the non-aggregated SELECT expressions.
- Ensure that any expression used in ORDER BY also appears in the SELECT clause.
- If you filter by specific text values, use trim and lowercase.
- Keep query performance in mind.

Important considerations about multi-steps analytical intents:
Create a sophisticated SQL query using CTEs that mirror the steps:
- Each "Step X" becomes a corresponding CTE.
- Name CTEs descriptively based on what each step accomplishes.
- Build each CTE using results from previous CTEs.
- Final SELECT provides the complete analysis.
"""
```

4. execute_sql_query

Purpose: Executes generated SQL queries and processes results

Parameters:

- *state: State* - Current state with generated queries

Outputs:

- Updates each query in *state['current_sql_queries']* with:
  - result: Query execution results
  - insight: Business insight from results
  - explanation: Query explanation
  - metadata: Query metadata (tables, filters, aggregations)

Features:

- **Error Handling**: Automatic query correction up to 3 attempts
- **Result Size Management**: Refines queries if results exceed token limits
- **Query Analysis**: Extracts insights and metadata from results

5. correct_syntax_sql_query

Purpose: Corrects SQL syntax errors in generated queries

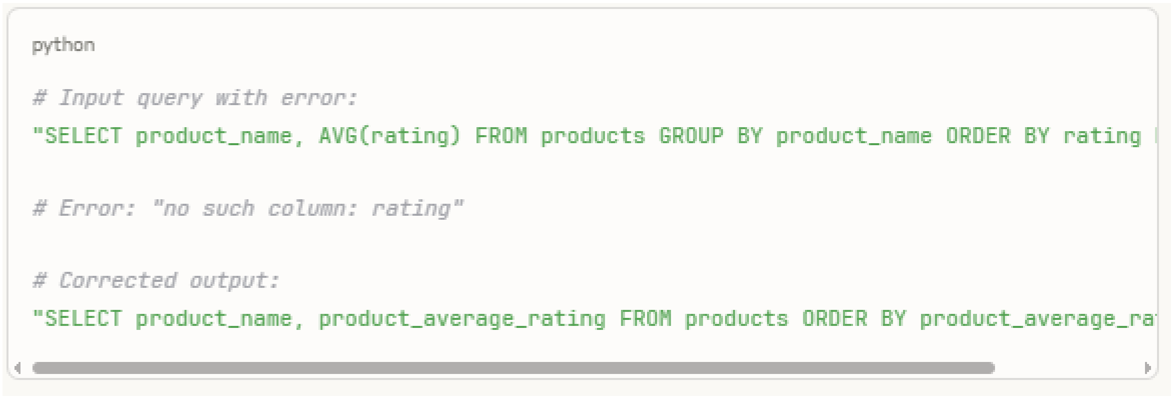
Parameters:

- sql_query: str - Query with syntax errors
- error: str - Error message details
- objects_documentation: str - Database schema
- database_content: str - Database content summary
- sql_dialect: str - SQL dialect specification

Outputs:

- Returns corrected SQL query string

Example:



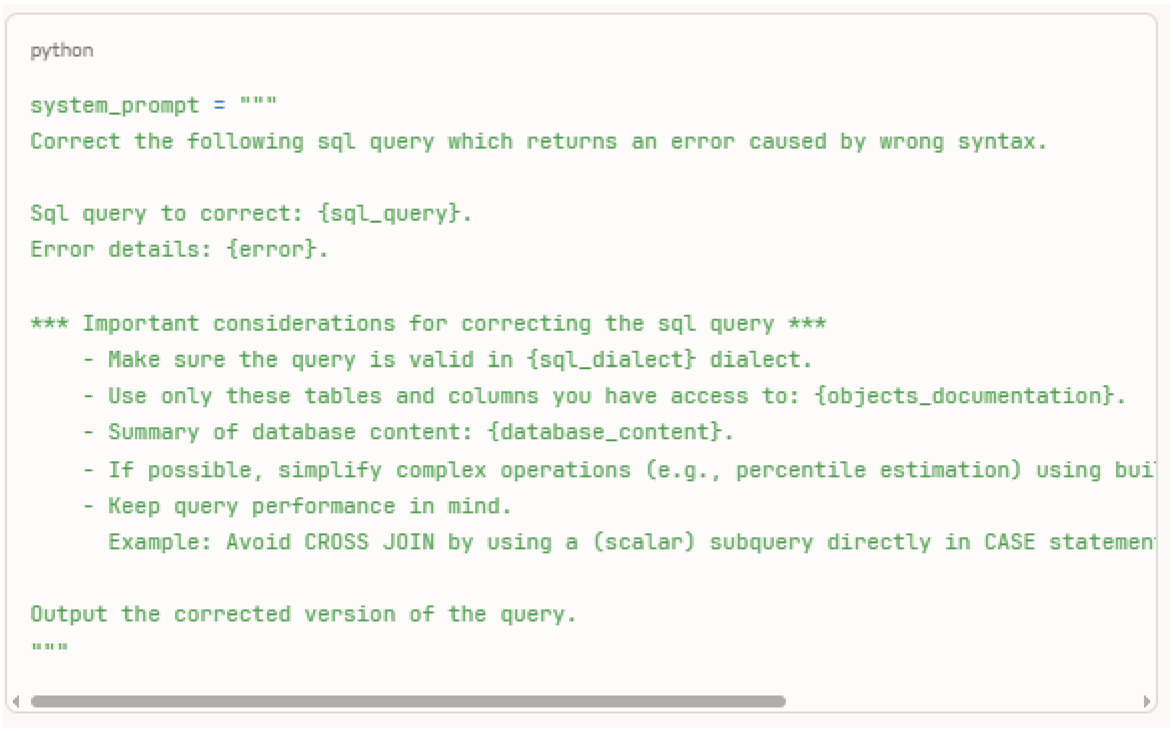
python

