# A Gentle Introduction to LangGraph
# A Step-by-Step Tutorial

A Quick Overview

Instructor: A. Namin

# What is LangGraph?

- Definition:
  - An orchestration framework for building complex agentic systems
- Key Characteristics:
  - **Low-level control**: More granular than LangChain agents
  - **Deterministic flows**: Pre-defined execution paths guaranteed every time
  - **Graph-based**: Workflows represented as directed graphs with nodes and edges
  - **State management**: Structured state shared across all nodes

# LangGraph vs LangChain

- LangChain Agents:
  - Dynamic reasoning at runtime
  - Plan changes based on observations
  - Less control over execution flow
  - Good for exploratory tasks
- LangGraph:
  - Pre-defined workflow structure
  - Guaranteed execution path
  - Full control over flow Production-ready determinism
- Key Insight:
  - Use LangGraph when you need predictable, controlled workflows that must follow specific business logic every time.

# Core Concepts in LangGraph

- 1. State
  - A shared data structure (TypedDict) that flows through the graph
- 2. Nodes
  - Functions that process and update the state
- 3. Edges
  - Connections defining flow between nodes
- 4. Conditional Edges
  - Decision points that route to different nodes based on conditions
- 5. Graph Builder
  - Tool to construct and compile the workflow.

# Best Practice - Bottom-Up Development

- Step-by-Step Workflow:
  - Step 1: Draw the graph on paper first
  - Step 2: Define nodes using add_node()
  - Step 3: Create edges with add_edge()
  - Step 4: Add conditional edges if needed
  - Step 5: Define the State class
  - Step 6: Implement node functions
  - Step 7: Compile and visualize
  - Step 8: Test and iterate
- Pro Tip: Always start with a visual diagram. It helps you think through the logic before writing code.

# Tutorial Example - Customer Feedback Processor

- Business Problem:
  - Process social media comments to identify questions vs compliments and route them appropriately
- Workflow Overview:

```
START
  ↓
Extract Content
  ↓
Route (Question/Compliment)
  ↓ Question → Run Question Code
  ↓ Compliment → Run Compliment Code
  ↓
Beautify
  ↓
END
```

Input: API payload with customer remarks, timestamps, social media channel, etc.

# Step 1 - Creating Nodes Code Implementation

```
from langgraph.graph import StateGraph

graph_builder = StateGraph(State)

# Add all nodes
graph_builder.add_node("extract_content", extract_content)
graph_builder.add_node("run_question_code", run_question_code)
graph_builder.add_node("run_compliment_code", run_compliment_code)
graph_builder.add_node("beautify", beautify)
```

Important Notes:

- START and END nodes are built-in (no need to create them)
- First argument = node name (string)
- Second argument = Python function name
- Names can differ but keeping them identical improves readability

# Step 2 - Creating Standard Edges Code Implementation

```
from langgraph.graph import END, START

# Define edges (connections between nodes)
graph_builder.add_edge(START, "extract_content")
graph_builder.add_edge("run_question_code", "beautify")
graph_builder.add_edge("run_compliment_code", "beautify")
graph_builder.add_edge("beautify", END)
```

- What are Edges?
  - Edges define the flow of execution from one node to another. They create the graph structure.
- Remember:
  - Use node names (strings), not function names when adding edges!

# Step 3 - Creating Conditional Edges Implementation

```
graph_builder.add_conditional_edges(
    "extract_content",            # Source node
    route_question_or_compliment,   # Routing function
    {
        "compliment": "run_compliment_code",
        "question": "run_question_code",
    },
)
```

Three Required Arguments:

1. Source node: Where the conditional check happens
2. Routing function: Contains conditional logic, returns a string
3. Route mapping: Dictionary mapping return values to target nodes

Key Point: The routing function must return one of the keys in the mapping dictionary.

# Step 4 - Defining the State Class Implementation

```
from typing_extensions import TypedDict

class State(TypedDict):
    text: str                    # Extracted content
    answer: str                  # Final response
    payload: dict[str, list]     # Input data
```

- State Purpose:
  - Centralized data storage: All variables accessible across nodes
  - Type safety: TypedDict provides type hints
  - Data flow: Information passes between nodes via state
  - Immutability: Each node returns updated values, doesn't mutate directly
- Best Practice: Define all variables you'll need upfront in the State class.

## Step 5 - Implementing Node Functions (Part 1)
## Implementation

- Extract Content Node:

```
def extract_content(state: State):
   # Access payload from state
   # Extract customer_remark field
   return {"text": state["payload"][0]["customer_remark"]}
```

- Routing Function:

```
def route_question_or_compliment(state: State):
   if "?" in state["text"]:
      return "question"
   else:
      return "compliment”
```

Pattern: Every node function:
Takes state: State as parameter
Returns a dictionary with updated state variables
Accesses existing state via state["variable_name"]

## Step 5 - Implementing Node Functions (Part 2)

## Implementation

- Action Nodes:

```
def run_compliment_code(state: State):
    return {"answer": "Thanks for the compliment."}

def run_question_code(state: State):
    return {"answer": "Wow nice question."}
```

- Beautify Node:

```
def beautify(state: State):
    # Access current answer and modify it
    return {"answer": state["answer"] + " beautified"}
```

Note: By default, returning a dictionary with an existing key overwrites that variable in the state. We'll learn how to append instead in upcoming slides.

