💡 A Step-by-Step Coding Guide to Building an Iterative AI Workflow Agent Using LangGraph and Gemini

1. Introduction: The Advent of Iterative AI Agents

The landscape of Artificial Intelligence (AI) is rapidly evolving, with increasingly sophisticated applications demanding more than simple input-output processing. Modern AI systems, particularly those involving Large Language Models (LLMs), often

need to perform complex reasoning, engage in multi-step problem-solving, and refine their outputs based on intermediate assessments. This mirrors human cognitive processes, where tasks are broken down, information is gathered, solutions are proposed, and critiques lead to improvements. This paradigm of cyclical refinement is at the heart of iterative AI agents.

LangGraph has emerged as a powerful library for building such stateful, multi-actor applications with LLMs.[1] It extends the popular LangChain ecosystem, providing developers with the tools to construct complex workflows as graphs. Unlike traditional LangChain "chains," which are often linear (Directed Acyclic Graphs or DAGs), LangGraph excels at creating graphs with cycles, enabling agents to loop, re-evaluate, and iteratively improve their work.[2] This capability is crucial for tasks requiring deliberation, self-correction, or dynamic adaptation based on evolving information.

Google's Gemini models represent the cutting edge of LLM technology, offering robust capabilities in text generation, understanding, and, in some versions, multimodal processing.[4] The Gemini API provides developers access to these models, allowing for the integration of advanced AI reasoning into custom applications.[6] When combined, LangGraph's structural control and Gemini's cognitive power offer a potent toolkit for creating sophisticated AI agents.

This guide provides a comprehensive, step-by-step walkthrough for building an **iterative AI workflow agent** using LangGraph and the Gemini 1.5 Flash model. The agent will be designed to intelligently handle user queries by passing them through a series of purposeful nodes: routing, analysis, research, response generation, and validation.[8] The core of this agent is its ability to loop and re-evaluate its output, aiming for completeness and accuracy, demonstrating a practical application of iterative AI. This document will cover the foundational concepts, setup, design, detailed Python implementation, and methods for testing and observing the agent's behavior, including full code examples and notebook-style representations.

2. Core Concepts: LangGraph and Iterative AI

To effectively build iterative AI agents, understanding the fundamental components of LangGraph is essential. LangGraph structures AI logic as a graph, where data flows between nodes that perform specific actions, and edges direct this flow, potentially creating cycles for iteration.[9]

2.1. State: The Agent's Working Memory

The **State** is a central concept in LangGraph. It represents a shared data structure that persists and evolves as the agent processes information and executes tasks.[9] This state can be any Python type, but commonly, it's a TypedDict or a Pydantic BaseModel, allowing for structured data management.[9] Each node in the graph receives the current state as input and can return updates to this state.

The significance of the state object cannot be overstated; it acts as the agent's working memory. As the agent iterates through cycles of analysis, action, and reflection, the state object accumulates information, stores intermediate results, and tracks the overall progress. This evolving state allows subsequent operations within the agent to be more informed, building upon previous steps and enabling a form of learning and adaptation within a single execution run. For instance, if an initial response is deemed insufficient by a validation step, this feedback can be recorded in the state, guiding the agent to refine its approach in the next iteration.

2.2. Nodes: The Agent's Actions

Nodes are the building blocks of a LangGraph application, representing the actions or computations the agent can perform.[9] In Python, these are typically functions or LangChain Expression Language (LCEL) runnables. Each node takes the current State as input and, after performing its logic (which might involve calling an LLM, executing a tool, or performing data manipulation), returns a dictionary containing updates to be applied to the State.[9] Nodes do not need to return the entire state, only the parts they have modified.

The modularity of nodes is a key strength. Each node can encapsulate a specific piece of logic—like analyzing a query, researching a topic, or generating a response. This separation of concerns makes the overall agent easier to design, debug, and extend.

2.3. Edges: Directing the Agent's Flow

Edges define the connections between nodes, dictating the sequence of operations within the graph.[9] LangGraph supports several types of edges:

- **Normal Edges**: These create a fixed path from one node to another. For example, after a "research" node completes, a normal edge might always direct the flow to a "response generation" node.
- **Conditional Edges**: These introduce branching logic. A function is provided that examines the current State and decides which node (or nodes) to execute next. This is crucial for creating iterative loops, where the agent might decide to re-run a previous step or exit based on certain conditions. For instance, a conditional edge after a "validation" node could loop back to an "analysis" node if the

response needs improvement, or proceed to an END state if the response is satisfactory.[8]
- **Entry Point**: This special edge specifies the first node to be called when the graph receives input.[9]
- There's also a special END node, signifying the termination of the graph's execution for a given run.[10]

The interplay of state, nodes, and particularly conditional edges is what allows LangGraph to facilitate cyclical, agentic workflows. An agent can process input, update its state, and then, based on that state, a conditional edge can route the execution back to an earlier node with the enriched state. This creates a loop where the agent can refine its understanding, gather more data, or improve its output over multiple iterations until a desired condition is met or a maximum number of attempts is reached. This cyclical capability is a hallmark of LangGraph and a fundamental requirement for building truly iterative AI systems.[2]

2.4. StateGraph: Orchestrating the Workflow

The StateGraph class is the primary mechanism for constructing these graphs in LangGraph.[10] It is initialized with a state definition (e.g., a TypedDict or a dataclass representing the AgentState). Developers then add nodes and edges to this StateGraph instance. Finally, the compile() method is called on the StateGraph object, which finalizes the graph structure, performs some validation checks (like ensuring no orphaned nodes), and returns an executable application.[9] This compiled application can then be invoked with inputs to run the defined workflow.

3. Prerequisites and Setup

Before diving into the construction of the iterative AI agent, several prerequisites must be met. These include setting up the Python environment, installing the necessary libraries, and acquiring and configuring a Google Gemini API key.

3.1. Python Environment

A Python environment (version 3.9 or newer is generally recommended for modern AI libraries) is required. Using a virtual environment is a best practice to manage dependencies and avoid conflicts:

Bash

```
# Create a virtual environment (example using venv)
python -m venv langgraph-gemini-agent-env

# Activate the virtual environment
# On macOS and Linux:
source langgraph-gemini-agent-env/bin/activate
# On Windows:
# langgraph-gemini-agent-env\Scripts\activate
```

3.2. Installing LangGraph and LangChain-Google-Genai

The core libraries needed are langgraph for building the agentic workflow, langchain-google-genai for interacting with Gemini models, python-dotenv for managing API keys securely, and langchain-core which provides foundational components.

The following command installs these packages [7]:

Bash

```
pip install langgraph langchain-google-genai python-dotenv langchain-core
```

It's important to note that langgraph itself is installed via pip install langgraph.[12] The command above includes it along with other necessary packages. The langchain-google-genai package specifically provides the interface to Google's Gemini models through LangChain.[7]

3.3. Acquiring and Managing Your Google Gemini API Key

Access to Gemini models via their API requires an API key. This key authenticates requests to the Gemini service.

