Building a Conversational TOSCA File Generator with Google's Gemini

Introduction

In the rapidly evolving world of cloud computing, deploying and managing applications across different platforms has become increasingly complex. While cloud systems offer tremendous benefits, many developers and system administrators struggle with the technical complexities of cloud orchestration and deployment automation. This is where TOSCA (Topology and Orchestration Specification for Cloud Applications) comes in - providing a standardized way to describe cloud applications. However, creating these files manually can be challenging and error-prone.

In this article, I'll walk you through how I built a conversational AI system using Google's Generative AI (Gemini) to help users create properly formatted TOSCA files through an interactive dialogue. You can find the complete implementation in this Jupyter notebook (https://github.com/yourusername/tosca-generator/blob/main/gen-ai-intensive-course-capstone-2025q1.ipynb).

The Challenge

TOSCA files are complex JSON documents that describe cloud applications' topology, relationships, and deployment requirements. Creating these files requires deep knowledge of:

- Cloud infrastructure components
- Network configurations
- Resource requirements
- Deployment specifications

Traditional approaches require users to:

1. Learn the TOSCA specification
2. Understand JSON structure
3. Manually create and validate files
4. Debug syntax errors

The Solution: A Conversational AI Assistant

I built a system that guides users through the TOSCA file creation process using natural language conversation. The system:

- Asks questions about the application requirements
- Validates inputs against TOSCA specifications
- Generates properly formatted JSON files
- Handles complex relationships between components

Key Components

The system uses a state graph architecture with specialized nodes:

```
# State Graph Architecture
graph_builder = StateGraph(RequestState)

# Add nodes to the graph
graph_builder.add_node("chatbot", chatbot_node)
graph_builder.add_node("converter", converter_node)
graph_builder.add_node("human", human_node)
graph_builder.add_node("converter_tools", tool_node)
```

The Conversation Flow

The system uses a structured approach to gather information:

```
# System message defining the conversation rules
CHATBOT_SYS_MSG = f"""
You are an inspector helping users fill out a form.
Retrieve the fields in the initial step and their formats from <FORMAT>.
Check the message history and ask one question at a time to fill each remaining field.

THE STEPS ARE AS FOLLOWS:
1. Ask about the fields in metadata.
2. Ask about the number of nodes.
3. Ask about the properties of each node...
"""

# State management for the conversation
class RequestState(TypedDict):
    messages: Annotated[list, add_messages]
    request: list[str]
    finished: bool
```

Leveraging GenAI Features

The implementation uses several advanced GenAI capabilities:

1. **Structured Output Generation**

```
converter_llm = ChatGoogleGenerativeAI(
    model="gemini-2.0-flash",
    response_format={"type": "json_object"}
)
```

2. **Function Calling**

```
@tool
def save_file(json_str: str) -> str:
    """Save the TOSCA file."""
    current_date = datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
    filename = f"result_{current_date}.json"
    with open(filename, "w") as file:
        json.dump(json_data, file, indent=4)
```

3. **State Management**

```
def chatbot_edges(state: RequestState) -> Literal["human", "converter"]:
    if msg.content.find("DONE") != -1:
        return "converter"
    else:
        return "human"
```

Implementation Details

1. Initialization and Setup

The system starts by setting up the necessary environment and dependencies:

```
# Install required packages
!pip install -qU 'fsspec==2024.10.0' 'async-timeout==4.0.3' 'langgraph==0.3.21' 'langchain-google-genai==2.1.2' 'langgraph-prebuilt==

# Initialize the GenAI client
client = genai.Client(api_key=GOOGLE_API_KEY)
```

2. TOSCA File Structure

The system supports a comprehensive TOSCA file structure:

```
format = {
    "metadata": {
        "tosca_definitions_version": "string",
        "description": "string",
        "template_name": {...},
        # ... other metadata fields
    },
    "nodes": [...],
    "virtual links": [...]
}
```

3. Conversation Management

The system uses a state graph to manage the conversation flow:

```
def chatbot_node(state: RequestState) -> RequestState:
    if state["messages"]:
        new_output = chatbot_llm.invoke(
            [CHATBOT_SYSTEM_INSTRUCTION] + state["messages"]
        )
    else:
        new_output = AIMessage(content=WELCOME_MSG)
    return state | {"messages": [new_output]}
```

Results and Benefits

The system provides several key benefits:

1. **Reduced Learning Curve**: Users don't need to learn TOSCA syntax
2. **Error Prevention**: Built-in validation ensures correct file structure
3. **Time Savings**: Automated file generation speeds up the process
4. **Flexibility**: Supports complex cloud application configurations

Future Improvements

Potential enhancements include:

- Support for more complex TOSCA features
- Integration with cloud platforms
- Visual representation of the generated topology
- Multi-language support

Accessing the Code

The complete implementation is available in this Jupyter notebook (https://github.com/yourusername/tosca-generator/blob/main/gen-ai-intensive-course-capstone-2025q1.ipynb). The notebook contains all the necessary code, including:

- Environment setup and dependencies
- State graph implementation
- Conversation flow management
- TOSCA file generation logic
- Example usage and testing

Feel free to clone the repository and experiment with the implementation. The notebook includes detailed comments and examples to help you understand each component.

Conclusion

This implementation demonstrates how GenAI can simplify complex technical tasks. By combining Google's Gemini with a state graph architecture, we've created a system that makes TOSCA file generation accessible to a wider range of users.

The code is available on GitHub, and I welcome contributions and feedback. Let me know if you'd like to explore any specific aspect of the implementation in more detail!

Note: This article is part of the GenAI Intensive Course Capstone 2025Q1. Special thanks to Google Generative AI, LangGraph, and LangChain for making this project possible.