## Step 6 - Compiling and Visualizing Code to Compile and Display

```
# Compile the graph
graph = graph_builder.compile()

# Visualize the graph
from IPython.display import Image, display
display(Image(graph.get_graph().draw_mermaid_png()))
```

- Visual Indicators:
  - Solid Edges: Always executed in the workflow
  - Dotted Edges: Conditional - only one branch executes
- Pro Tip: Always visualize your graph before running to verify the structure matches your design.

## Step 7 - Executing the Graph

Using invoke():

```
result = graph.invoke({
    "payload": [{
        "time_of_comment": "20-01-2025",
        "customer_remark": "I hate this.",
        "social_media_channel": "facebook",
        "number_of_likes": 100,
    }]
})

# Output shows final state
print(result)
```

Output Example:

```
{
    'text': 'I hate this.',
    'answer': 'Thanks for the compliment. beautified',
    'payload': [...]
}
```

invoke() returns: The complete final state after graph execution

# Monitoring Execution - Using stream()

```
for step in graph.stream({
    "payload": [{
        "customer_remark": "I hate this.",
        ...
    }]
}):
    print(step)
```

Output (step-by-step):

```
{'extract_content': {'text': 'I hate this.'}}
{'run_compliment_code': {'answer': 'Thanks for the compliment.'}}
{'beautify': {'answer': 'Thanks for the compliment. beautified'}}
```

Use Case: stream() is perfect for:

- Showing progress bars in UI
- Debugging node-by-node execution
- Real-time updates to users

# Advanced - Appending vs Overwriting State

The Problem: By default, updating a state variable overwrites its value. Sometimes you want to append instead.

Solution: Annotated Types

```
from typing import Annotated
import operator

class State(TypedDict):
    text: str
    # Changed from str to list with operator.add
    answer: Annotated[list, operator.add]
    payload: dict[str, list]
```

- Annotated Format: Annotated[data_type, operator]
  - operator.add works with lists, strings, and numbers
  - Each return value gets appended instead of replacing

# Updating Functions for Appending

Modified Node Functions:

```
def run_compliment_code(state: State):
    # Return list instead of string
    return {"answer": ["Thanks for the compliment."]}

def run_question_code(state: State):
    return {"answer": ["Wow nice question."]}
```

Modified Beautify Function:

```
def beautify(state: State):
    # Access last item in list
    last_answer = state["answer"][-1]
    return {"answer": [last_answer + " beautified"]}
```

Result: Now answer contains both intermediate and final values: ['Thanks for the compliment.', 'Thanks for the compliment. beautified']

## Custom Operators for Complex Types

The Problem: operator.add doesn't work with dictionaries. For complex data structures, you need custom merge logic.

Creating a Custom Operator:

```
def merge_dicts(dict1, dict2):
    # Merge two dictionaries
    return {**dict1, **dict2}

class State(TypedDict):
    text: str
    answer: Annotated[dict, merge_dicts]  # Custom operator
    payload: dict[str, list]
```

Power Tip: You can create custom operators for any complex merge logic - nested dictionaries, custom objects, etc.

## Using Custom Operators in Practice

Updated Node Functions:

```
def run_compliment_code(state: State):
    return {"answer": {"temp_answer": "Thanks for the compliment."}}

def beautify(state: State):
    return {
        "answer": {
            "final_beautified_answer":
                state["answer"]["temp_answer"] + " beautified"
        }
    }
```

## Using Custom Operators in Practice

Final Output:

```
{
    'answer': {
        'temp_answer': 'Thanks for the compliment.',
        'final_beautified_answer': 'Thanks for the compliment. beautified'
    }
}
```

## Parallel Node Execution - Use Case

New Requirement: Tag the type of customer remark (packaging, sustainability, medical) while determining if it's a question or compliment.

Enhanced Workflow:

```
START
  ↓
Extract Content
  ↓ (parallel execution)
   ├–→ Tag Query → ┐
  └→ Route       → Beautify → END
     ↓ Question → Run Question Code ↑
     ↓ Compliment → Run Compliment Code ↑
```

Benefit: Nodes execute concurrently in the same <u>superstep</u> for efficiency.