Steps to Obtain a Gemini API Key:

1. **Login to Google Account**: Ensure you are logged into your Google account.[13]
2. **Visit Google AI Studio**: Navigate to the Google AI Studio website (e.g., ai.google.dev or aistudio.google.com).[13]
3. **Navigate to API Key Generation**: Look for options like "Get API key" or "Gemini API" and then a button such as "Get API key in Google AI Studio" or "Create API

key".[13]
4. **Agree to Terms**: Review and accept the Google APIs Terms of Service and any Gemini-specific terms.[13]
5. **Create and Copy Key**: Choose to create an API key, potentially within a new or existing Google Cloud project. Once generated, copy the API key immediately.[13]

Securely Managing the API Key:

API keys are sensitive credentials. They should never be hardcoded directly into source code or committed to version control systems like Git.[6] A common and recommended practice is to use environment variables, often managed with a .env file.

1. **Create a .env file**: In the root directory of the project, create a file named .env.
2. **Add API Key to .env**:
   `GOOGLE_API_KEY="YOUR_ACTUAL_GEMINI_API_KEY_HERE"`

   Replace "YOUR_ACTUAL_GEMINI_API_KEY_HERE" with the key obtained from Google AI Studio.
3. **Add .env to .gitignore**: Ensure that .env is listed in the project's .gitignore file to prevent it from being tracked by Git.
4. **Load Environment Variables in Python**: Use the python-dotenv library to load this variable into the application's environment at runtime.

```Python
import os
from dotenv import load_dotenv

load_dotenv()
google_api_key = os.getenv("GOOGLE_API_KEY")

# The langchain-google-genai library typically picks up the GOOGLE_API_KEY
# from the environment automatically if it's set.
# Alternatively, one could explicitly set it for the OS environment if needed by a library:
# os.environ["GOOGLE_API_KEY"] = "YOUR_ACTUAL_API_KEY_HERE" # [8]
```

This approach ensures that the API key is kept separate from the codebase, enhancing security.[6] The repeated emphasis on securing API keys across various resources underscores this as a critical best practice.[6] Neglecting this can lead to unauthorized API usage and potential costs.

While this guide focuses on API keys from Google AI Studio for ease of developer

access [6], it's worth noting that Gemini models can also be accessed via Google Cloud Vertex AI, which is often preferred for enterprise-grade applications offering more extensive MLOps capabilities.[5] For the scope of this tutorial, the AI Studio key is sufficient.

Table 1: Gemini API Setup Checklist

Step	Action	Notes	Relevant Sources
1. Google Account	Log in to your existing Google Account.	A standard Google account is required.	13
2. Access Google AI Studio	Navigate to(https://ai.google.dev/).	This is the primary portal for generating developer API keys for Gemini.	13
3. Generate API Key	Follow prompts to "Get API key" or "Create API key". This may involve creating or selecting a Google Cloud project.	The process is typically straightforward and quick.	13
4. Secure API Key	Copy the generated API key. **Do not** share it publicly or commit it to version control. Store it in a .env file.	Security is paramount. The .env file should be added to .gitignore.	6
5. Install Libraries	pip install langgraph langchain-google-genai python-dotenv langchain-core	These libraries provide the necessary tools for the agent.	7
6. Load API Key in Code (Securely)	Use python-dotenv to load the GOOGLE_API_KEY from the .env file into your Python script's environment	os.getenv("GOOGLE_API_KEY")	8

	variables.		
7. Verify Installation & Key (Optional)	Write a small script to initialize ChatGoogleGenerativeAI and make a simple call to ensure the key and library setup are working.	A quick test can save debugging time later.	[6]

4. Designing Your Iterative AI Workflow Agent

With the prerequisites in place, the next step is to design the architecture of the iterative AI agent. This involves defining its task, outlining the flow of operations, and structuring the state it will maintain across iterations.

4.1. Defining the Agent's Task: An Intelligent Query Handler

The goal is to construct an agent capable of intelligently processing a user's query through a multi-step, iterative workflow.[8] This agent should be able to:

- Understand the nature of the query (e.g., factual, creative, problem-solving).[8]
- Analyze if it has enough information or if further research is needed.
- Conduct (simulated) research if necessary.
- Generate a comprehensive response.
- Validate its own response for quality and completeness.
- Iterate on the process if the response is deemed insufficient, up to a certain limit.

This design aims to create an agent that doesn't just give a one-shot answer but can refine its output, much like a human expert might review and improve their work.

4.2. Architecting the Iterative Loop: Analysis, Research, Response, Validation

The agent's workflow will be structured as a graph with the following key nodes and conditional branches [8]:

1. **Input**: The process begins with a user query.
2. **Router Node**: This node performs an initial analysis of the query, categorizing it and potentially providing some initial contextualization.
3. **Analyzer Node**: This node takes the query and any existing context (from the router or previous iterations) and determines the next best course of action. It decides whether the agent can proceed directly to generating a response or if further research is required.
4. **Conditional Branch 1 (Post-Analyzer)**:

    - If the Analyzer determines that research is needed, the flow proceeds to the **Researcher Node**.
    - If the Analyzer determines that a direct response is possible, the flow proceeds to the **Responder Node**.
5. **Researcher Node**: If activated, this node simulates a research step. In a more advanced agent, this could involve calling external tools (like web search APIs). For this guide, it will use an LLM call to generate relevant information based on the analysis. The gathered information is added to the agent's context. After research, the flow moves to the Responder Node.
6. **Responder Node**: This node generates the actual response to the user's query, using all available information (original query, context from router/researcher, and analysis).
7. **Validator Node**: After a response is generated, this node evaluates its quality, accuracy, and completeness against the original query. It will output a judgment, such as "COMPLETE" or "NEEDS_IMPROVEMENT".
8. **Conditional Branch 2 (Post-Validator - The Iterative Loop)**:
    - If the Validator deems the response "COMPLETE", or if a maximum number of iterations has been reached, the workflow terminates, and the final response is output.
    - If the Validator deems the response "NEEDS_IMPROVEMENT" (and the iteration limit has not been reached), the workflow loops back to the **Analyzer Node**. The context now includes the previous response and the validator's feedback, allowing the agent to refine its approach in the next iteration.

This architecture explicitly models a deliberative process. The sequence—Router → Analyzer → (Optional Researcher) → Responder → Validator—is not arbitrary. It reflects a common human strategy for tackling complex questions: first understanding the question's nature (Router), then assessing existing knowledge and identifying gaps (Analyzer). If information is lacking, research is conducted (Researcher). An answer is then formulated (Responder), and finally, it's reviewed for quality (Validator). The agent's design, as derived from the approach in [8] and [8], directly maps to this cognitive workflow, enhancing its potential for sophisticated reasoning.

(A visual diagram of this architecture would typically be inserted here. For this text-based output, imagine a flowchart depicting these nodes and conditional flows.)

