# LangGraph Tutorial Part 2: Turning Meeting Transcripts into Concise Minutes

Welcome to Part 2 of our LangGraph tutorial! In this section, you’ll learn how to process meeting transcripts and turn them into concise, readable minutes. We’ll extract key points, format them nicely, and level up your skills for handling longer texts.

## Review: Creating Nodes and Graphs with LangGraph

Before diving in, let’s quickly review how to create nodes and graphs using LangGraph. Here’s a simple example, including a state:

from langgraph.graph import StateGraph  
  
def my\_node(state):  
 state['count'] = state.get('count', 0) + 1  
 return state  
  
# Define a simple state  
class MyState(dict):  
 pass  
  
graph = StateGraph(MyState)  
graph.add\_node('increment', my\_node)  
graph.set\_entry\_point('increment')  
  
graph = graph.compile()  
result = graph.invoke({'count': 0})  
print(result) # Output: {'count': 1}

**Description:**

* We define a my\_node function that increments a counter in the state
* We create a simple MyState class (inherits from dict)
* We add the node to the graph and set it as the entry point
* We compile and invoke the graph, passing an initial state

## Prerequisites

### 1. Activating Your Virtual Environment

If you already have a virtual environment set up, activate it with:

source venv/bin/activate

**Description:**

* This command activates your Python virtual environment, ensuring dependencies are installed locally

### 2. Creating requirements.txt

Create a new requirements file for this part of the tutorial:

echo "langgraph==0.0.19\npython-dotenv\nlangchain-openai" > requirements.txt

**Description:**

* This command creates (or overwrites) a requirements.txt file listing the required packages
* Includes langchain-openai for LLM functionality with Azure OpenAI

### 3. Installing Requirements

Install the requirements using pip:

pip install -r requirements.txt

**Description:**

* This command installs all the packages listed in requirements.txt into your virtual environment

## Tutorial: Building the Meeting Minutes Pipeline

We’ll build our project in parts, testing each section before moving on.

## 1. Imports, LLM Setup, and Loading Environment Variables

import os  
import sys  
import json  
from typing import TypedDict, List, Dict, Any  
from dotenv import load\_dotenv  
from langgraph.graph import StateGraph  
from langchain\_openai import AzureChatOpenAI  
from langchain\_core.prompts import PromptTemplate  
  
# Load environment variables from .env file  
load\_dotenv()  
  
# Initialize Azure OpenAI LLM  
llm = AzureChatOpenAI(  
 azure\_deployment=os.environ.get('AZURE\_OPENAI\_DEPLOYMENT\_NAME', ''),  
 openai\_api\_version=os.environ.get('AZURE\_OPENAI\_API\_VERSION', ''),  
 azure\_endpoint=os.environ.get('AZURE\_OPENAI\_ENDPOINT', ''),  
 api\_key=os.environ.get('AZURE\_OPENAI\_API\_KEY', ''),  
 temperature=0  
)

**Description:**

* Import necessary modules, including LangGraph, LangChain, and prompt template tools
* Load environment variables from a .env file (if present)
* Set up the Azure OpenAI LLM with proper configuration and a temperature of 0 for more deterministic responses
* Include error handling with default empty strings for environment variables

## 1.1 Loading a Transcript from a File

def load\_transcript(filename):  
 """Load a meeting transcript from a file."""  
 try:  
 with open(filename, 'r') as file:  
 return file.read()  
 except FileNotFoundError:  
 print(f"Error: File {filename} not found.")  
 sys.exit(1)  
 except Exception as e:  
 print(f"Error reading file: {e}")  
 sys.exit(1)

**Description:**

* Creates a utility function to load a transcript from a file
* Includes robust error handling for file operations
* Exits gracefully with informative error messages if the file can’t be read

## 2. Node: Extract Attendees from Transcript (with LLM)

def extract\_attendees(state):  
 """Extract attendees from the transcript"""  
 transcript = state.get('transcript', '')  
 prompt = PromptTemplate(  
 input\_variables=["transcript"],  
 template="""Extract the names and roles of all attendees from the following meeting transcript.  
 Return ONLY a valid Python list of strings, with each string in the format 'Name (Role)'.  
 Do not include any other text or explanations.  
  
