Create an Agent using LLM and custom mathematical functions

i. Brief Explanation of LLM Agent Creation

How the Agent Works

This agent is a **conditional routing system** built using LangGraph that intelligently decides whether to handle mathematical queries locally or route general questions to Google's Gemini LLM. The agent operates on a simple but effective principle:

1. **Input Analysis**: Every user query is first analyzed to determine if it contains mathematical operations
2. **Conditional Routing**: Based on the analysis, the agent routes to either:
    - **Math Processing**: For arithmetic operations (addition, subtraction, multiplication, division)
    - **LLM Processing**: For general knowledge questions, conversations, and complex queries

Agent Architecture

User Input → Router Node → [Math Node OR LLM Node] → Output

The agent leverages:

- **Local computation** for simple math (fast, reliable, no API costs)
- **Gemini Pro LLM** for complex reasoning, general knowledge, and natural language tasks
- **State management** to maintain context and data flow between processing nodes

---

ii. Code Structure and Program Flow

1. Dependencies and Setup

Core libraries

import google.generativeai as genai # Gemini LLM integration

from langgraph.graph import StateGraph, END # Graph-based workflow

import re # Pattern matching for math detection

Purpose: Establishes the foundation with Google's Generative AI SDK and LangGraph for workflow orchestration.

2. Mathematical Operations Module

```
def plus (a, b): return a + b

def subtract (a, b): return a - b

def multiply (a, b): return a * b

def divide (a, b): return "Error: Division by zero." if b == 0 else a / b
```

Flow: These pure functions handle basic arithmetic with built-in error handling for edge cases like division by zero.

3. Query Parsing Engine

```
def parse_math_query(text):

    text = text.lower()

    patterns = [

        (r'(\d+) \s*(\+|plus) \s*(\d+)', plus),

        (r'(\d+) \s*(-|minus|subtract) \s*(\d+)', subtract),

        (r'(\d+) \s*(\*|x|times|multiply) \s*(\d+)', multiply),

        (r'(\d+) \s*(/|divided by|divide) \s*(\d+)', divide)

    ]
```

Logic Flow:

1. Convert input to lowercase for consistent matching
2. Define regex patterns that capture:
    - Numbers before and after operators
    - Multiple operator representations (+, plus, times, x, etc.)
3. Return the appropriate function and extracted numbers

4. State Management System

```
class Graph State(TypedDict):

    input: str      # Original user query

    output: str     # Final response
```

```
    is_math: bool   # Classification flag
```

Purpose: Maintains data integrity and type safety throughout the graph execution pipeline.

5. Core Processing Nodes

Router Node (Decision Maker)

```
def router_node(state: GraphState) -> GraphState:
    text = state.get("input", "")
    func, a, b = parse_math_query(text)
    state["is_math"] = bool(func) # Sets routing flag
    return state
```

Flow: Analyzes input → Detects math patterns → Sets classification flag → Passes state forward

Math Node (Local Computation)

```
def math_node(state: GraphState) -> GraphState:
    text = state.get("input", "")
    func, a, b = parse_math_query(text)
    if func:
        result = func(a, b)
        state["output"] = str(result)
    else:
        state["output"] = "Invalid math expression"
    return state
```

Flow: Re-parses query → Executes mathematical function → Stores result in state

LLM Node (AI Processing)

```
def llm_node(state: GraphState) -> GraphState:
    try:
        input_text = state.get("input", "")
        response = model.generate_content(input_text)
        state["output"] = response.text
```

```
    except Exception as e:
        state["output"] = f"Error: {str(e)}"
    return state
```

Flow: Validates input → Calls Gemini API → Handles responses/errors → Updates state

6. Routing Logic

```
def router_condition(state: GraphState) -> str:
    return "math" if state.get("is_math", False) else "llm"
```

Decision Tree:

- is_math = True → Route to Math Node
- is_math = False → Route to LLM Node

7. Graph Construction and Execution

```
builder = StateGraph(GraphState)
builder.add_node("router", router_node)
builder.add_node("math", math_node)
builder.add_node("llm", llm_node)
builder.set_entry_point("router")
builder.add_conditional_edges("router", router_condition, {
    "math": "math", "llm": "llm"
})
```

Graph Structure:

START → router → [conditional_edge] → math/llm → END

8. Program Execution Flow

```
graph TD
    A[User Input] --> B[Router Node]
    B --> C{Math Detection?}
    C -->|Yes| D[Math Node]
    C -->|No| E[LLM Node]
```

D --> F[Return Result]

E --> F[Return Result]

F --> G[Display Output]

Detailed Execution Steps:

1. **Initialization**: Create initial state with user input
2. **Entry Point**: Graph starts at router node
3. **Classification**: Router analyzes input and sets is_math flag
4. **Conditional Routing**: Based on flag, graph routes to appropriate processor
5. **Processing**: Either mathematical computation or LLM generation occurs
6. **Completion**: Result stored in state and returned to user

Key Design Benefits:

- **Efficiency**: Math queries don't consume LLM tokens/API calls
- **Reliability**: Local math computation eliminates API failures for basic operations
- **Scalability**: Easy to extend with more specialized processing nodes
- **Error Handling**: Comprehensive exception management at each stage
- **Type Safety**: TypedDict ensures state integrity throughout execution

Example Flow Trace:

Input: "What is 5 plus 7?"

1. router_node: Detects "5 plus 7" pattern → Sets is_math = True
2. router_condition: Returns "math" based on flag
3. math_node: Parses query → Executes plus(5, 7) → Sets output = "12"
4. Graph terminates → Returns final state with result

This architecture demonstrates how LangGraph enables sophisticated decision-making workflows while maintaining clean separation of concerns between different processing capabilities.