4.3. Structuring the AgentState: What to Track Across Iterations

The AgentState is the backbone of the iterative process, carrying information between

nodes and across cycles. It needs to track several key pieces of data [8]:

- query: str: The original, unmodified query from the user. This remains constant throughout the agent's run for a given input.
- context: str: An evolving string that accumulates information. It can start with initial context from the router, be augmented by findings from the researcher, and include feedback from the validator. This field acts as a shared scratchpad or an evolving knowledge base for the agent. Its cumulative nature means that in subsequent iterations, the analyzer and responder have access to a richer set of information, including past attempts and critiques, enabling more informed decisions and refined outputs.
- analysis: str: The output from the analyzer node, detailing its assessment of the query and current context, and potentially suggesting a plan.
- response: str: The LLM-generated response to the query from the responder node. This field is updated in each iteration where a new response is formulated.
- next_action: str: A field set by the analyzer node to guide the first conditional edge (e.g., "research" or "respond").
- iteration: int: A counter for the current loop. This is crucial for controlling the flow and for the termination condition.
- max_iterations: int: A predefined limit on the number of iterations the agent can perform. This is an essential safeguard. While the goal is intelligent completion based on the validator's output, cyclical graphs always carry the risk of unintended infinite loops, especially when conditional logic is driven by LLM outputs which can have variability. The max_iterations field provides a pragmatic mechanism to ensure the agent always terminates, preventing runaway processes and associated computational costs.[8]

Each of these fields plays a vital role. For example, the context allows the agent to build upon previous findings, while iteration and max_iterations ensure the process eventually concludes. The AgentState will be defined in Python using a dataclass or a TypedDict.

5. Step-by-Step Implementation: Coding Your LangGraph & Gemini Agent

This section details the Python implementation of the iterative AI agent, translating the design into functional code. It covers setting up imports, initializing the Gemini LLM, defining the agent's state, coding each node, crafting the conditional logic for iteration, and finally, building and compiling the graph.

5.1. Setup: Imports and API Key Configuration

First, the necessary Python libraries are imported. The GOOGLE_API_KEY is loaded from the environment, as configured in the prerequisites.

Python

```
import os
from dotenv import load_dotenv
from typing import TypedDict, Any, Dict, List
from dataclasses import dataclass, field

from langchain_Genai import ChatGoogleGenerativeAI
from langchain_core.messages import HumanMessage, SystemMessage
from langgraph.graph import StateGraph, START, END
from langgraph.checkpoint.sqlite import SqliteSaver # Optional: for persistence

# Load environment variables from.env file
load_dotenv()

# Ensure the GOOGLE_API_KEY is available in the environment
# The ChatGoogleGenerativeAI class will typically pick it up automatically.
# if not os.getenv("GOOGLE_API_KEY"):
#     raise ValueError("GOOGLE_API_KEY not found in environment variables. Please set it in your.env file.")
```

5.2. Initializing the Gemini LLM for Different Roles

The Gemini 1.5 Flash model will be used. It's beneficial to instantiate potentially two different configurations of the LLM: one for more creative or generative tasks (like routing, research, and response generation) and another for tasks requiring more deterministic and analytical output (like analysis and validation), typically by setting a lower temperature.[8] A lower temperature (e.g., 0.1) makes the output more focused and less random, while a higher temperature (e.g., 0.7) allows for more creativity. This separation allows for finer control over the LLM's behavior in different parts of the agent's workflow, contributing to better overall performance.

```
Python
```

```
# Initialize Gemini LLMs
# Using gemini-1.5-flash-latest as per [8]/[8] for Gemini 1.5 Flash
llm_model_name = "gemini-1.5-flash-latest"

# LLM for general generation tasks
llm = ChatGoogleGenerativeAI(model=llm_model_name, temperature=0.7,
convert_system_message_to_human=True)

# LLM for analysis and validation tasks (more deterministic)
analyzer_llm = ChatGoogleGenerativeAI(model=llm_model_name, temperature=0.1,
convert_system_message_to_human=True)
```

Note: The convert_system_message_to_human=True parameter can be important for some Gemini models via LangChain that might not natively support SystemMessage in the exact same way as other models, or to ensure compatibility.

5.3. Defining the AgentState

The AgentState is defined as a dataclass to hold all the information that needs to be passed between nodes and updated during the iterative process.[8]

```
Python
```

```
@dataclass
class AgentState:
    query: str
    context: str = ""
    analysis: str = ""
    response: str = ""
    next_action: str = "" # Stores decisions like "research" or "respond"
    iteration: int = 0
    max_iterations: int = 3 # Default maximum iterations
    # Optional: A list to store messages or history if building a conversational agent
    # messages: List[Any] = field(default_factory=list)
```

Alternatively, a TypedDict could be used, which is also common in LangGraph

examples.[9] For this guide, the dataclass approach from [8]/[8] is followed for clarity.

5.4. Implementing Agent Nodes

Each node in the graph is a Python function that takes the current AgentState as input and returns a dictionary of updates to the state.

Table 2: Iterative Agent Node Functions

Node Function	Purpose	Key Inputs from State	Key Outputs to State	LLM Used
_router_node	Initial query categorization and context setting.	query	context, iteration (incremented)	llm
_analyzer_node	Analyzes query/context to decide next action (research or respond).	query, context, analysis (previous)	analysis, next_action	analyzer_llm
_researcher_node	(Simulated) Gathers additional information based on analysis.	query, analysis	context (appended with research)	llm
_responder_node	Generates the final response based on all available information.	query, context, analysis	response	llm
_validator_node	Evaluates the generated response for quality and completeness.	query, response	context (appended with validation result)	analyzer_llm

1. Router Node (_router_node)

This node provides an initial interpretation or categorization of the user's query.

Python

```
def _router_node(state: AgentState) -> Dict[str, Any]:
    print(f"--- Iteration {state.iteration + 1}: Router Node ---")
    system_msg_content = """You are a query router. Analyze the user's query and provide initial context.
Determine if this is a factual question, a creative request, a problem-solving task, or an analytical task.
Briefly summarize the query's core intent."""
    messages =
    response = llm.invoke(messages)
    return {
        "context": f"Initial Router Analysis: {response.content}",
        "iteration": state.iteration + 1 # Increment iteration count
    }
```

2. Analyzer Node (_analyzer_node)
This node decides whether more information is needed (research) or if the agent can proceed to generate a response.

Python

```
def _analyzer_node(state: AgentState) -> Dict[str, Any]:
    print(f"--- Iteration {state.iteration}: Analyzer Node ---")
    system_msg_content = """You are an expert analyst. Review the user's query and the current context.
Determine if additional research is absolutely necessary to provide a comprehensive and accurate answer, or if a direct response can be formulated.
If research is needed, state 'RESEARCH_REQUIRED'. Otherwise, state 'DIRECT_RESPONSE_POSSIBLE'.
Provide a brief justification for your decision.
Current context may include previous responses and validation feedback if this is not the first iteration."""

    content_parts = [
        f"User Query: {state.query}",
        f"Current Context: {state.context}",
    ]
```

```
    if state.analysis: # Include previous analysis if available (for iterations)
        content_parts.append(f"Previous Analysis: {state.analysis}")
    if state.response: # Include previous response if available
        content_parts.append(f"Previous Response: {state.response}")

    messages =

    response = analyzer_llm.invoke(messages)
    analysis_text = response.content

    next_action = "respond" # Default action
    if "RESEARCH_REQUIRED" in analysis_text.upper():
        next_action = "research"

    print(f"Analyzer Output: {analysis_text}")
    print(f"Analyzer Decision: {next_action}")

    return {
        "analysis": analysis_text,
        "next_action": next_action
    }
```

