This document provides a detailed explanation of the gen_ai_service Python function, which defines an AI-powered Eco Lifestyle Agent using langchain_ibm and ibm_watsonx_ai.

Overview

The gen_ai_service function acts as an entry point for an AI service designed to help users live more sustainably. It configures a large language model (LLM) and integrates various tools to create an intelligent agent capable of providing eco-friendly advice, product suggestions, and local environmental information. The service supports both single-response and streaming interactions.

Key Components and Functions

The service is structured around several helper functions:

1. **gen_ai_service(context, params, **custom)**:
   - This is the main function that initializes the AI service.
   - It sets up the ibm_watsonx_ai.APIClient using credentials obtained from the context.
   - It defines the model to be used ("meta-llama/llama-3-3-70b-instruct") and the service_url.
   - It returns two functions: generate (for non-streaming responses) and generate_stream (for streaming responses).
2. **create_chat_model(watsonx_client)**:
   - Configures and returns a ChatWatsonx instance.
   - Sets various model parameters such as frequency_penalty, max_tokens, presence_penalty, temperature, and top_p to control the LLM's generation behavior.
3. **create_utility_agent_tool(tool_name, params, api_client, **kwargs)**:
   - A utility function to create StructuredTool instances from pre-defined tools available via the ibm_watsonx_ai.foundation_models.utils.Toolkit.
   - It fetches the tool's description and input schema. If no specific input schema is defined for the utility tool, it defaults to a simple {"input": "string"} schema.
   - The run_tool inner function handles the execution of the utility agent tool with the provided input.
4. **create_custom_tool(tool_name, tool_description, tool_code, tool_schema, tool_params)**:
   - This function allows for the creation of custom tools by dynamically executing provided Python code.
   - It parses the tool_code to extract the function name and then executes the

  - compiled code within a namespace that can include tool_params.
  - **Note:** While defined, this function is not explicitly used in the create_tools function in the provided code, indicating it's a potential extension point.
5. **create_tools(inner_client, context)**:
   - This function defines and returns a list of tools that the AI agent can use.
   - In the provided code, it includes:
     - GoogleSearch: For general web searches.
     - DuckDuckGo: Another search engine tool.
     - WebCrawler: For crawling web content.
6. **create_agent(model, tools, messages)**:
   - Initializes a LangGraph agent using create_react_agent.
   - It uses MemorySaver for checkpointing, allowing the agent to maintain conversation history.
   - The instructions string is crucial here, as it defines the agent's persona, responsibilities, tone, and how it should operate.
7. **convert_messages(messages)**:
   - Converts the incoming message format (e.g., {"role": "user", "content": "..."}) into langchain_core.messages.HumanMessage or AIMessage objects, which are compatible with the LangGraph agent.
8. **generate(context)**:
   - Handles non-streaming API requests.
   - It extracts messages from the payload, initializes the model and tools, creates the agent, and then invokes the agent with the converted messages.
   - The final response content from the agent is then formatted into a standard JSON response.
9. **generate_stream(context)**:
   - Handles streaming API requests, providing a more interactive experience.
   - It streams chunks of the agent's response, which can include:
     - messages: Actual content generated by the AI assistant.
     - updates: Information about tool calls (when the agent decides to use a tool) and tool responses (the result of a tool execution).
   - It formats these chunks into a streamable JSON format, including delta messages, finish_reason, and usage statistics.

Agent Persona: EcoGuide

The instructions within the create_agent function define the AI's persona as "EcoGuide":

- **Role:** An AI-powered Eco Lifestyle Agent designed to help users live more sustainably.

- **Responsibilities:**
  - Recommend simple daily habits to reduce environmental impact.
  - Provide eco-friendly product suggestions, recycling tips, and green alternatives.
  - Share local recycling guidelines, government schemes, and city-specific resources.
  - Offer realistic, budget-friendly advice.
- **Tone & Style:** Friendly, encouraging, non-judgmental, supportive, informative, and uses clear, actionable language.
- **How it Works:** Uses the IBM Granite model (via ChatWatsonx) and integrates Retrieval-Augmented Generation (RAG) to fetch real-time, up-to-date, and localized environmental content from trusted sources.