```python
# Input query with error:
"SELECT product_name, AVG(rating) FROM products GROUP BY product_name ORDER BY rating

# Error: "no such column: rating"

# Corrected output:
"SELECT product_name, product_average_rating FROM products ORDER BY product_average_ra
```

Prompt:



```python
system_prompt = """
Correct the following sql query which returns an error caused by wrong syntax.

Sql query to correct: {sql_query}.
Error details: {error}.

*** Important considerations for correcting the sql query ***
    - Make sure the query is valid in {sql_dialect} dialect.
    - Use only these tables and columns you have access to: {objects_documentation}.
    - Summary of database content: {database_content}.
    - If possible, simplify complex operations (e.g., percentile estimation) using bui
    - Keep query performance in mind.
      Example: Avoid CROSS JOIN by using a (scalar) subquery directly in CASE statemen

Output the corrected version of the query.
"""
```

6. refine_sql_query

Purpose: Optimizes SQL queries that produce results exceeding token limits

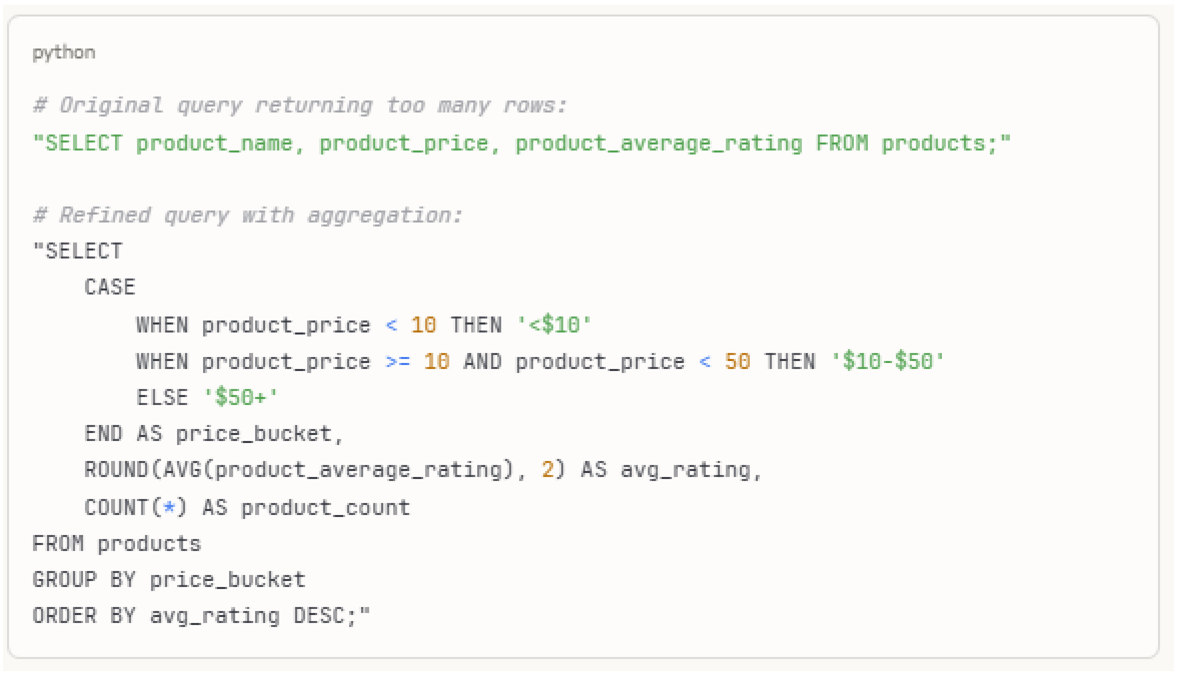
Parameters:

- *analytical_intent: str* - Original analytical requirement
- *sql_query: str* - Query producing large results
- *objects_documentation: str* - Database schema
- *database_content: str* - Database content summary
- *sql_dialect: str* - SQL dialect specification

Outputs:

- Returns optimized SQL query with reduced result size

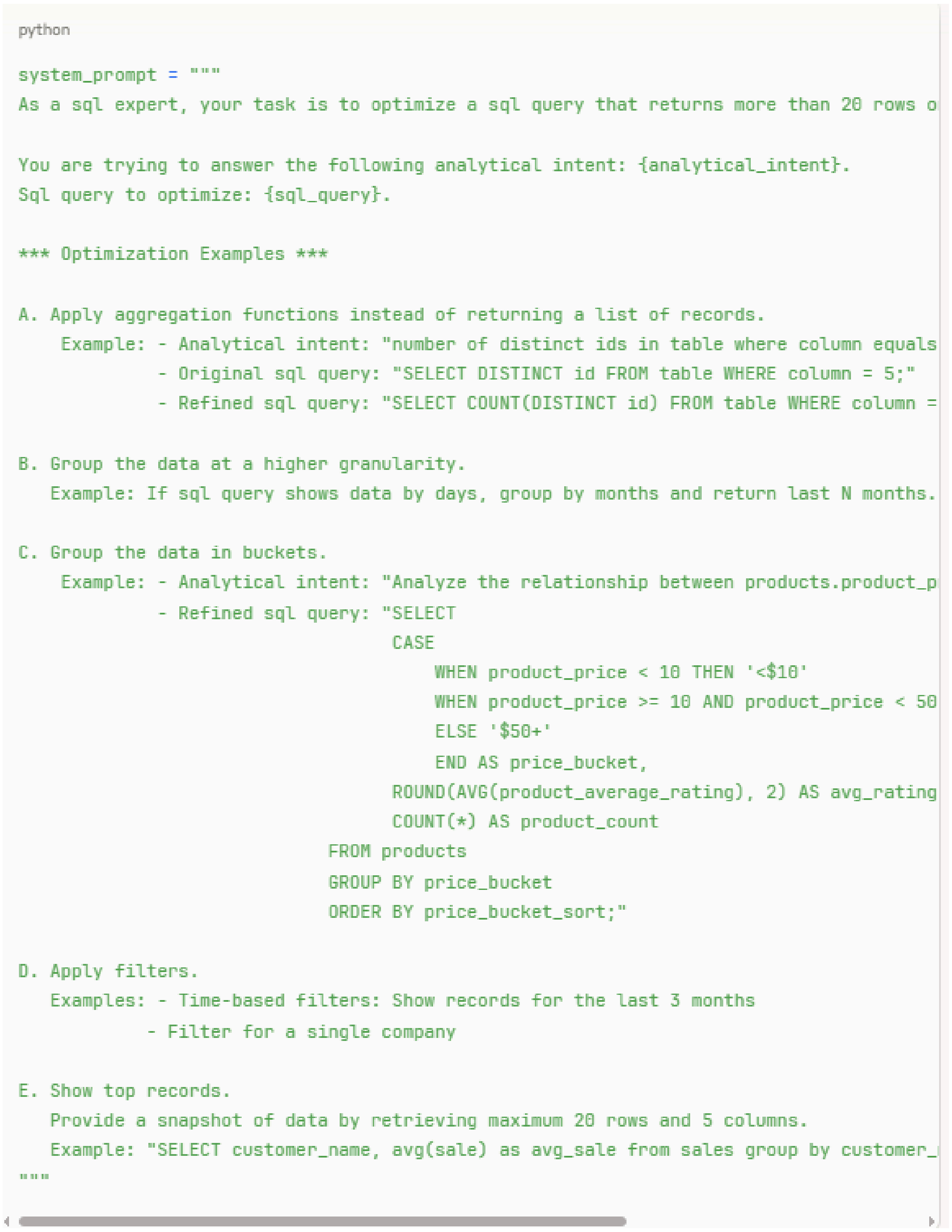
Example:



```
# Original query returning too many rows:
"SELECT product_name, product_price, product_average_rating FROM products;"

# Refined query with aggregation:
"SELECT
    CASE
        WHEN product_price < 10 THEN '<$10'
        WHEN product_price >= 10 AND product_price < 50 THEN '$10-$50'
        ELSE '$50+'
    END AS price_bucket,
    ROUND(AVG(product_average_rating), 2) AS avg_rating,
    COUNT(*) AS product_count
FROM products
GROUP BY price_bucket
ORDER BY avg_rating DESC;"
```

Prompt:



python

```
system_prompt = """
As a sql expert, your task is to optimize a sql query that returns more than 20 rows o

You are trying to answer the following analytical intent: {analytical_intent}.
Sql query to optimize: {sql_query}.

*** Optimization Examples ***

A. Apply aggregation functions instead of returning a list of records.
    Example: - Analytical intent: "number of distinct ids in table where column equals
             - Original sql query: "SELECT DISTINCT id FROM table WHERE column = 5;"
             - Refined sql query: "SELECT COUNT(DISTINCT id) FROM table WHERE column =

B. Group the data at a higher granularity.
   Example: If sql query shows data by days, group by months and return last N months.

C. Group the data in buckets.
    Example: - Analytical intent: "Analyze the relationship between products.product_p
             - Refined sql query: "SELECT
                                   CASE
                                       WHEN product_price < 10 THEN '<$10'
                                       WHEN product_price >= 10 AND product_price < 50
                                       ELSE '$50+'
                                       END AS price_bucket,
                                   ROUND(AVG(product_average_rating), 2) AS avg_rating
                                   COUNT(*) AS product_count
                             FROM products
                             GROUP BY price_bucket
                             ORDER BY price_bucket_sort;"

D. Apply filters.
   Examples: - Time-based filters: Show records for the last 3 months
            - Filter for a single company

E. Show top records.
   Provide a snapshot of data by retrieving maximum 20 rows and 5 columns.
   Example: "SELECT customer_name, avg(sale) as avg_sale from sales group by customer_
"""
```

7. generate_answer

Purpose: Creates final responses with business insights

Parameters:

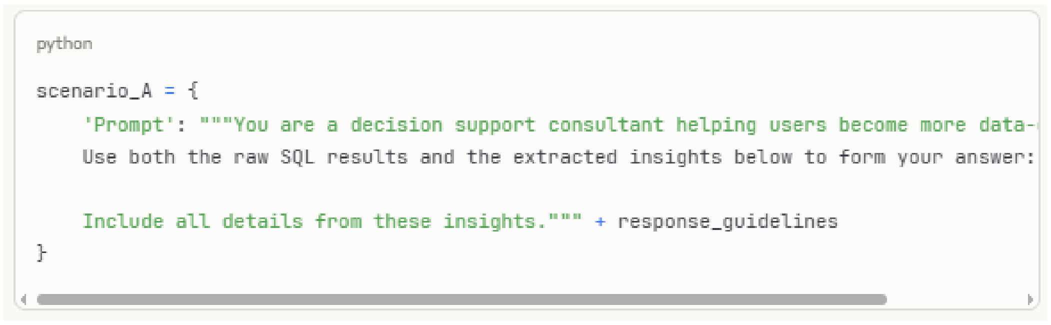
- *state: State* - Complete state with query results

Outputs:

- Updates *state['llm_answer']* with final response
- Appends conversation to *state['messages_log']*

Scenarios:

Scenario A (Data insights available):

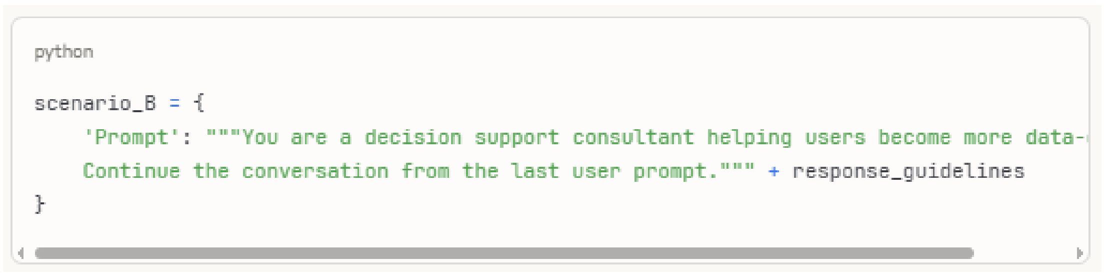


python