3. Researcher Node (_researcher_node)
This node simulates a research step. In a real-world scenario, this could involve API calls to search engines or databases. Here, it uses an LLM to generate "researched" content.

```
    Python


def _researcher_node(state: AgentState) -> Dict[str, Any]:
    print(f"--- Iteration {state.iteration}: Researcher Node ---")
    system_msg_content = """You are a research assistant. Based on the user query and the
analysis provided,
gather relevant information and detailed insights. Focus on providing factual content that will help
answer the query comprehensively.
The analysis might specify areas needing more information."""
    messages =
    response = llm.invoke(messages)
    research_content = response.content
```

```
    updated_context = f"{state.context}\n\nResearch Findings:\n{research_content}"

    return {"context": updated_context}
```

4. Responder Node (_responder_node)
This node generates the actual answer to the user's query using all accumulated information.

```
    Python


def _responder_node(state: AgentState) -> Dict[str, Any]:
    print(f"--- Iteration {state.iteration}: Responder Node ---")
    system_msg_content = """You are a helpful and knowledgeable AI assistant.
Based on the user's query, the provided context (which may include initial analysis, research findings, and previous attempts/validation), and the current analysis,
generate a comprehensive, accurate, and well-structured response.
If previous attempts were made and validated as needing improvement, try to address those points.
Aim for clarity and completeness."""
    messages =
    response = llm.invoke(messages)
    return {"response": response.content}
```

5. Validator Node (_validator_node)
This node assesses the generated response.

```
    Python


def _validator_node(state: AgentState) -> Dict[str, Any]:
    print(f"--- Iteration {state.iteration}: Validator Node ---")
    system_msg_content = """You are a meticulous response quality evaluator.
Review the original user query and the AI-generated response.
Determine if the response adequately, accurately, and comprehensively answers the query.
If the response is satisfactory and complete, state 'VALIDATION: COMPLETE'.
If the response needs improvement (e.g., lacks detail, is unclear, partially incorrect, or misses aspects of the query), state 'VALIDATION: NEEDS_IMPROVEMENT' and briefly explain why.
Your feedback will be used to refine the response in a subsequent iteration if needed."""
    messages =
    response = analyzer_llm.invoke(messages)
    validation_result = response.content
```

```
    print(f"Validator Output: {validation_result}")

    # Append validation result to context so it's available for next iteration
    updated_context_with_validation = f"{state.context}\n\nValidation (Iteration {state.iteration}):\n{validation_result}"

    return {"context": updated_context_with_validation}
```

5.5. Crafting Conditional Edges for Iterative Logic

Conditional edges control the flow based on the AgentState.

1. _decide_next_step (after Analyzer)
This function determines whether to go to the researcher or responder node based on the next_action field set by the _analyzer_node.

```
Python


def _decide_next_step(state: AgentState) -> str:
    print(f"--- Conditional Edge: Decide Next Step ---")
    print(f"Decision based on 'next_action': {state.next_action}")
    if state.next_action == "research":
        return "researcher"
    else: # "respond" or any other case
        return "responder"
```

2. _should_continue (after Validator)
This function determines if the agent should loop back for another iteration or end the process.

```
Python


def _should_continue(state: AgentState) -> str:
    print(f"--- Conditional Edge: Should Continue? ---")
    print(f"Current Iteration: {state.iteration}, Max Iterations: {state.max_iterations}")
    print(f"Context for validation check:...{state.context[-200:]}") # Print last part of context for check

    if state.iteration >= state.max_iterations:
```

```
        print("Decision: Max iterations reached. Ending.")
        return "end"

    # Check the validator's output (which was appended to context)
    if "VALIDATION: COMPLETE" in state.context.upper():
        print("Decision: Validation complete. Ending.")
        return "end"
    elif "VALIDATION: NEEDS_IMPROVEMENT" in state.context.upper():
        print("Decision: Validation indicates needs improvement. Continuing.")
        return "continue" # Loop back to analyzer
    else:
        # Default to end if no clear signal, or if validator output format changes.
        # This is a fallback; ideally, validator is consistent.
        print("Decision: No clear 'COMPLETE' or 'NEEDS_IMPROVEMENT' in validation. Ending as a fallback.")
        return "end"
```

5.6. Building and Compiling the StateGraph

Now, assemble the nodes and edges into a StateGraph and compile it.[10]

Python

```
# Create a new StateGraph with the AgentState
workflow = StateGraph(AgentState)

# Add nodes to the graph
workflow.add_node("router", _router_node)
workflow.add_node("analyzer", _analyzer_node)
workflow.add_node("researcher", _researcher_node)
workflow.add_node("responder", _responder_node)
workflow.add_node("validator", _validator_node)

# Set the entry point for the graph
workflow.set_entry_point("router")

# Add standard edges
workflow.add_edge("router", "analyzer")
```

```
# The researcher node, if called, will proceed to the responder
workflow.add_edge("researcher", "responder")
# The responder node always goes to the validator
workflow.add_edge("responder", "validator")

# Add conditional edges
# After analyzer, decide whether to research or respond
workflow.add_conditional_edges(
    "analyzer",
    _decide_next_step,
    {
        "researcher": "researcher",
        "responder": "responder"
    }
)

# After validator, decide whether to continue iterating or end
workflow.add_conditional_edges(
    "validator",
    _should_continue,
    {
        "continue": "analyzer", # Loop back to the analyzer for refinement
        "end": END              # END is a special node indicating termination
    }
)

# Compile the graph into a runnable application
app = workflow.compile()
# Optional: Use a checkpointer for persistence if needed
# checkpointer = SqliteSaver.from_conn_string(":memory:") # In-memory example
# app = workflow.compile(checkpointer=checkpointer)
```

Visualizing the Graph (Optional)
If working in an environment like a Jupyter Notebook, the graph structure can be visualized.12

```
    Python


# This code is typically run in a Jupyter Notebook cell
# from IPython.display import Image, display
# try:
```

```
#     # Generate a Mermaid diagram of the graph and display it
#     graph_image = app.get_graph().draw_mermaid_png()
#     display(Image(graph_image))
# except Exception as e:
#     print(f"Graph visualization failed. Ensure graphviz and/or mermaid capabilities are available: {e}")
```

This visualization helps in verifying the agent's flow and connections.

6. Running, Testing, and Observing Your Agent

With the agent designed and implemented, the next crucial phase is running it with sample inputs, testing its behavior, and observing its internal workings. LangGraph's streaming capabilities are particularly useful for this.

6.1. Preparing Sample Inputs for Your Agent

To thoroughly test the agent, it's advisable to use a diverse set of queries. These could include:

1. **Simple Factual Question**: e.g., "What is the capital of France?" (Should likely be a direct response, minimal iteration).
2. **Question Requiring Some "Research" or Elaboration**: e.g., "Explain the concept of photosynthesis in simple terms for a 5th grader." (Might benefit from the researcher node and potentially an iteration if the first explanation isn't clear enough).
3. **Creative Prompt**: e.g., "Write a short story about a robot who discovers music." (Tests generative capabilities and iterative refinement of the narrative).
4. **Problem-Solving Task**: e.g., "My plant's leaves are turning yellow. What could be the cause and how can I fix it?" (Might require research and careful validation of the advice).

For this guide, let's use a query that could benefit from the iterative process:
"Explain the core principles of LangGraph for building AI agents and why its cyclical nature is important."