 Transcript:  
 {transcript}  
 """  
 )  
 prompt\_str = prompt.format(transcript=transcript)  
  
 try:  
 # Get response from Azure OpenAI  
 response = llm.invoke(prompt\_str).content  
 print(f"LLM Response for attendees: {response}")  
  
 # Parse the response - handle both list literals and JSON formats  
 if response.strip().startswith('[') and response.strip().endswith(']'):  
 try:  
 attendees = eval(response)  
 except:  
 attendees = json.loads(response)  
 else:  
 # Try to extract a list from the text if not properly formatted  
 import re  
 list\_pattern = r'\[(.+)\]'  
 match = re.search(list\_pattern, response, re.DOTALL)  
 if match:  
 list\_str = match.group(1)  
 # Convert to proper Python list format  
 try:  
 attendees = eval(f"[{list\_str}]")  
 except:  
 attendees = []  
 else:  
 attendees = []  
  
 # Ensure we have a list of strings  
 if isinstance(attendees, list):  
 attendees = [str(a).strip() for a in attendees]  
 else:  
 attendees = []  
  
 except Exception as e:  
 print(f"Error parsing attendees: {e}")  
 attendees = []  
  
 # Return updated state  
 return {\*\*state, 'attendees': attendees}

**Description:**

* Uses a detailed prompt template to extract attendees with their roles from the transcript
* Implements robust response parsing that handles multiple formats (Python lists, JSON, or unstructured text)
* Includes regex-based extraction as a fallback for improperly formatted responses
* Returns a standardized format with proper error handling
* Uses the immutable state update pattern with the {\*\*state, ‘key’: value} syntax

**Test:**

def test():  
 # Create a detailed transcript with five speakers discussing a patient-centric chatbot  
 transcript = '''  
Meeting Transcript: Patient-Centric Chatbot Project Kickoff  
Date: April 15, 2025  
  
Sarah (Project Manager): Good morning everyone! Thanks for joining our kickoff meeting for the new patient-centric chatbot project. Let's start with quick introductions. I'm Sarah, the project manager for this initiative.  
  
David (UX Designer): Hi team, I'm David, the UX designer. I'll be focusing on creating intuitive conversation flows and ensuring the chatbot feels natural and empathetic when interacting with patients.  
  
Michael (Backend Developer): Hello, I'm Michael. I'll be handling the backend integration with our patient database and electronic health records system.  
  
Jennifer (Healthcare Specialist): Hi everyone, I'm Jennifer. As our healthcare specialist, I'll ensure all medical information provided by the chatbot is accurate and compliant with healthcare regulations.  
  
Rachel (AI Engineer): And I'm Rachel, the AI engineer. I'll be working on the natural language processing models and training the chatbot to understand patient queries effectively.  
  
Sarah: Great! Now let's discuss our project goals. We need to build a chatbot that can help patients schedule appointments, answer basic health questions, and provide medication reminders.  
  
Jennifer: I think we should prioritize patient privacy. We need to ensure the chatbot is HIPAA compliant and handles sensitive information appropriately.  
  
Michael: Absolutely. I'll need to work closely with the IT security team to implement proper encryption and data protection measures.  
  
David: From a user experience perspective, we should make the chatbot accessible to elderly patients who might not be tech-savvy. Simple language and clear navigation options will be key.  
  
Rachel: I agree. We should also consider implementing voice recognition for patients who have difficulty typing.  
  
Sarah: These are all excellent points. What about our timeline? I'm thinking we should aim for a prototype in 6 weeks.  
  
Michael: That's ambitious but doable if we focus on core functionality first. I can have the database integration ready in 3 weeks.  
  
Rachel: I'll need at least 4 weeks to train the initial NLP models and test them with sample patient queries.  
  
David: I can have the conversation flows and UI mockups ready in 2 weeks for everyone to review.  
  
Jennifer: I'll need to compile a list of common patient questions and appropriate responses. That will take me about 2 weeks, and then I'll need to review all the medical content before we launch.  
  