Dependencies

The Python code relies on the following key libraries:

- langchain_ibm: For integrating with IBM Watsonx AI models.
- ibm_watsonx_ai: The official Python SDK for IBM Watsonx AI, providing access to foundation models and utility agents.
- langchain_core: Core components for LangChain.
- langgraph: For building stateful, multi-step agent applications.
- json, requests, ast: Standard Python libraries for JSON handling, HTTP requests, and abstract syntax tree manipulation (for custom tools).

Usage Flow

A typical interaction with this service would involve:

1. A user sends a message (e.g., "How can I reduce plastic use at home?").
2. The generate or generate_stream function receives the message.
3. The create_agent sets up the EcoGuide persona and available tools.
4. The agent processes the message. It might:
   - Directly generate a response based on its knowledge.
   - Decide to call a tool (e.g., GoogleSearch for "government subsidies for solar panels in my area").
   - Process the tool's output and then generate a final, informed response.
5. The service returns the AI's response to the user, either as a complete message or a stream of updates.

This service provides a robust framework for building intelligent, tool-augmented AI agents with a specific persona and capabilities.

```
params = {
    "space_id": "a2385290-d257-40ef-b617-a1db596c9316",
}


def gen_ai_service(context, params = params, **custom):
    # import dependencies
    from langchain_ibm import ChatWatsonx
    from ibm_watsonx_ai import APIClient
    from ibm_watsonx_ai.foundation_models.utils import Tool, Toolkit
    from langchain_core.messages import AIMessage, HumanMessage
    from langgraph.checkpoint.memory import MemorySaver
    from langgraph.prebuilt import create_react_agent
    import json
    import requests

    model = "meta-llama/llama-3-3-70b-instruct"

    service_url = "https://us-south.ml.cloud.ibm.com"
    # Get credentials token
    credentials = {
        "url": service_url,
        "token": context.generate_token()
    }

    # Setup client
    client = APIClient(credentials)
    space_id = params.get("space_id")
    client.set.default_space(space_id)


    def create_chat_model(watsonx_client):
        parameters = {
            "frequency_penalty": 0.77,
            "max_tokens": 2000,
            "presence_penalty": -1.81,
            "temperature": 0.03,
            "top_p": 1
        }

        chat_model = ChatWatsonx(
            model_id=model,
            url=service_url,
            space_id=space_id,
            params=parameters,
            watsonx_client=watsonx_client,
        )
        return chat_model
```

```
def create_utility_agent_tool(tool_name, params, api_client, **kwargs):
    from langchain_core.tools import StructuredTool
    utility_agent_tool = Toolkit(
        api_client=api_client
    ).get_tool(tool_name)

    tool_description = utility_agent_tool.get("description")

    if (kwargs.get("tool_description")):
        tool_description = kwargs.get("tool_description")
    elif (utility_agent_tool.get("agent_description")):
        tool_description = utility_agent_tool.get("agent_description")

    tool_schema = utility_agent_tool.get("input_schema")
    if (tool_schema == None):
        tool_schema = {
            "type": "object",
            "additionalProperties": False,
            "$schema": "http://json-schema.org/draft-07/schema#",
            "properties": {
                "input": {
                    "description": "input for the tool",
                    "type": "string"
                }
            }
        }

    def run_tool(**tool_input):
        query = tool_input
        if (utility_agent_tool.get("input_schema") == None):
            query = tool_input.get("input")

        results = utility_agent_tool.run(
            input=query,
            config=params
        )

        return results.get("output")

    return StructuredTool(
        name=tool_name,
        description = tool_description,
        func=run_tool,
        args_schema=tool_schema
    )
```

```python
    def create_custom_tool(tool_name, tool_description, tool_code, tool_schema, tool_params):
        from langchain_core.tools import StructuredTool
        import ast

        def call_tool(**kwargs):
            tree = ast.parse(tool_code, mode="exec")
            custom_tool_functions = [ x for x in tree.body if isinstance(x, ast.FunctionDef) ]
            function_name = custom_tool_functions[0].name
            compiled_code = compile(tree, 'custom_tool', 'exec')
            namespace = tool_params if tool_params else {}
            exec(compiled_code, namespace)
            return namespace[function_name](**kwargs)

        tool = StructuredTool(
            name=tool_name,
            description = tool_description,
            func=call_tool,
            args_schema=tool_schema
        )
        return tool

    def create_custom_tools():
        custom_tools = []


    def create_tools(inner_client, context):
        tools = []

        config = None
        tools.append(create_utility_agent_tool("GoogleSearch", config, inner_client))
        config = {
        }
        tools.append(create_utility_agent_tool("DuckDuckGo", config, inner_client))
        config = {
        }
        tools.append(create_utility_agent_tool("WebCrawler", config, inner_client))
        return tools

    def create_agent(model, tools, messages):
        memory = MemorySaver()
        instructions = """# Notes
- Use markdown syntax for formatting code snippets, links, JSON, tables, images, files.
- Any HTML tags must be wrapped in block quotes, for example <html>.
- When returning code blocks, specify language.
- Sometimes, things don't go as planned. Tools may not provide useful information on the first few tries. You should always try a few different approaches before declaring the problem unsolvable.
- When the tool doesn't give you what you were asking for, you must either use another tool or a different tool input.
```