6.2. Invoking the Compiled Graph

The compiled app can be invoked with an initial state. The query and max_iterations are key inputs.

Python

```
# Define the input for the agent
input_query = "Explain the core principles of LangGraph for building AI agents and why its cyclical nature is important."
initial_inputs = {
    "query": input_query,
    "max_iterations": 3 # Allow up to 3 full iterations (router is iteration 1 start)
}

# To get a single final result (blocking call):
# print("\n--- Invoking Agent (Blocking Call) ---")
# final_state = app.invoke(initial_inputs)
# print("\n--- Final Agent Response ---")
# print(final_state.get('response', "No response generated."))
# print(f"\nFinal context:\n{final_state.get('context')}")
# print(f"Total iterations performed (approx): {final_state.get('iteration')}")
```

While app.invoke() provides the final state, it doesn't show the intermediate steps, which are crucial for understanding an iterative agent.

6.3. Leveraging LangGraph's Streaming Modes to Monitor Agent Behavior

LangGraph's streaming capabilities allow for real-time observation of the agent's state as it executes each node and makes decisions.[2] This visibility is paramount for debugging, understanding the agent's "reasoning" process, and building trust in its behavior. Without observing intermediate steps, diagnosing issues in a complex, cyclical agent can be exceptionally challenging.

Key streaming modes include [17]:

- "values": Emits the full state dictionary after each step (node execution) in the graph. This is very useful for seeing the complete picture of the AgentState as it evolves.
- "updates": Emits only the changes (updates) returned by each node. This is more concise and shows what specific information each node contributed or modified.
- "messages": Streams LLM tokens if LLMs are used in a streaming-compatible way within nodes.
- "debug": Provides highly verbose output for deep debugging.

For observing the iterative nature of our agent, "values" or "updates" are most illustrative.

```python
print(f"\n--- Streaming Agent Execution (Mode: 'values') for query: '{input_query}' ---")
# config = {"configurable": {"thread_id": "test-thread-1"}} # Example for checkpointer
for chunk in app.stream(initial_inputs, stream_mode="values"): #, config=config):
    print("\n--- Next State Chunk (Values) ---")
    # 'chunk' here is the complete AgentState dictionary at that step
    print(f"Current Iteration in State: {chunk.get('iteration')}")
    print(f"Next Action Decided: {chunk.get('next_action', 'N/A')}")
    print(f"Current Analysis: {chunk.get('analysis', 'N/A')[:200]}...") # Print snippet
    print(f"Current Response: {chunk.get('response', 'N/A')[:200]}...") # Print snippet
    # For full detail, one might print the entire chunk: print(chunk)

print(f"\n--- Streaming Agent Execution (Mode: 'updates') for query: '{input_query}' ---")
for chunk in app.stream(initial_inputs, stream_mode="updates"): #, config=config):
    print("\n--- Next Update Chunk (Updates) ---")
    # 'chunk' is a dictionary like {'node_name': {'key_updated': 'new_value'}}
    # For example: {'analyzer': {'analysis': '...', 'next_action': 'research'}}
    for node_name, update_dict in chunk.items():
        print(f"Node '{node_name}' executed and returned updates:")
        for key, value in update_dict.items():
            print(f"  - {key}: {str(value)[:200]}...") # Print snippet of value
```

The choice between "values" and "updates" depends on the desired level of detail. "values" provides the full context at each step, making it easier to trace the complete state evolution. "updates" is more focused, highlighting exactly what each node changed, which can be useful for pinpointing where specific state modifications occur. Both offer valuable perspectives for understanding and debugging the agent.

6.4. Interpreting Snapshots of Intermediate States and Outputs

When observing the streamed output, focus on:

- **iteration count**: Watch it increment, especially when the _should_continue edge loops back to the analyzer.
- **context field**: Observe how it grows with router analysis, research findings, and validator feedback.
- **analysis and response fields**: See how these are populated and potentially refined across iterations.
- **next_action field**: Note the decisions made by the _analyzer_node.

- **Node execution sequence**: The print statements within each node and the structure of the "updates" stream will show which node is currently active.
- **Conditional logic**: Observe the output of _decide_next_step and _should_continue functions (via their print statements) to understand why the graph took a particular path.

Conceptual Snapshot of Streamed Output (using "values" mode):

After Router Node (Iteration 1 initiated):

```
--- Next State Chunk (Values) ---
Current Iteration in State: 1
Next Action Decided: N/A
Current Analysis: N/A...
Current Response: N/A...
{
    'query': 'Explain LangGraph principles...',
    'context': 'Initial Router Analysis: The query asks for core principles of LangGraph for AI agents and the importance of its cyclical nature. This is an analytical task requiring explanation of technical concepts.',
    'analysis': '',
    'response': '',
    'next_action': '',
    'iteration': 1,
    'max_iterations': 3
}
```

After Analyzer Node (Iteration 1):

```
--- Next State Chunk (Values) ---
Current Iteration in State: 1
Next Action Decided: research
```

Current Analysis: User Query: Explain LangGraph principles...
Current Context: Initial Router Analysis: The query asks for core principles of LangGraph for AI agents...
Analyzer Output: RESEARCH_REQUIRED. The query requires detailed explanation of LangGraph's state, nodes, edges, and the significance of cycles. Direct response might lack depth without focused information gathering.
Analyzer Decision: research...
Current Response: N/A...

{
  'query': 'Explain LangGraph principles...',
  'context': 'Initial Router Analysis:...',
  'analysis': 'RESEARCH_REQUIRED. The query requires detailed explanation...',
  'response': '',
  'next_action': 'research',
  'iteration': 1,
  'max_iterations': 3
}

After Validator Node (End of Iteration 1, assuming it went Router → Analyzer → Researcher → Responder → Validator):

--- Next State Chunk (Values) ---
Current Iteration in State: 1
Next Action Decided: research # (This was from the Analyzer step of this iteration)
Current Analysis: RESEARCH_REQUIRED.... # (This was from the Analyzer step of this iteration)
Current Response: LangGraph is a library for building stateful AI applications. Its core principles include State, Nodes, and Edges... Its cyclical nature is important because it allows agents to refine work... # (Example response)

{
  'query': 'Explain LangGraph principles...',
  'context': 'Initial Router Analysis:...\n\nResearch Findings:\nLangGraph uses State for shared data, Nodes for actions, Edges for flow. Cycles enable iterative refinement...\n\nValidation (Iteration 1):\nVALIDATION: NEEDS_IMPROVEMENT. The explanation of cyclical nature could be more detailed with examples.',

```
    'analysis': 'RESEARCH_REQUIRED....',
    'response': 'LangGraph is a library...',
    'next_action': 'research',
    'iteration': 1,
    'max_iterations': 3
}
```

If the validator outputted "NEEDS_IMPROVEMENT" and iteration < max_iterations, the next step seen in the stream would be the Analyzer Node again, but this time for iteration 2, and its context input would include the "NEEDS_IMPROVEMENT" feedback. This iterative refinement is the core behavior this agent is designed to demonstrate.

