# LangGraph Tutorial for Students: Building AI Workflows

Welcome to this beginner-friendly tutorial on **LangGraph**, a powerful framework for creating AI workflows! LangGraph is part of the LangChain ecosystem and helps you build applications where multiple AI tasks (or “agents”) work together, like a flowchart for smart systems. Whether you’re new to coding or an experienced developer, this guide will walk you through practical examples over three weeks. We’ll start with a simple app this week and build up to more complex projects.

In this tutorial, you’ll learn:

* What LangGraph is and why it’s useful.
* How to build a **Question Processor** app that answers questions, classifies them, and generates tags—all in parallel—then packages the results into a neat JSON file.
* Hands-on coding with clear steps and tests to ensure everything works.

Let’s dive in!

## What is LangGraph?

LangGraph is a tool for designing workflows where AI agents collaborate to complete tasks. Imagine it as a coordinator that manages different steps—like answering a question, tagging it, or summarizing text—and keeps everything organized. It’s great for:

* **Memory**: Remembering past steps or conversations.
* **Parallel Tasks**: Running multiple processes at once to save time.
* **Human-in-the-Loop**: Letting you jump in to make decisions.
* **Saving Progress**: Storing work so you can pick up where you left off.

Big companies like Klarna (for customer support bots) and Uber (for generating code) use LangGraph in real-world projects, proving its value. Over the next three weeks, we’ll explore three fun use cases:

1. **Week 1**: A Question Processor (this week!).
2. **Week 2**: Turning meeting transcripts into concise minutes.
3. **Week 3**: Refining the tone of your writing.

## Part 1: Building a Question Processor

This week, we’ll create an app that:

* Takes a question (e.g., “What is photosynthesis?”).
* Processes it in parallel to:
  + Generate an answer.
  + Classify it into a category (e.g., “science”).
  + Create tags with weights (e.g., “plants: 0.9”).
* Combines everything into a JSON document.

We’ll break it into bite-sized sections with code, explanations, and tests.

### Setup and Prerequisites

Before we start coding, let’s set up our environment:

* **Python**: You need Python 3.11 or higher installed.
* **Virtual Environment**: We’ll create a dedicated environment for our project.
* **Packages**: We’ll use langgraph and langchain-openai.
* **API Key**: An OpenAI API key is optional (skip it if you use a local model).

#### Create a Virtual Environment

First, let’s create and activate a Python virtual environment:

# Create a virtual environment named 'venv'  
python3.11 -m venv venv  
  
# Activate the virtual environment  
# On macOS/Linux:  
source venv/bin/activate  
# On Windows:  
# venv\Scripts\activate  
  
# To deactivate the environment when you're done:  
# deactivate

#### Set Up Dependencies

Create a requirements.txt file with the following content:

langgraph==0.0.19  
langchain-openai==0.0.2 # Contains both OpenAI and AzureOpenAI integrations  
python-dotenv==1.0.1

Then install the packages:

pip install -r requirements.txt

#### Environment Variables

Create a .env file in your project root to store your API keys securely:

# Azure OpenAI API Configuration  
AZURE\_OPENAI\_API\_KEY=your\_key\_here  
AZURE\_OPENAI\_ENDPOINT=your\_endpoint\_here  
AZURE\_OPENAI\_API\_VERSION=2023-05-15  
AZURE\_OPENAI\_DEPLOYMENT\_NAME=your\_deployment\_name  
  
# Other configuration variables  
# TEMPERATURE=0.7

(Replace your\_key\_here with your actual key from OpenAI.)

To load these environment variables in your code, you’ll need to add:

from dotenv import load\_dotenv  
load\_dotenv() # This loads the variables from .env

## Create a file called question\_processor.py—this is where all our code will live

### Section 1: Define State and Nodes

First, we’ll define the “state” (data that flows through our app) and “nodes” (tasks like answering or tagging). Open question\_processor.py and add this code:

from typing import Annotated, TypedDict  
import os  
from langgraph.graph import StateGraph, END  
from langchain\_openai import AzureChatOpenAI  
from pydantic import BaseModel, Field  
  
# Define the state: what data we’ll track  
class State(TypedDict):  
 question: str # The input question  
 answer: str = "" # The generated answer  
 category: str = "" # The category (e.g., "science")  
 tags: list = [] # List of tags with weights  
  
# Define structured outputs using Pydantic  
class Answer(BaseModel):  
 answer: str = Field(description="Readable answer to the question")  
  
class Category(BaseModel):  
 category: str = Field(description="Single word category, e.g., 'science'")  
  
class Tags(BaseModel):  
 tags: list = Field(description="List of dicts with 'tag' and 'weight' (0-1)")  
  
# Set up the language model (we’re using OpenAI’s GPT-4o)  
llm = AzureChatOpenAI(  
 azure\_deployment=os.environ.get("AZURE\_OPENAI\_DEPLOYMENT\_NAME"),  
 openai\_api\_version=os.environ.get("AZURE\_OPENAI\_API\_VERSION")  
)  
  
# Node to generate an answer  
def answer\_node(state: State):  
 prompt = f"Answer this question in a readable way: {state['question']}"  
 response = llm.invoke(prompt).content  
 return {"answer": response}  
  
# Node to classify the question  
def classify\_node(state: State):  
 prompt = f"Classify this question into a single word category: {state['question']}"  
 response = llm.invoke(prompt).content  
 return {"category": response}  
  
# Node to generate tags  
def tag\_node(state: State):  
 prompt = f"Generate 4 tags for this question with weights (0-1) showing importance, in JSON format: {state['question']}"  
 response = llm.invoke(prompt).content  
 import json  
 tags = json.loads(response)  
 return {"tags": tags}

#### Code Explanation

* **Imports**: We bring in tools for typing (TypedDict), the graph framework (StateGraph), the OpenAI model (ChatOpenAI), and structured data (BaseModel).
* **State**: A TypedDict called State holds our data: the question, answer, category, and tags. Default values are empty so we can fill them later.
* **Pydantic Models**: These (e.g., Answer, Category) ensure our outputs are structured and easy to check.
* **Language Model**: llm is our AI brain, set to use OpenAI’s “gpt-4o” model. You can swap this for another model if you prefer.
* **Nodes**:
  + answer\_node: Takes the question from state and asks the AI for a readable answer.
  + classify\_node: Asks for a one-word category.
  + tag\_node: Requests four tags in JSON format, then parses them into a Python list.