- When using search engines, you try different formulations of the query, possibly even in a different language.
- You cannot do complex calculations, computations, or data manipulations without using tools.
- If you need to call a tool to compute something, always call it instead of saying you will call it.

If a tool returns an IMAGE in the result, you must include it in your answer as Markdown.

Example:

Tool result:
IMAGE({commonApiUrl}/wx/v1-beta/utility_agent_tools/cache/images/plt-04e3c91ae04b47f8934a4e6b7d1fdc2c.png)
Markdown to return to user: ![Generated image]({commonApiUrl}/wx/v1-beta/utility_agent_tools/cache/images/plt-04e3c91ae04b47f8934a4e6b7d1fdc2c.png)

You are EcoGuide, an AI-powered Eco Lifestyle Agent designed to help users live more sustainably. Your role is to provide practical, personalized, and trustworthy guidance on eco-friendly living by retrieving accurate information from verified environmental sources using Retrieval-Augmented Generation (RAG).

Your Responsibilities:

Recommend simple daily habits that reduce environmental impact.

Provide eco-friendly product suggestions, recycling tips, and green alternatives.

Share local recycling guidelines, government schemes, and city-specific resources.

Offer realistic, budget-friendly advice to make sustainable choices accessible to everyone.

Tone & Style:

Friendly, encouraging, and non-judgmental.

Supportive and informative, never preachy.

Use clear and actionable language, avoiding jargon.

How You Work:

Use IBM Granite model on IBM Cloud Lite for natural language understanding and generation.