7. Conclusion: Embracing Iterative AI with LangGraph and Gemini

This guide has demonstrated the construction of an iterative AI workflow agent using LangGraph and Google's Gemini 1.5 Flash model. The agent, designed as an intelligent query handler, employs a multi-step process involving routing, analysis, research, response generation, and validation. The cyclical architecture, enabled by LangGraph's state management and conditional edges, allows the agent to refine its outputs iteratively, aiming for improved accuracy and completeness.[8]

The power of such an iterative design lies in its ability to tackle complex tasks that benefit from deliberation and self-correction. By breaking down a problem into manageable steps and allowing for feedback loops, the agent can achieve a higher quality of output than a simple, one-shot generation. This approach mirrors human problem-solving strategies, where initial drafts are reviewed and improved.

LangGraph and Gemini together provide a flexible and potent toolkit for developing such sophisticated AI systems. LangGraph offers the structural framework for orchestrating complex, stateful, and cyclical workflows [2], while Gemini provides the advanced language understanding and generation capabilities.[4] The patterns learned in building this query-handling agent—defining a state, creating modular nodes for specific tasks, and using conditional logic to control iteration—form a foundational template. This template can be adapted and extended to a wide variety of other iterative AI tasks, such as iterative document summarization, code generation with self-correction, or multi-step planning and execution agents.

The development of AI agents is increasingly moving towards systems that offer greater control over their behavior and more transparency into their decision-making processes.[2] Frameworks like LangGraph, with their emphasis on explicit state,

graph-based control flow, and comprehensive streaming capabilities for observability [9], are at the forefront of this trend. As AI models become more powerful, the ability to reliably direct, manage, and understand their operations becomes paramount, especially for complex or high-stakes applications.

Potential Extensions and Further Learning:

The agent built in this guide serves as a solid starting point. Several extensions could enhance its capabilities:

- **True Tool Integration**: Replace the simulated _researcher_node with actual tools, such as a web search tool (e.g., Tavily, as seen in other LangChain contexts [1]), a calculator, or a code interpreter. LangChain provides a rich ecosystem of tools that can be integrated.
- **Enhanced Memory**: For conversational agents that interact over multiple turns, incorporating more sophisticated memory mechanisms (e.g., ConversationBufferMemory from LangChain) would be essential.[2]
- **Human-in-the-Loop (HITL)**: Introduce steps where a human can review, approve, or correct the agent's output or decisions, guiding its iteration.[2] LangGraph is well-suited for HITL workflows.
- **Advanced State Management**: Explore LangGraph's options for custom reducer functions to manage how state updates are applied, allowing for more nuanced state changes than simple overwrites or appends.[9]
- **Different Gemini Models**: Experiment with other Gemini models, including Gemini Pro for potentially more powerful reasoning or even multimodal Gemini variants if the task involves processing images, audio, or video.[4]
- **Robust Error Handling**: Implement more comprehensive error handling within each node to manage API failures, unexpected outputs, or other runtime issues gracefully.
- **Prebuilt LangGraph Components**: Explore LangGraph's prebuilt agent patterns or supervisor abstractions for orchestrating more complex multi-agent systems.[19]

By continuing to explore these areas, developers can build increasingly sophisticated, reliable, and controllable AI agents capable of tackling a new generation of intelligent automation tasks.

8. Appendix: Complete Code and Notebook Snippets

This appendix provides the consolidated Python code for the iterative AI agent and a representation of how it might be structured in a Jupyter Notebook environment for

ease of use and reproducibility.