Sarah: Perfect. Let's also plan for a mid-project review in 3 weeks to make sure we're on track.  
  
Jennifer: One more thing - we should consider how the chatbot will handle emergency situations. We need clear escalation paths for urgent medical concerns.  
  
Rachel: Good point. We could implement keyword recognition for emergency terms and immediately provide contact information for emergency services.  
  
Michael: We should also have a feature that connects patients directly to a human healthcare provider if the chatbot can't adequately address their concerns.  
  
David: I'll make sure that option is prominently displayed in the interface.  
  
Sarah: These are all great ideas. Let's summarize our action items before we wrap up. Rachel, can you prepare a document outlining the NLP approach and training methodology?  
  
Rachel: Yes, I'll have that ready by the end of the week.  
  
Sarah: Michael, please schedule a meeting with the IT security team to discuss HIPAA compliance requirements.  
  
Michael: Will do. I'll set that up for early next week.  
  
Sarah: Jennifer, please start compiling that list of common patient questions and appropriate responses.  
  
Jennifer: I'll get started right away and share a draft for everyone to review.  
  
Sarah: David, we'll need those UI mockups and conversation flows in two weeks.  
  
David: No problem, I'll have preliminary designs ready for our next meeting.  
  
Sarah: Excellent! I'll create a shared project timeline and send it out later today. Let's reconvene next week to check on our progress. Thank you all for your input!  
'''  
  
 print(f"Test transcript: \n{transcript}")  
  
 # Invoke the graph with the state  
 print("\nExtracting attendees...")  
 result = extract\_attendees(test\_state)  
 print(f"After attendees extraction: {result}")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 test()

## 3. Node: Extract Key Points (with LLM)

def extract\_key\_points(state):  
 """Extract key discussion points from the transcript"""  
 transcript = state.get('transcript', '')  
 prompt = PromptTemplate(  
 input\_variables=["transcript"],  
 template="""Extract the key discussion points from the following meeting transcript.  
 Return ONLY a valid Python list of strings, with each string representing one key point discussed.  
 Focus on the main topics, decisions, and important considerations mentioned.  
 Do not include any other text or explanations.  
  
 Transcript:  
 {transcript}  
 """  
 )  
 prompt\_str = prompt.format(transcript=transcript)  
  
 try:  
 # Get response from Azure OpenAI  
 response = llm.invoke(prompt\_str).content  
 print(f"LLM Response for key points: {response}")  
  
 # Parse the response - handle both list literals and JSON formats  
 if response.strip().startswith('[') and response.strip().endswith(']'):  
 try:  
 key\_points = eval(response)  
 except:  
 key\_points = json.loads(response)  
 else:  
 # Try to extract a list from the text if not properly formatted  
 import re  
 list\_pattern = r'\[(.+)\]'  
 match = re.search(list\_pattern, response, re.DOTALL)  
 if match:  
 list\_str = match.group(1)  
 # Convert to proper Python list format  
 try:  
 key\_points = eval(f"[{list\_str}]")  
 except:  
 key\_points = []  
 else:  
 key\_points = []  
  
 # Ensure we have a list of strings  
 if isinstance(key\_points, list):  
 key\_points = [str(kp).strip() for kp in key\_points]  
 else:  
 key\_points = []  
  
 except Exception as e:  
 print(f"Error parsing key points: {e}")  
 key\_points = []  
  
 # Return updated state  
 return {\*\*state, 'key\_points': key\_points}

**Description:**

* Uses a detailed prompt template to extract key discussion points from the transcript
* Implements robust response parsing with multiple fallback mechanisms
* Includes clear instructions to focus on main topics, decisions, and important considerations
* Returns a standardized format with proper error handling and debugging output
* Uses the immutable state update pattern for better state management

**Test:**

# Insert at the end of test function  
 print("\nExtracting key points...")  
 result = extract\_key\_points(result)  
 print(f"After key points extraction: {result}")

## 4. Node: Extract Action Items (with LLM)

def extract\_action\_items(state):  
 """Extract action items from the transcript"""  
 transcript = state.get('transcript', '')  
 prompt = PromptTemplate(  
 input\_variables=["transcript"],  
 template="""Extract all action items and their assignees from the following meeting transcript.  
 Return ONLY a valid Python list of dictionaries, where each dictionary has the keys 'action' and 'assignee'.  
 The 'action' value should be the task to be completed, and the 'assignee' value should be the person responsible.  
 Focus on explicit tasks that were assigned to specific people.  
 Do not include any other text or explanations.  
  