#### Test It Out

Add this at the bottom of your file and run it:

state = {"question": "What is photosynthesis?"}  
print(answer\_node(state)) # Outputs something like {"answer": "Photosynthesis is..."}  
print(classify\_node(state)) # Outputs {"category": "science"}  
print(tag\_node(state)) # Outputs {"tags": [{"tag": "plants", "weight": 0.9}, ...]}

Check the outputs:

* Is the answer clear and readable?
* Does the category make sense?
* Are there four tags with weights between 0 and 1?

### Section 2: Build the Graph with Parallel Execution

Now, let’s connect our nodes into a graph where tasks run in parallel. Add this code to question\_processor.py:

# Create the graph  
graph = StateGraph(State)  
  
# Add nodes to the graph  
graph.add\_node("answer", answer\_node)  
graph.add\_node("classify", classify\_node)  
graph.add\_node("tag", tag\_node)  
graph.add\_node("combine", lambda x: x) # Temporary placeholder  
  
# Set the starting point and parallel edges  
graph.add\_edge("start", "answer")  
graph.add\_edge("start", "classify")  
graph.add\_edge("start", "tag")  
  
# Connect all nodes to "combine"  
graph.add\_edge("answer", "combine")  
graph.add\_edge("classify", "combine")  
graph.add\_edge("tag", "combine")  
  
# Define entry and exit points  
graph.set\_entry\_point("start")  
graph.set\_finish\_point("combine")  
  
# Compile the graph into an app  
app = graph.compile()

#### Code Explanation

* **Graph Creation**: StateGraph(State) sets up our workflow using the State we defined.
* **Adding Nodes**: Each node (answer, classify, tag) is added with its function. The combine node is a placeholder for now.
* **Edges**:
  + graph.add\_edge("start", "answer") means when the graph starts, it kicks off the answer\_node.
  + We add edges from “start” to all three nodes, so they run at the same time (parallel execution!).
  + After finishing, each node connects to “combine” to bring the results together.
* **Entry and Finish**: “start” is where we begin, and “combine” is where we end.
* **Compile**: app = graph.compile() turns our graph into a runnable program.

#### Test It Out

Try this at the bottom of your file:

result = app.invoke({"question": "What is photosynthesis?"})  
print(result)

You should see a dictionary with the question, answer, category, and tags. Since combine is a placeholder, it just passes the state through. Check:

* Did all three tasks (answer, classify, tag) run?
* Are the results stored in result?

### Section 3: Combine Results into JSON

Let’s finish by updating the combine node to package everything into a JSON-friendly format. Replace the placeholder combine node with this:

# Node to combine results into a JSON-like structure  
def combine\_node(state: State):  
 return {  
 "response": {  
 "question": state["question"],  
 "answer": state["answer"],  
 "category": state["category"],  
 "tags": state["tags"]  
 }  
 }  
  
# Update the graph with the new combine node  
graph.add\_node("combine", combine\_node)  
app = graph.compile() # Recompile after updating

#### Code Explanation

* **Combine Node**: combine\_node takes the state and builds a nested dictionary under a "response" key. This makes it easy to turn into JSON later.
* **Update Graph**: We overwrite the old combine node and recompile the app to use the new version.

#### Test It Out

Run this:

result = app.invoke({"question": "What is photosynthesis?"})  
import json  
print(json.dumps(result, indent=2))

You’ll get output like:

{  
 "response": {  
 "question": "What is photosynthesis?",  
 "answer": "Photosynthesis is the process by which plants use sunlight to make food.",  
 "category": "science",  
 "tags": [  
 {"tag": "plants", "weight": 0.9},  
 {"tag": "sunlight", "weight": 0.8},  
 {"tag": "biology", "weight": 0.7},  
 {"tag": "energy", "weight": 0.6}  
 ]  
 }  
}

Check:

* Is the JSON well-formed (no errors when printing)?
* Do the answer, category, and tags match the question?

#### Final Test

Let’s try multiple questions:

questions = ["What is photosynthesis?", "How do rockets work?", "Tell me about history"]  
for q in questions:  
 result = app.invoke({"question": q})  
 print(json.dumps(result, indent=2))

Look at the outputs:

* Are the answers relevant?
* Do the categories and tags fit each question?

## What’s Next?

Great job building your first LangGraph app! In **Week 2**, we’ll tackle a new challenge: turning messy meeting transcripts into concise minutes. You’ll learn to extract key points and format them nicely, leveling up your skills with longer texts.

Here’s a quick summary of what we did today:

| **Step** | **What We Did** | **How to Test** |
| --- | --- | --- |
| Setup | Installed packages, set up environment | Check pip install worked |
| Section 1 | Defined state and nodes | Test each node individually |
| Section 2 | Built a graph with parallel tasks | Run the graph, check all outputs |
| Section 3 | Combined results into JSON | Verify JSON format and accuracy |
| Final Test | Ran multiple questions | Ensure consistent, relevant results |

Keep your question\_processor.py file handy—you’re now a LangGraph explorer! See you next week for Part 2. Happy coding!