Complete LangGraph Sequential Workflow Documentation

📖 Table of Contents

📚 Introduction

This documentation covers a **LangGraph Sequential Workflow Agent** that demonstrates how to build production-ready agentic AI systems. The agent:

- Retrieves documents from a knowledge base
- Generates reasoning using Claude/OpenAI
- Produces validated answers
- Tracks workflow progression

Use Cases:

- Customer support chatbots
- Research assistants
- Question-answering systems
- Document analysis pipelines
- Automated report generation

🔧 Prerequisites

Before starting, ensure you have:

1. **Python 3.9+** installed
2. **OpenAI API Key** (get from https://platform.openai.com/api-keys)
3. **Basic Python knowledge** (functions, dictionaries, type hints)
4. **Understanding of LLMs** (what models are and how they work)

Required Python Version Check

```bash
python --version  # Should be 3.9 or higher
```

🏗️ Architecture Overview

```
┌─────────────────────────────────────────────────────────┐
│              USER QUERY INPUT                            │
└────────────────────────┬────────────────────────────────┘
        │
        STATE CREATED
        │
        ▼
    ┌──────────────────────┐
    │ RETRIEVE DOCUMENTS   │ ◄─── Knowledge Base
    │ (Step 1)             │
    └──────────┬───────────┘
        │
        ▼
    ┌──────────────────────┐
    │ GENERATE REASONING   │ ◄─── OpenAI API
    │ (Step 2)             │
    └──────────┬───────────┘
        │
        ▼
    ┌──────────────────────┐
    │ GENERATE ANSWER      │ ◄─── OpenAI API
```

```
    │  (Step 3)       │
    └────────┬────────┘
             │
             ▼
    ┌─────────────────┐
    │ VALIDATE ANSWER │ ◄──── OpenAI API
    │  (Step 4)       │
    └────────┬────────┘
             │
             ▼
   ┌──────────────────────┐
   │ FINAL STATE RETURNED │
   │ (with all results)   │
   └──────────────────────┘
```

🎯 Core Concepts

1. State Management

What is State? State is the data structure that flows through your agent. It carries information between nodes.

python

```python
class AgentState(TypedDict):
    input: str                        # User's question
    steps: Annotated[list, add]       # Accumulated workflow steps
    documents: list                   # Retrieved documents
    reasoning: str                    # LLM reasoning
    final_answer: str                 # Final output
```

Why TypedDict?

- Type safety at runtime
- IDE autocompletion
- Clear contract for state shape
- Easy debugging

2. Annotated Reducer Pattern

python

```
steps: Annotated[list, add]
```

This is advanced Python typing:

- `Annotated` marks metadata
- `add` operator concatenates lists
- Each node can append steps without overwriting

Without this pattern (❌ wrong):

```python
state["steps"] = ["step1"]  # Overwrites previous
```

With this pattern (✅ correct):

```python
state["steps"].append("step1")  # Accumulates
```

3. Nodes

A node is a function that:

1. Takes current state as input
2. Performs an action
3. Updates and returns state

```python
def my_node(state: AgentState) -> AgentState:
    # Do something
    state["field"] = "value"
    return state
```

4. Edges

Edges define the flow between nodes:

```python
```

```
workflow.add_edge("node_a", "node_b")        # Sequential
workflow.add_conditional_edges(              # Conditional
    "node_x",
    decision_function,
    {"path1": "node_y", "path2": "node_z"}
)
```

5. Graph Compilation

```python
graph = workflow.compile()
```

This converts your graph definition into an executable object. It optimizes and validates the workflow.

💻 Code Breakdown

Section 1: Imports & Dependencies

```python
from langgraph.graph import StateGraph, START, END
from typing import TypedDict, Annotated
from openai import OpenAI
from operator import add
```

Import	Purpose
`StateGraph`	Create workflow graph
`START, END`	Special nodes for entry/exit
`TypedDict`	Type-safe state definition
`Annotated`	Add metadata to types
`OpenAI`	Call OpenAI models
`add`	Concatenate lists in state

Section 2: State Definition

```python
class AgentState(TypedDict):
    input: str                    # Raw user input
    steps: Annotated[list, add]       # Workflow progress tracker
    documents: list               # Retrieved context
    reasoning: str                # Intermediate thinking
    final_answer: str             # Final output
```

Each field explained:

Field	Type	Purpose	Initial Value
`input`	str	User's question	Passed in
`steps`	Annotated list	Track executed steps	Empty list
`documents`	list	Retrieved knowledge	Empty list
`reasoning`	str	LLM thinking process	Empty string
`final_answer`	str	Answer to user	Empty string

Section 3: Initialize OpenAI Client

```python
client = OpenAI()  # Reads OPENAI_API_KEY from environment
```

How it works:

- Automatically reads `OPENAI_API_KEY` environment variable
- Creates authenticated client
- Ready to make API calls

Setting the API key:

```bash
```

```
export OPENAI_API_KEY="sk-proj-..."  # Linux/Mac
set OPENAI_API_KEY=sk-proj-...       # Windows
```

Section 4: Knowledge Base

```python
KNOWLEDGE_BASE = {
    "python": ["Python is...", "It uses..."],
    "api": ["API stands for...", "REST APIs..."],
    "database": ["Databases store...", "SQL is..."]
}
```

In production:

- Replace with vector database (Pinecone, Weaviate)
- Connect to SQL/NoSQL database
- Call external APIs
- Load from files

Section 5: Node Functions

Node 1: Retrieve Documents

```python
def retrieve_documents(state: AgentState) -> AgentState:
    """Step 1: Retrieve relevant documents"""

    query = state["input"].lower()
    docs = []

    # Keyword matching (in production: use embeddings + similarity search)
    for topic, content in KNOWLEDGE_BASE.items():
        if topic in query:
            docs.extend(content)

    state["steps"].append("✓ Retrieved documents")
    state["documents"] = docs

    return state
```

What it does:

1. Extracts user query
2. Searches knowledge base
3. Updates steps tracker
4. Stores documents in state
5. Returns updated state

Real-world improvement:

```python
# Use embeddings for better matching
from openai import OpenAI
client = OpenAI()

query_embedding = client.embeddings.create(
    model="text-embedding-3-small",
    input=state["input"]
)
# Compare with document embeddings
```

Node 2: Generate Reasoning

```python
```

```python
def generate_reasoning(state: AgentState) -> AgentState:
    """Step 2: Generate reasoning using OpenAI"""

    documents_text = "\n".join(state["documents"])

    response = client.messages.create(
        model="gpt-4o-mini",
        max_tokens=200,
        messages=[{
            "role": "user",
            "content": f"""Explain how these docs relate to: {state['input']}

Documents:
{documents_text}"""
        }]
    )

    state["reasoning"] = response.content[0].text
    state["steps"].append("✓ Generated reasoning")

    return state
```

Key concepts:

- `messages.create()` - Call OpenAI API
- `model="gpt-4o-mini"` - Fast, cheap model
- `max_tokens=200` - Limit response length
- Prompt engineering included
- Chain-of-thought reasoning

Node 3: Generate Answer

```python
```

```python
def generate_answer(state: AgentState) -> AgentState:
    """Step 3: Generate final answer"""

    response = client.messages.create(
        model="gpt-4o-mini",
        max_tokens=300,
        messages=[{
            "role": "user",
            "content": f"""Answer this question:

Question: {state['input']}
Knowledge: {"\n".join(state["documents"])}
Reasoning: {state["reasoning"]}

Provide a clear answer."""
        }]
    )

    state["final_answer"] = response.content[0].text
    state["steps"].append("✓ Generated final answer")

    return state
```

Why multiple steps?

- Reasoning helps LLM think through problem
- Better answers than direct approach
- Follows chain-of-thought pattern
- More transparent process

Node 4: Validate Answer

```python
```

```
def validate_answer(state: AgentState) -> AgentState:
    """Step 4: Validate answer quality"""

    response = client.messages.create(
        model="gpt-4o-mini",
        max_tokens=100,
        messages=[{
            "role": "user",
            "content": f"""Is this answer helpful? YES or NO:

Question: {state['input']}
Answer: {state['final_answer']}"""
        }]
    )

    validation = response.content[0].text.strip()
    state["steps"].append(f"✓ Validated answer ({validation})")

    return state
```

Real-world use:

- Filter poor answers
- Trigger re-generation if NO
- Add confidence scoring
- Log for quality monitoring