# Implementing Parallel Execution

Add New Node and Edge:

```
graph_builder.add_node("tag_query", tag_query)
graph_builder.add_edge("tag_query", "beautify")
```

Tag Query Function:

```
def tag_query(state: State):
    if "package" in state["text"]:
        return {"tag": "Packaging"}
    elif "price" in state["text"]:
        return {"tag": "Pricing"}
    else:
        return {"tag": "General"}
```

# Implementing Parallel Execution

Update State:

```
class State(TypedDict):
    text: str
    tag: str  # New variable
    answer: Annotated[dict, merge_dicts]
    payload: dict[str, list]
```

## Using Parallel Results

Updated Beautify Function:

```
def beautify(state: State):
    return {
        "answer": {
            "final_beautified_answer":
                state["answer"]["temp_answer"] +
                f' I will pass it to the {state["tag"]} Department'
        }
    }
```

Final Output:

```
{
    'tag': 'General',
    'answer': {
        'final_beautified_answer':
            'Thanks for the compliment. I will pass it to the General Department'
    }
}
```

# Advanced LLM Integration

## Enhanced Routing with LLM

```
from langchain_openai import AzureChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser

llm = AzureChatOpenAI(
    deployment_name="gpt-4o",
    model_name="gpt-4o",
    temperature=0.1,
)

template = """
I have a piece of text: {text}.
Tell me whether it is a 'compliment' or a 'question'.
"""

prompt = ChatPromptTemplate([("user", template)])
chain = prompt | llm | StrOutputParser()

def route_question_or_compliment(state: State):
    response = chain.invoke({"text": state["text"]})
    return response
```

# Key Takeaways

Core Principles:

1. Start with a diagram - visualize before coding
2. State is central - all data flows through it
3. Nodes are functions - they transform state
4. Edges define flow - control execution order
5. Conditional edges - enable decision making
6. Operators control merging - append vs overwrite

When to Use LangGraph:

- Production workflows requiring deterministic execution
- Complex multi-step AI agent workflows
- When you need full control over execution flow
- Building reliable, reproducible AI systems:

# Next Steps and Resources

What We Covered:

- Basic graph construction
- State management ? Conditional routing
- Custom operators
- Parallel execution
- LLM integration

Next Steps:

- Part 2: RAG-based workflows with LangGraph
- Using the Send API for advanced map-reduce patterns
- Building production-ready AI agents
- Implementing human-in-the-loop workflows

Resources:

- GitHub: V-Sher/LangGraphTutorial LangChain
- Documentation: docs.langchain.com
- Original Article: levelup.gitconnected.com

# HANDS-ON EXERCISE SLIDES

EXERCISE 1: Build Your First Graph

Task: Create a simple graph that:

1. Takes a customer email as input
2. Classifies it as "urgent" or "routine"
3. Routes to different response nodes
4. Combines final response

Hints:

- Use 4 nodes: classify, urgent_response, routine_response, format_output
- Use conditional edge after classify
- Test with sample emails

# HANDS-ON EXERCISE SLIDES

EXERCISE 2: Implement State Appending

Task: Modify the graph to:

1. Keep track of all processing steps
2. Store intermediate results
3. Return complete history

Requirements:

- Use Annotated[list, operator.add]
- Each node should append to history
- Final output shows full processing chain

# HANDS-ON EXERCISE SLIDES

EXERCISE 3: Create Custom Operator

Task: Build a custom operator that:

1. Merges nested dictionaries intelligently
2. Handles conflicts by keeping most recent value
3. Maintains metadata timestamps

```
def smart_merge(dict1, dict2):
    # Your implementation here
    pass
```

# HANDS-ON EXERCISE SLIDES

EXERCISE 4: Parallel Processing

Task: Create a graph that processes data in parallel:

1. Extract content from input
2. Simultaneously: (a) Analyze sentiment, (b) Extract keywords, (c) Detect language
3. Combine all results in final node

Bonus: Add timing information to see parallel speedup

# DEBUGGING TIPS

Common Issues:

1. State variables not updating
   1. Check you're returning dictionary from functions
   2. Verify variable names match State class
2. Conditional edge errors
   1. Ensure routing function returns exact key from mapping
   2. Check for typos in node names
3. Graph visualization shows unexpected flow
   1. Review edge definitions
   2. Verify conditional logic
4. Import errors
   1. Install: pip install langgraph langchain
   2. Check Python version (3.9+)

# Complete Code

```
from typing_extensions import TypedDict
from typing import Annotated
import operator
from langgraph.graph import StateGraph, END, START

# Define State
class State(TypedDict):
    text: str
    answer: Annotated[list, operator.add]
    payload: dict[str, list]

# Define node functions
def extract_content(state: State):
    return {"text": state["payload"][0]["customer_remark"]}

def route_question_or_compliment(state: State):
    if "?" in state["text"]:
        return "question"
    else:
        return "compliment"

def run_compliment_code(state: State):
    return {"answer": ["Thanks for the compliment."]}

def run_question_code(state: State):
    return {"answer": ["Wow nice question."]}

def beautify(state: State):
    return {"answer": [state["answer"][-1] + " beautified"]}

# Build graph
graph_builder = StateGraph(State)

# Add nodes
graph_builder.add_node("extract_content", extract_content)
graph_builder.add_node("run_question_code", run_question_code)
```

# References

- **Gentle Introduction to LangGraph: A Step-by-Step Tutorial (**Dr. Varshita Sher)
  - https://levelup.gitconnected.com/gentle-introduction-to-langgraph-a-step-by-step-tutorial-2b314c967d3c?gi=fbded980ec3f
- Cloud LLM used for slides generation