Integrate RAG to retrieve real-time, up-to-date, and localized environmental content from trusted databases, NGOs, and government sources.

```
Analyze user questions and deliver personalized recommendations in seconds.


Sample Use Cases:

\"How can I reduce plastic use at home?\"

\"Are there government subsidies for installing solar panels in my area?\"

\"What’s the best eco-friendly detergent available online?\"

\"Where can I recycle electronics near me?\"


Be a sustainability companion—always ready to help users make smarter, greener choices for a
healthier planet."""
        for message in messages:
            if message["role"] == "system":
                instructions += message["content"]
        graph = create_react_agent(model, tools=tools, checkpointer=memory,
state_modifier=instructions)
        return graph

    def convert_messages(messages):
        converted_messages = []
        for message in messages:
            if (message["role"] == "user"):
                converted_messages.append(HumanMessage(content=message["content"]))
            elif (message["role"] == "assistant"):
                converted_messages.append(AIMessage(content=message["content"]))
        return converted_messages

    def generate(context):
        payload = context.get_json()
        messages = payload.get("messages")
        inner_credentials = {
            "url": service_url,
            "token": context.get_token()
        }

        inner_client = APIClient(inner_credentials)
        model = create_chat_model(inner_client)
        tools = create_tools(inner_client, context)
        agent = create_agent(model, tools, messages)

        generated_response = agent.invoke(
            { "messages": convert_messages(messages) },
            { "configurable": { "thread_id": "42" } }
        )

        last_message = generated_response["messages"][-1]
        generated_response = last_message.content

        execute_response = {
            "headers": {
                "Content-Type": "application/json"
            },
            "body": {
                "choices": [{
                    "index": 0,
                    "message": {
                        "role": "assistant",
                        "content": generated_response
                    }
                }]
            }
        }

        return execute_response

    def generate_stream(context):
        print("Generate stream", flush=True)
        payload = context.get_json()
        headers = context.get_headers()
        is_assistant = headers.get("X-Ai-Interface") == "assistant"
        messages = payload.get("messages")
        inner_credentials = {
            "url": service_url,
            "token": context.get_token()
        }
        inner_client = APIClient(inner_credentials)
        model = create_chat_model(inner_client)
        tools = create_tools(inner_client, context)
        agent = create_agent(model, tools, messages)

        response_stream = agent.stream(
            { "messages": messages },
            { "configurable": { "thread_id": "42" } },
            stream_mode=["updates", "messages"]
        )

        for chunk in response_stream:
            chunk_type = chunk[0]
            finish_reason = ""
            usage = None
            if (chunk_type == "messages"):
```

```
    message_object = chunk[1][0]
    if (message_object.type == "AIMessageChunk" and message_object.content != ""):
        message = {
            "role": "assistant",
            "content": message_object.content
        }
    else:
        continue
elif (chunk_type == "updates"):
    update = chunk[1]
    if ("agent" in update):
        agent = update["agent"]
        agent_result = agent["messages"][0]
        if (agent_result.additional_kwargs):
            kwargs = agent["messages"][0].additional_kwargs
            tool_call = kwargs["tool_calls"][0]
            if (is_assistant):
                message = {
                    "role": "assistant",
                    "step_details": {
                        "type": "tool_calls",
                        "tool_calls": [
                            {
                                "id": tool_call["id"],
                                "name": tool_call["function"]["name"],
                                "args": tool_call["function"]["arguments"]
                            }
                        ]
                    }
                }
            else:
                message = {
                    "role": "assistant",
                    "tool_calls": [
                        {
                            "id": tool_call["id"],
                            "type": "function",
                            "function": {
                                "name": tool_call["function"]["name"],
                                "arguments": tool_call["function"]["arguments"]
                            }
                        }
                    ]
                }
        elif (agent_result.response_metadata):
            # Final update
            message = {
                "role": "assistant",
                "content": agent_result.content
            }
            finish_reason = agent_result.response_metadata["finish_reason"]
            if (finish_reason):
                message["content"] = ""

            usage = {
                "completion_tokens": agent_result.usage_metadata["output_tokens"],
                "prompt_tokens": agent_result.usage_metadata["input_tokens"],
                "total_tokens": agent_result.usage_metadata["total_tokens"]
            }
        elif ("tools" in update):
            tools = update["tools"]
            tool_result = tools["messages"][0]
            if (is_assistant):
                message = {
                    "role": "assistant",
                    "step_details": {
                        "type": "tool_response",
                        "id": tool_result.id,
                        "tool_call_id": tool_result.tool_call_id,
                        "name": tool_result.name,
                        "content": tool_result.content
                    }
                }
            else:
                message = {
                    "role": "tool",
                    "id": tool_result.id,
                    "tool_call_id": tool_result.tool_call_id,
                    "name": tool_result.name,
                    "content": tool_result.content
                }
    else:
        continue

chunk_response = {
    "choices": [{
        "index": 0,
        "delta": message
    }]
}
if (finish_reason):
    chunk_response["choices"][0]["finish_reason"] = finish_reason
if (usage):
    chunk_response["usage"] = usage
yield chunk_response
```

```
return generate, generate_stream
```