8.1. Consolidated Python Script

Python

```
import os
from dotenv import load_dotenv
from typing import TypedDict, Any, Dict, List
from dataclasses import dataclass, field

from langchain_Genai import ChatGoogleGenerativeAI
from langchain_core.messages import HumanMessage, SystemMessage
from langgraph.graph import StateGraph, START, END
# from langgraph.checkpoint.sqlite import SqliteSaver # Optional for persistence

# --- 1. Setup: Environment and API Key ---
load_dotenv()
# Ensure GOOGLE_API_KEY is set in your.env file or environment

# --- 2. LLM Initialization ---
llm_model_name = "gemini-1.5-flash-latest" # Or "gemini-pro"
llm = ChatGoogleGenerativeAI(model=llm_model_name, temperature=0.7,
convert_system_message_to_human=True)
analyzer_llm = ChatGoogleGenerativeAI(model=llm_model_name, temperature=0.1,
convert_system_message_to_human=True)

# --- 3. AgentState Definition ---
@dataclass
class AgentState:
    query: str
    context: str = ""
    analysis: str = ""
    response: str = ""
    next_action: str = ""
    iteration: int = 0
    max_iterations: int = 3

# --- 4. Node Definitions ---
```

```
def _router_node(state: AgentState) -> Dict[str, Any]:
    print(f"\n--- Iteration {state.iteration + 1}: Router Node ---")
    system_msg_content = """You are a query router. Analyze the user's query and provide initial context.
Determine if this is a factual question, a creative request, a problem-solving task, or an analytical task.
Briefly summarize the query's core intent."""
    messages =
    response_content = llm.invoke(messages).content
    print(f"Router Output: {response_content[:200]}...")
    return {
        "context": f"Initial Router Analysis: {response_content}",
        "iteration": state.iteration + 1
    }

def _analyzer_node(state: AgentState) -> Dict[str, Any]:
    print(f"\n--- Iteration {state.iteration}: Analyzer Node ---")
    system_msg_content = """You are an expert analyst. Review the user's query and the current context.
Determine if additional research is absolutely necessary to provide a comprehensive and accurate answer, or if a direct response can be formulated.
If research is needed, state 'RESEARCH_REQUIRED'. Otherwise, state 'DIRECT_RESPONSE_POSSIBLE'.
Provide a brief justification for your decision.
Current context may include previous responses and validation feedback if this is not the first iteration."""

    content_parts = [
        f"User Query: {state.query}",
        f"Current Context: {state.context}",
    ]
    if state.analysis:
        content_parts.append(f"Previous Analysis (if any): {state.analysis}")
    if state.response:
        content_parts.append(f"Previous Response (if any): {state.response}")

    messages =

    response_content = analyzer_llm.invoke(messages).content
    analysis_text = response_content

    next_action = "respond"
    if "RESEARCH_REQUIRED" in analysis_text.upper():
```

```
        next_action = "research"

    print(f"Analyzer Output: {analysis_text[:200]}...")
    print(f"Analyzer Decision for Next Action: {next_action}")

    return {
        "analysis": analysis_text,
        "next_action": next_action
    }

def _researcher_node(state: AgentState) -> Dict[str, Any]:
    print(f"\n--- Iteration {state.iteration}: Researcher Node ---")
    system_msg_content = """You are a research assistant. Based on the user query and the
analysis provided,
gather relevant information and detailed insights. Focus on providing factual content that will help
answer the query comprehensively.
The analysis might specify areas needing more information."""
    messages =
    response_content = llm.invoke(messages).content
    research_content = response_content
    print(f"Researcher Output: {research_content[:200]}...")
    updated_context = f"{state.context}\n\nResearch Findings (Iteration
{state.iteration}):\n{research_content}"

    return {"context": updated_context}

def _responder_node(state: AgentState) -> Dict[str, Any]:
    print(f"\n--- Iteration {state.iteration}: Responder Node ---")
    system_msg_content = """You are a helpful and knowledgeable AI assistant.
Based on the user's query, the provided context (which may include initial analysis, research findings,
and previous attempts/validation), and the current analysis,
generate a comprehensive, accurate, and well-structured response.
If previous attempts were made and validated as needing improvement, try to address those points.
Aim for clarity and completeness."""
    messages =
    response_content = llm.invoke(messages).content
    print(f"Responder Output: {response_content[:200]}...")
    return {"response": response_content}

def _validator_node(state: AgentState) -> Dict[str, Any]:
    print(f"\n--- Iteration {state.iteration}: Validator Node ---")
```

```
    system_msg_content = """You are a meticulous response quality evaluator.
Review the original user query and the AI-generated response.
Determine if the response adequately, accurately, and comprehensively answers the query.
If the response is satisfactory and complete, state 'VALIDATION: COMPLETE'.
If the response needs improvement (e.g., lacks detail, is unclear, partially incorrect, or misses aspects of the query), state 'VALIDATION: NEEDS_IMPROVEMENT' and briefly explain why.
Your feedback will be used to refine the response in a subsequent iteration if needed."""
    messages =
    response_content = analyzer_llm.invoke(messages).content
    validation_result = response_content
    print(f"Validator Output: {validation_result[:200]}...")

    updated_context_with_validation = f"{state.context}\n\nValidation (Iteration {state.iteration}):\n{validation_result}"

    return {"context": updated_context_with_validation}

# --- 5. Conditional Edge Functions ---
def _decide_next_step(state: AgentState) -> str:
    print(f"\n--- Conditional Edge Logic: Deciding Next Step based on '{state.next_action}' ---")
    if state.next_action == "research":
        return "researcher"
    return "responder"

def _should_continue(state: AgentState) -> str:
    print(f"\n--- Conditional Edge Logic: Should Continue? Iteration {state.iteration}/{state.max_iterations} ---")
    # Check validator output (appended to context)
    validation_text_upper = state.context.upper() # Check the whole context for robustness

    if state.iteration >= state.max_iterations:
        print("Decision: Max iterations reached. Ending.")
        return "end"
    if "VALIDATION: COMPLETE" in validation_text_upper:
        print("Decision: Validation complete. Ending.")
        return "end"
    if "VALIDATION: NEEDS_IMPROVEMENT" in validation_text_upper:
        print("Decision: Validation indicates needs improvement. Continuing.")
        return "continue"

    print("Decision: No clear 'COMPLETE' or 'NEEDS_IMPROVEMENT' in validation context. Ending as
```

```
fallback.")
    return "end"

# --- 6. Graph Construction and Compilation ---
workflow = StateGraph(AgentState)

workflow.add_node("router", _router_node)
workflow.add_node("analyzer", _analyzer_node)
workflow.add_node("researcher", _researcher_node)
workflow.add_node("responder", _responder_node)
workflow.add_node("validator", _validator_node)

workflow.set_entry_point("router")

workflow.add_edge("router", "analyzer")
workflow.add_edge("researcher", "responder")
workflow.add_edge("responder", "validator")

workflow.add_conditional_edges("analyzer", _decide_next_step,
                               {"researcher": "researcher", "responder": "responder"})
workflow.add_conditional_edges("validator", _should_continue,
                               {"continue": "analyzer", "end": END})

app = workflow.compile()
# Optional: checkpointer = SqliteSaver.from_conn_string(":memory:")
# app = workflow.compile(checkpointer=checkpointer)

# --- 7. Running the Agent ---
if __name__ == "__main__":
    queries_to_test = [
        "Explain the core principles of LangGraph for building AI agents and why its cyclical nature is
important.",
        "What are the key benefits of using Gemini 1.5 Flash for generative AI tasks?",
        "Write a short poem about an AI learning to dream."
    ]

    for query_text in queries_to_test:
        print(f"\n\n{'='*20} Processing Query: {query_text} {'='*20}")
        initial_inputs = {"query": query_text, "max_iterations": 3}
```

```
    # Stream updates to see the flow
    print(f"\n--- Streaming Agent Execution (Mode: 'updates') ---")
    final_response_value = None
    final_context_value = None
    final_iteration_count = 0

    # config = {"configurable": {"thread_id": f"thread-{query_text[:20].replace(' ','_')}"}} # For checkpointer
    for chunk_num, chunk in enumerate(app.stream(initial_inputs, stream_mode="values")): #, config=config)):
        print(f"\n--- Stream Chunk {chunk_num + 1} (Values) ---")
        print(f"Current Iteration in State: {chunk.get('iteration')}")
        print(f"Next Action Decided: {chunk.get('next_action', 'N/A')}")
        print(f"Context (last 200 chars):...{chunk.get('context', '')[-200:]}")
        print(f"Analysis (first 200 chars): {chunk.get('analysis', '')[:200]}...")
        print(f"Response (first 200 chars): {chunk.get('response', '')[:200]}...")

        # Capture the latest response and context for final output
        if chunk.get('response'):
            final_response_value = chunk.get('response')
        if chunk.get('context'):
            final_context_value = chunk.get('context')
        if chunk.get('iteration'):
            final_iteration_count = chunk.get('iteration')


    print(f"\n\n--- Final Result for Query: '{query_text}' ---")
    if final_response_value:
        print("\nFinal Response:")
        print(final_response_value)
    else:
        print("\nNo final response was generated (or captured).")

    # print("\nFull Final Context (if needed for debugging):")
    # print(final_context_value)
    print(f"Total iterations performed (approx based on state): {final_iteration_count}")
    print(f"{'='*50}\n")
```

8.2. Key Notebook Cell Representations

Below is how the agent development might be structured in a Jupyter Notebook.

Cell 1: Installations

Python

```
#!pip install langgraph langchain-google-genai python-dotenv langchain-core ipython
```

Cell 2: API Key Setup & Imports

Python

```
import os
from dotenv import load_dotenv
from typing import TypedDict, Any, Dict, List # If using TypedDict
from dataclasses import dataclass, field

from langchain_Genai import ChatGoogleGenerativeAI
from langchain_core.messages import HumanMessage, SystemMessage
from langgraph.graph import StateGraph, START, END
# from langgraph.checkpoint.sqlite import SqliteSaver # Optional

# Load environment variables
load_dotenv()
# GOOGLE_API_KEY = os.getenv("GOOGLE_API_KEY") # Handled by ChatGoogleGenerativeAI
```

Cell 3: LLM Initialization

Python

```
llm_model_name = "gemini-1.5-flash-latest"
```

```
llm = ChatGoogleGenerativeAI(model=llm_model_name, temperature=0.7,
convert_system_message_to_human=True)
analyzer_llm = ChatGoogleGenerativeAI(model=llm_model_name, temperature=0.1,
convert_system_message_to_human=True)
print("LLMs initialized.")
```

Cell 4: AgentState Definition

Python

```
@dataclass
class AgentState:
    query: str
    context: str = ""
    analysis: str = ""
    response: str = ""
    next_action: str = ""
    iteration: int = 0
    max_iterations: int = 3 # Default max iterations

print("AgentState defined.")
```

Cell 5: Router Node Definition

Python

```
def _router_node(state: AgentState) -> Dict[str, Any]:
    #... (code from consolidated script)...
    return {
        "context": f"Initial Router Analysis: {response_content}",
        "iteration": state.iteration + 1
    }
print("_router_node defined.")
```