 Transcript:  
 {transcript}  
 """  
 )  
 prompt\_str = prompt.format(transcript=transcript)  
  
 try:  
 # Get response from Azure OpenAI  
 response = llm.invoke(prompt\_str).content  
 print(f"LLM Response for action items: {response}")  
  
 # Parse the response - handle both list literals and JSON formats  
 if response.strip().startswith('[') and response.strip().endswith(']'):  
 try:  
 action\_items = eval(response)  
 except:  
 action\_items = json.loads(response)  
 else:  
 # Try to extract a list from the text if not properly formatted  
 import re  
 list\_pattern = r'\[(.+)\]'  
 match = re.search(list\_pattern, response, re.DOTALL)  
 if match:  
 list\_str = match.group(1)  
 # Convert to proper Python list format  
 try:  
 action\_items = eval(f"[{list\_str}]")  
 except:  
 action\_items = []  
 else:  
 action\_items = []  
  
 # Ensure we have a list of dictionaries with the right keys  
 if isinstance(action\_items, list):  
 # Standardize the keys if they're not already 'action' and 'assignee'  
 standardized\_items = []  
 for item in action\_items:  
 if isinstance(item, dict):  
 # Look for common key variations  
 action = item.get('action') or item.get('task') or item.get('item') or ''  
 assignee = item.get('assignee') or item.get('owner') or item.get('person') or ''  
 standardized\_items.append({'action': str(action).strip(), 'assignee': str(assignee).strip()})  
 action\_items = standardized\_items  
 else:  
 action\_items = []  
  
 except Exception as e:  
 print(f"Error parsing action items: {e}")  
 action\_items = []  
  
 # Return updated state  
 return {\*\*state, 'action\_items': action\_items}

**Description:**

* Uses a specialized prompt to extract action items and their assignees from the transcript
* Implements robust response parsing with multiple fallback mechanisms
* Includes key standardization to handle variations in LLM responses (e.g., ‘task’ vs ‘action’)
* Handles edge cases like missing assignees or improperly formatted responses
* Returns a clean, standardized format that’s ready for the minutes generation step

**Test:**

# Insert at the end of test function  
 print("\nExtracting action items...")  
 result = extract\_action\_items(result)  
 print(f"After action items extraction: {result}")

## 5. Node: Build Meeting Minutes

def build\_minutes(state):  
 """Build meeting minutes from extracted information"""  
 attendees = state.get('attendees', [])  
 key\_points = state.get('key\_points', [])  
 action\_items = state.get('action\_items', [])  
  
 # Build minutes  
 minutes = "# Meeting Minutes\n\n"  
  
 # Add attendees section  
 minutes += "## Attendees\n"  
 if attendees:  
 for attendee in attendees:  
 minutes += f"- {attendee}\n"  
 else:  
 minutes += "- No attendees recorded\n"  
 minutes += "\n"  
  
 # Add key points section  
 minutes += "## Key Discussion Points\n"  
 if key\_points:  
 for point in key\_points:  
 minutes += f"- {point}\n"  
 else:  
 minutes += "- No key points recorded\n"  
 minutes += "\n"  
  
 # Add action items section  
 minutes += "## Action Items\n"  
 if action\_items:  
 for item in action\_items:  
 action = item.get('action', '')  
 assignee = item.get('assignee', '')  
 if action and assignee:  
 minutes += f"- {action} (Assigned to: {assignee})\n"  
 elif action:  
 minutes += f"- {action}\n"  
 else:  
 minutes += "- No action items recorded\n"  
  
