Building Conversational AI with Amazon Bedrock, LangChain, and LangGraph: A Step-by-Step Guide

Objective

The objective of this project is to **build an intelligent, stateful conversational AI assistant** powered by **Claude (via Amazon Bedrock)**, using **LangChain**, **LangGraph**, and **AWS-native session storage**. This assistant goes beyond a basic chatbot by maintaining conversation history, dynamically managing execution using graph-based logic, and persisting sessions through checkpointing — enabling real-time, memory-aware interactions across long conversations or even across multiple sessions.

Through this blog, readers will learn how to:

- Integrate **Claude 3** models using **Amazon Bedrock**
- Build a **graph-based conversational pipeline** using **LangGraph**
- Persist and manage **session memory** with **AWS-native checkpointing**
- Stream responses interactively and **maintain multi-turn dialogue**
- Structure modular conversational workflows for real-world AI applications

The goal is to help developers and AI enthusiasts **create smarter, context-aware AI systems** that scale reliably and can handle complex, long-running conversations — just like a human assistant would.

What Are We Building?

In this project, we're creating a smart, context-aware AI assistant that goes far beyond a basic chatbot. By combining LangGraph, LangChain, and Claude 3 via Amazon Bedrock, our assistant is capable of the following:

- **Maintains conversation context** using session-aware memory

- **Supports long-running interactions** that users can resume later
- **Implements graph-based logic** for clean, modular, and maintainable workflows
- **Streams responses in real time** to mimic natural conversation
- **Runs on secure, serverless AWS infrastructure** using Amazon Bedrock

Real-World Applications

This architecture is ideal for building:

- E-commerce chatbots that remember user preferences
- AI agents that manage and recall previous tasks
- Internal tools that blend AI with human decision-making
- Customer support bots that maintain conversation history across sessions

Why Are We Building This?

Most chatbots today are stateless — they respond to each message in isolation and forget everything that came before. This leads to frustrating user experiences, especially in real-world scenarios where continuity matters.

We're building this system to solve that problem.

By combining **LangGraph**, **LangChain**, and **Amazon Bedrock**, we aim to create a **memory-aware, modular, and production-ready AI assistant** that can:

- **Understand the full context** of a conversation, not just the latest input
- **Remember previous interactions**, even across sessions
- **Streamline AI development** using a graph-based design that's easy to debug and extend
- **Run efficiently and securely** using Claude 3 models via Amazon's serverless Bedrock infrastructure

This project isn't just a demo — it's a blueprint for building scalable, real-world AI systems that people can actually rely on.

How We Are Doing This

We're combining several modern AI tools to build a smart, memory-enabled assistant:

- **Amazon Bedrock + Claude 3**: Claude is the core language model, accessed securely and serverlessly via Amazon Bedrock.
- **LangChain**: Acts as a clean interface to Claude, handling the conversation inputs and outputs.
- **LangGraph**: Structures the conversation flow as a state machine, making it modular and easy to manage.
- **Checkpointing with BedrockSessionSaver**: Stores session data so conversations can be resumed later, even after long gaps.
- **Streaming Responses**: Delivers model replies in real time, creating a smooth, live chat experience.
- **Custom Tools**: Adds plug-and-play functions the assistant can use, like searching for shoes or performing tasks.

Together, this setup makes the assistant intelligent, persistent, and production-ready.

1. Setting Up the Environment

:- **Install dependency–**

```bash
pip install boto3 botocore langchain langchain-aws langgraph
```

:- **Import dependency–**

```
import boto3
from langgraph_checkpoint_aws.saver import BedrockSessionSaver
from botocore.config import Config
from langchain_aws import ChatBedrockConverse
from langchain_core.tools import tool
from langgraph.graph import StateGraph
from langchain_core.runnables import RunnableLambda
from langchain_core.messages import BaseMessage
```

Explanation:

We're importing all necessary libraries:

- `boto3`: To connect to AWS Bedrock.
- `ChatBedrockConverse`: Claude 3 wrapper from LangChain.
- `StateGraph` and `RunnableLambda`: Core parts of LangGraph — the engine behind our conversational flow.
- `BedrockSessionSaver`: Stores conversations as **resumable sessions**.

2. Connecting to Amazon Bedrock

```
# AWS region for the Bedrock client
region_name='us-east-1'
# Create Bedrock runtime client
bedrock_client = boto3.client(service_name='bedrock-runtime',
                              region_name=region_name,
                              config=Config(read_timeout=2000))
```

Explanation:

We're telling `boto3` to use **Bedrock's runtime API** in the `us-east-1` region. A longer `read_timeout` helps with big responses from Claude.

3. Initializing the Claude Model

```
# Initialize conversational AI model
llm = ChatBedrockConverse(
    model="anthropic.claude-3-sonnet-20240229-v1:0",
    temperature=0,
    max_tokens=None,
    client=bedrock_client,
)
```

Explanation:

We create a Claude 3 instance using `ChatBedrockConverse`. The context gives the model a system-level instruction ("You're a helpful assistant"). The temperature is `0`, meaning deterministic (non-random) responses.

4. Saving Sessions with LangGraph Checkpointing

```
# Saver for session state
session_saver = BedrockSessionSaver(
    region_name=region_name,
  )
```

Explanation:

This creates a checkpointing mechanism that stores:

- The conversation messages
- The Claude model's responses
- All graph state transitions

You'll be able to resume a previous conversation even after a system crash or refresh.