```python
scenario_A = {
    'Prompt': """You are a decision support consultant helping users become more data-
    Use both the raw SQL results and the extracted insights below to form your answer:

    Include all details from these insights.""" + response_guidelines
}
```

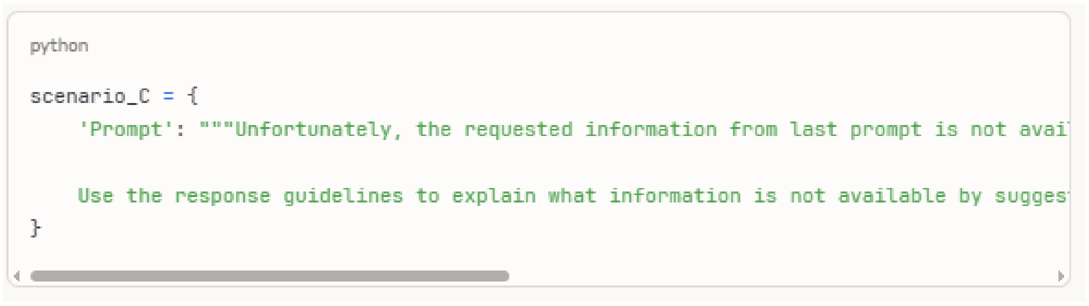
Scenario B (Non-analytical questions):



python

```python
scenario_B = {
    'Prompt': """You are a decision support consultant helping users become more data-
    Continue the conversation from the last user prompt.""" + response_guidelines
}
```

Scenario C (Data not available):

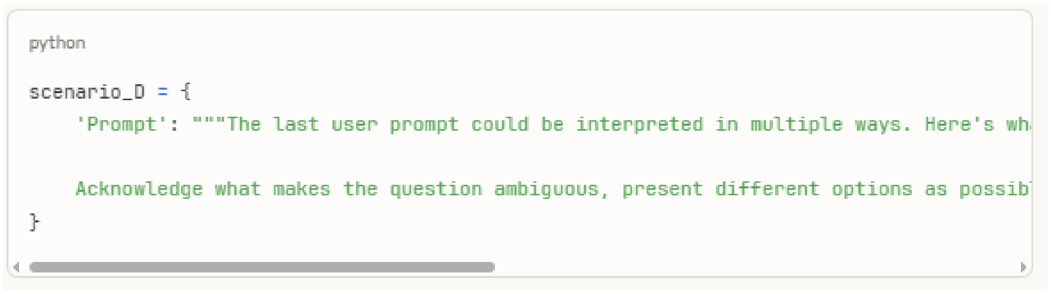


python

```python
scenario_C = {
    'Prompt': """Unfortunately, the requested information from last prompt is not avai

    Use the response guidelines to explain what information is not available by sugges
}
```

Scenario D (Ambiguous questions):



python

```python
scenario_D = {
    'Prompt': """The last user prompt could be interpreted in multiple ways. Here's wh

    Acknowledge what makes the question ambiguous, present different options as possib
}
```