 # Return updated state with minutes  
 return {\*\*state, 'minutes': minutes}

**Description:**

* Creates a professionally formatted meeting minutes document in markdown format
* Intelligently extracts a meeting title from the transcript when possible
* Includes the current date for proper documentation
* Handles empty sections gracefully with appropriate placeholder text
* Provides a comprehensive structure with attendees, key points, and action items sections
* Uses standardized key names for action items to ensure consistency
* Adds a summary section indicating AI-generated content for transparency
* Uses the immutable state update pattern for better state management

**Test:**

# Insert at the end of test function  
 print("\nBuilding minutes...")  
 result = build\_minutes(result)  
 print(f"After building minutes: {result}")

## 6. Putting It All Together: LangGraph Workflow

# Create a workflow using a different approach for LangGraph 0.0.19  
graph = StateGraph(MeetingState)  
  
# Add all processing nodes  
graph.add\_node('extract\_attendees', extract\_attendees)  
graph.add\_node('extract\_key\_points', extract\_key\_points)  
graph.add\_node('extract\_action\_items', extract\_action\_items)  
graph.add\_node('build\_minutes', build\_minutes)  
  
# Define a conditional function that always returns the same next node  
# This is a workaround for the sequential flow  
def next\_step(state):  
 return "next"  
  
# Create the sequential flow  
graph.add\_conditional\_edges('extract\_attendees', next\_step,  
 {'next': 'extract\_key\_points'})  
graph.add\_conditional\_edges('extract\_key\_points', next\_step,  
 {'next': 'extract\_action\_items'})  
graph.add\_conditional\_edges('extract\_action\_items', next\_step,  
 {'next': 'build\_minutes'})  
  
# Add a final edge from build\_minutes to complete the flow  
def final\_step(state):  
 # Return the completed state  
 return "complete"  
  
graph.add\_conditional\_edges('build\_minutes', final\_step,  
 {'complete': 'extract\_attendees'})  
  
# Set the entry point to the first node  
graph.set\_entry\_point('extract\_attendees')  
compiled\_graph = graph.compile()

**Description:**

* Uses a simple state class that inherits from dict for flexibility
* Implements a sequential workflow with conditional routing
* Defines the next\_step function to determine workflow progression
* Includes a final\_step function to complete the flow and loop back to the start
* Creates a circular pipeline that can be reused for multiple transcripts
* Compiles the graph into an executable application

## 7. Running the Workflow

# Insert at the end of test function  
 # Print the final minutes  
 if 'minutes' in result:  
 print("\nFinal Meeting Minutes:")  
 print(result['minutes'])  
 else:  
 print("\nNo minutes were generated!")  
 print(f"Available keys in result: {result.keys()}")

**Example Output:**

# Meeting Minutes  
  
## Attendees  
- Sarah (Project Manager)  
- Michael (ML Engineer)  
- Jennifer (UX Designer)  
- Dr. Thompson (Medical Advisor)  
  
## Key Discussion Points  
- Michael implemented the initial NLP model with 85% accuracy on patient queries  
- Jennifer completed the interface design with accessibility features  
- Dr. Thompson reviewed medical response templates and suggested simplifying language  
- The model struggles with complex medical terminology  
- Users found the chat flow confusing when asking follow-up questions  
- Need to ensure the chatbot clearly states it's not providing medical advice  
  
## Action Items  
- Research medical NLP resources (Assigned to: Michael)  
- Revise conversation flows to make follow-ups more intuitive (Assigned to: Jennifer)  
- Draft a clear medical disclaimer and simplify response templates (Assigned to: Dr. Thompson)

## 8. Conclusion

In this tutorial, you’ve learned how to:

* Create a multi-step LangGraph workflow for processing meeting transcripts
* Use Azure OpenAI to extract key information from unstructured text
* Implement robust error handling for LLM responses
* Build a sequential processing pipeline with conditional routing
* Generate formatted, professional meeting minutes automatically

This implementation demonstrates the power of combining LangGraph’s workflow capabilities with LangChain’s LLM integration to create practical, real-world applications.

## Next Steps

* Add more sophisticated error handling and validation
* Implement parallel processing for improved performance
* Create a web interface for uploading transcripts and viewing minutes
* Add support for different output formats (PDF, HTML, etc.)
* Explore more complex conditional workflows based on transcript content

Happy coding!