5. Context Injection and State Extraction

```
context = "you are a conversational AI, and the user is asking you a question"
```

```
def inject_context(state):  # Function to inject context into the state
    messages = state.get("messages", [])
    return {
        "context": context,
        "messages": messages
    }

context_loader = RunnableLambda(inject_context)  # Runnable lambda for context injection

def extract_messages(state: dict):  # Function to extract messages from state
    return state["messages"]

extract_messages_node = RunnableLambda(extract_messages)  # Runnable lambda for extracting messages
```

Explanation:

The `inject_context` function adds a predefined system context along with existing chat messages to the state dictionary.

`RunnableLambda(inject_context)` wraps that function so it can be used as a node in a LangGraph flow.

The `extract_messages` function simply pulls out the message history from the state.

`RunnableLambda(extract_messages)` also wraps this extractor function for graph use.

These functions help manage and prepare conversational data as it flows through the AI pipeline.

6. Building the LangGraph

```python
# Build graph
graph_builder = StateGraph(dict)  # Initialize state graph builder

# Add nodes
graph_builder.add_node("load_context", context_loader)  # Node to load context
graph_builder.add_node("extract_messages", extract_messages_node)  # Node to extract messages
graph_builder.add_node("llm", llm)  # Node for language model

# Connect nodes
graph_builder.add_edge("load_context", "extract_messages")  # Edge from context loader to message extractor
graph_builder.add_edge("extract_messages", "llm")  # Edge from message extractor to language model

# Set entry/exit
graph_builder.set_entry_point("load_context")  # Entry point of the graph
graph_builder.set_finish_point("llm")  # Finish point of the graph

# Compile graph
graph = graph_builder.compile(checkpointer=session_saver)  # Compile the graph with session saver
```

Explanation:

This is where we build the **actual flow** of the chatbot:

1. `load_context`: Adds the system prompt
2. `extract_messages`: Pulls user messages
3. `llm`: Sends them to Claude

Every transition is **checkpointed** automatically.

7. Creating a Session

```python
client = session_saver.session_client.client  # AWS client for session saver
# Create a new session
session_id = session_saver.session_client.client.create_session()["sessionId"]  # Create a new session ID

print(f"Session Created {session_id}")  # Print session creation confirmation


config = {"configurable": {"thread_id": session_id}}  # Configuration with thread ID
```

Explanation:

This creates a new **conversation session** on AWS and generates a `session_id`. This ID is passed to LangGraph so that everything — messages, checkpoints — is tracked under a single thread.

8. Running the Conversational Loop

```python
chat_history = []  # Initialize chat history
while True:
    user_input = input("User: ")  # Get user input
    if user_input.lower() in ["quit", "exit", "q"]:  # Check for exit commands
        print("Goodbye!")  # Print goodbye message
        break

    # Append user message
    chat_history.append(("user", user_input))  # Add user message to history

    # Run graph with current history
    for event in graph.stream(
        {"messages": chat_history},
        config
    ):
        for value in event.values():
            print("Raw value from event:", value,)  # Print raw event value

            if isinstance(value, BaseMessage):  # Check if value is a message
                print("Assistant:", value.content)  # Print assistant response
                chat_history.append(("ai", value.content))  # Add assistant response to history
```

Explanation:

This infinite loop waits for the user's input. If they type "quit", it ends the program.

- The user's message is added to the chat history.
- We run LangGraph, passing in both the `chat_history` and `config`.
- The graph returns an event stream with all state changes.
- If we get a Claude message (`BaseMessage`), we print it and store it in the chat.

9. Output

```
Session Created f0141b87-4b95-4c5f-87ea-f03f6c68f13f
User: hello, my name is Emma.
Assistant: Hello Emma, it's nice to meet you! I'm Claude, an AI assistant created by Anthropic. How are you doing tod
ay?
User: my age is 31.
Assistant: It's nice to meet you Emma. I don't actually have a specific age since I'm an AI assistant. But I'm happy
to chat and get to know you! Do you have any fun plans for the weekend?
User: what is my name?
Assistant: You said your name is Emma.
User: what is my age?
Assistant: You told me your age is 31.
User:
```

How Does It All Work?

Session Handling

Sessions are managed by `BedrockSessionSaver`, which:

- Saves conversation checkpoints at each node
- Assigns each conversation a `sessionId`
- Lets you **resume a conversation** later from the same point

Chunking (Graph Execution)

- The LangGraph breaks down your chatbot's logic into **nodes** and **edges**
- Each message goes through a small pipeline:
    - Add context → Extract user message → Ask Claude → Save output
- These steps are **modular and chunked**, making debugging and replaying easy

Summary:

This project implements a smart, memory-enabled conversational assistant using Amazon Bedrock, LangChain, and LangGraph. It integrates Claude 3 (Sonnet) via Bedrock to generate natural, contextual responses. The assistant maintains full conversation history using LangGraph's state machine and session checkpointing. Nodes in the graph handle specific tasks like loading context, extracting messages, and generating replies. Sessions are saved and resumed through `BedrockSessionSaver`, making the assistant capable of long-running conversations. The conversation is streamed in real time, providing an interactive user experience. Overall, the code builds a scalable and extensible chatbot that remembers and evolves with each session.