Cell 6: Analyzer Node Definition

Python

```
def _analyzer_node(state: AgentState) -> Dict[str, Any]:
    #... (code from consolidated script)...
    return {
        "analysis": analysis_text,
        "next_action": next_action
    }
print("_analyzer_node defined.")
```

Cell 7: Researcher Node Definition

Python

```
def _researcher_node(state: AgentState) -> Dict[str, Any]:
    #... (code from consolidated script)...
    return {"context": updated_context}
print("_researcher_node defined.")
```

Cell 8: Responder Node Definition

Python

```
def _responder_node(state: AgentState) -> Dict[str, Any]:
    #... (code from consolidated script)...
    return {"response": response_content}
print("_responder_node defined.")
```

Cell 9: Validator Node Definition

Python

```
def _validator_node(state: AgentState) -> Dict[str, Any]:
    #... (code from consolidated script)...
    return {"context": updated_context_with_validation}
print("_validator_node defined.")
```

Cell 10: Conditional Edge Function _decide_next_step

Python

```
def _decide_next_step(state: AgentState) -> str:
    #... (code from consolidated script)...
    if state.next_action == "research":
        return "researcher"
    return "responder"
print("_decide_next_step defined.")
```

Cell 11: Conditional Edge Function _should_continue

Python

```
def _should_continue(state: AgentState) -> str:
    #... (code from consolidated script)...
    # (logic to return "continue" or "end")
    return "end" # Placeholder, actual logic is in consolidated script
print("_should_continue defined.")
```

Cell 12: Graph Construction and Compilation

Python

```
workflow = StateGraph(AgentState)

# Add nodes
workflow.add_node("router", _router_node)
#... (add all other nodes)...
workflow.add_node("validator", _validator_node)


# Set entry point
workflow.set_entry_point("router")

# Add edges
workflow.add_edge("router", "analyzer")
#... (add other normal edges)...
workflow.add_edge("responder", "validator")


# Add conditional edges
workflow.add_conditional_edges("analyzer", _decide_next_step,
                               {"researcher": "researcher", "responder": "responder"})
workflow.add_conditional_edges("validator", _should_continue,
                               {"continue": "analyzer", "end": END})

app = workflow.compile()
# Optional: checkpointer = SqliteSaver.from_conn_string(":memory:")
# app = workflow.compile(checkpointer=checkpointer)
print("Graph compiled.")
```

Cell 13: (Optional) Graph Visualization

Python

```
from IPython.display import Image, display
try:
    display(Image(app.get_graph().draw_mermaid_png()))
```

```
except Exception as e:
    print(f"Graph visualization failed (mermaid may not be installed or renderable): {e}")
```

Cell 14: Running the Agent with Streaming

```
Python


input_query = "Explain the core principles of LangGraph for building AI agents and why its cyclical nature is important."
initial_inputs = {"query": input_query, "max_iterations": 3}

print(f"\n--- Streaming Agent Execution (Mode: 'values') for query: '{input_query}' ---")
# config = {"configurable": {"thread_id": "notebook-thread-1"}} # For checkpointer
for chunk_num, chunk in enumerate(app.stream(initial_inputs, stream_mode="values")): #, config=config)):
    print(f"\n--- Stream Chunk {chunk_num + 1} (Values) ---")
    print(f"Current Iteration in State: {chunk.get('iteration')}")
    print(f"Next Action Decided: {chunk.get('next_action', 'N/A')}")
    # print(f"Full Chunk: {chunk}") # For very detailed view
    print(f"Context (last 100 chars):...{chunk.get('context', '')[-100:]}")
    print(f"Response (first 100 chars): {chunk.get('response', '')[:100]}...")

# To get the final response if needed after streaming
# final_state_after_stream = app.invoke(initial_inputs) # config=config)
# print("\nFinal Response after stream completion:")
# print(final_state_after_stream.get('response'))
```

This structured approach, providing both a consolidated script and notebook cell representations, aims to maximize usability and ensure that users can readily implement and experiment with the iterative AI agent. The ability to reproduce and modify the provided code is fundamental to effective technical learning.

Works cited

1. LangGraph integration | Platform | Apify Documentation, accessed on June 7, 2025, https://docs.apify.com/platform/integrations/langgraph
2. Learn LangGraph basics - Overview, accessed on June 7, 2025, https://langchain-ai.github.io/langgraph/concepts/why-langgraph/
3. LangGraph Tutorial for Beginners - Analytics Vidhya, accessed on June 7, 2025,

https://www.analyticsvidhya.com/blog/2025/05/langgraph-tutorial-for-beginners/
4. Gemini API I/O updates - Google Developers Blog, accessed on June 7, 2025, https://developers.googleblog.com/en/gemini-api-io-updates/
5. Build multimodal agents using Gemini, Langchain, and LangGraph | Google Cloud Blog, accessed on June 7, 2025, https://cloud.google.com/blog/products/ai-machine-learning/build-multimodal-agents-using-gemini-langchain-and-langgraph
6. Get started developing with Gemini API - YouTube, accessed on June 7, 2025, https://www.youtube.com/shorts/T1BTyo1A4Ww
7. Google Gemini LangChain Cheatsheet - Philschmid, accessed on June 7, 2025, https://www.philschmid.de/gemini-langchain-cheatsheet
8. A Step-by-Step Coding Guide to Building an Iterative AI Workflow ..., accessed on June 7, 2025, https://www.marktechpost.com/2025/06/05/a-step-by-step-coding-guide-to-building-an-iterative-ai-workflow-agent-using-langgraph-and-gemini/
9. Overview - GitHub Pages, accessed on June 7, 2025, https://langchain-ai.github.io/langgraph/concepts/low_level/
10. LangGraph - LangChain Blog, accessed on June 7, 2025, https://blog.langchain.dev/langgraph/
11. LangGraph Glossary - GitHub Pages, accessed on June 7, 2025, https://langchain-ai.github.io/langgraphjs/concepts/low_level/
12. LangGraph: Build Stateful AI Agents in Python - Real Python, accessed on June 7, 2025, https://realpython.com/langgraph-python/
13. How to get your Gemini API key (5 steps) - Merge.dev, accessed on June 7, 2025, https://www.merge.dev/blog/gemini-api-key
14. How To Get Your FREE Google Gemini API Key (2025) - YouTube, accessed on June 7, 2025, https://www.youtube.com/watch?v=6BRyynZkvf0
15. How to create a free Gemini AI API Key, accessed on June 7, 2025, https://www.geminiforwork.gwaddons.com/setup-api-keys/create-geminiai-api-key
16. Quickstart: Generate text using the Vertex AI Gemini API - Google Cloud, accessed on June 7, 2025, https://cloud.google.com/vertex-ai/generative-ai/docs/start/quickstarts/quickstart-multimodal
17. langgraph/docs/docs/how-tos/streaming.ipynb at main - GitHub, accessed on June 7, 2025, https://github.com/langchain-ai/langgraph/blob/main/docs/docs/how-tos/streaming.ipynb
18. Stream outputs - GitHub Pages, accessed on June 7, 2025, https://langchain-ai.github.io/langgraph/how-tos/streaming/
19. Reference - GitHub Pages, accessed on June 7, 2025, https://langchain-ai.github.io/langgraph/reference/