Section 6: Build the Graph

```
python
```

```
workflow = StateGraph(AgentState)

# Add all nodes
workflow.add_node("retrieve", retrieve_documents)
workflow.add_node("reasoning", generate_reasoning)
workflow.add_node("answer", generate_answer)
workflow.add_node("validate", validate_answer)

# Connect nodes sequentially
workflow.add_edge(START, "retrieve")        # Entry point
workflow.add_edge("retrieve", "reasoning")  # Step 1 → 2
workflow.add_edge("reasoning", "answer")    # Step 2 → 3
workflow.add_edge("answer", "validate")     # Step 3 → 4
workflow.add_edge("validate", END)          # Exit point

# Compile to executable
graph = workflow.compile()
```

Graph visualization:

```
START → retrieve → reasoning → answer → validate → END
```

🔄 Execution Flow

Step-by-step execution:

1. User Input

```python
query = "What is Python and why is it useful?"
```

2. Initialize State

```python
```

```
initial_state = AgentState(
    input="What is Python and why is it useful?",
    steps=[],
    documents=[],
    reasoning="",
    final_answer=""
)
```

3. Invoke Graph

```python
final_state = graph.invoke(initial_state)
```

4. Node 1: Retrieve

```
- Search KB for "python"
- Find 3 documents
- state["documents"] = ["Python is...", "It uses...", ...]
- state["steps"] = ["✓ Retrieved documents"]
```

5. Node 2: Reasoning

```
- Call OpenAI with documents + question
- Generate: "Python is relevant because..."
- state["reasoning"] = "Python is relevant..."
- state["steps"] = [..., "✓ Generated reasoning"]
```

6. Node 3: Answer

```
- Call OpenAI with all context
- Generate full answer
- state["final_answer"] = "Python is a high-level language..."
- state["steps"] = [..., "✓ Generated final answer"]
```

7. Node 4: Validate

```
- Ask OpenAI: "Is this good?"
- Get: "YES"
- state["steps"] = [..., "✓ Validated answer (YES)"]
```

8. Return Final State

```python
{
  "input": "What is Python...",
  "steps": ["✓ Retrieved documents", "✓ Generated reasoning", ...],
  "documents": ["Python is...", ...],
  "reasoning": "Python is relevant...",
  "final_answer": "Python is a high-level language..."
}
```

🚀 Installation & Setup

Step 1: Create Virtual Environment

```bash
# Create
python -m venv venv

# Activate
source venv/bin/activate      # Linux/Mac
venv\Scripts\activate         # Windows
```

Step 2: Install Dependencies

```bash
pip install langgraph openai python-dotenv
```

What each package does:

Package	Purpose
`langgraph`	Workflow orchestration
`openai`	OpenAI API client
`python-dotenv`	Load environment variables

Step 3: Set API Key

Option A: Environment Variable (Recommended)

```bash
export OPENAI_API_KEY="sk-proj-your-key-here"
```

Option B: .env File

Create `.env`:

```
OPENAI_API_KEY=sk-proj-your-key-here
```

Load in code:

```python
from dotenv import load_dotenv
import os

load_dotenv()
api_key = os.getenv("OPENAI_API_KEY")
client = OpenAI(api_key=api_key)
```

Step 4: Verify Installation

```bash
python -c "import langgraph; import openai; print('✓ All packages installed')"
```

📝 Usage Examples

Example 1: Basic Usage

```python
from agent import run_agent

result = run_agent("What is Python?")
print(result["final_answer"])
```

Output:

```
Python is a high-level, interpreted programming language known for its
simplicity and readability. It uses indentation for code blocks and has
powerful frameworks like Django and Flask for web development.
```

Example 2: Multiple Queries

```python
queries = [
    "What is Python and why is it useful?",
    "Explain REST APIs to me",
    "Tell me about databases"
]

for query in queries:
    print(f"\n❓ Query: {query}")
    result = run_agent(query)
    print(f"✅ Answer: {result['final_answer'][:100]}...")
```

Example 3: Custom Knowledge Base

```python
KNOWLEDGE_BASE = {
    "machine-learning": [
        "ML is subset of AI",
        "Common algorithms: regression, classification",
        "Libraries: TensorFlow, PyTorch, scikit-learn"
    ],
    "deep-learning": [
        "Deep learning uses neural networks",
        "Popular architectures: CNN, RNN, Transformers",
        "Used in NLP and computer vision"
    ]
}

result = run_agent("How does deep learning work?")
```

Example 4: Extract Results

```python
```

```
final_state = graph.invoke(initial_state)

# Get specific outputs
answer = final_state["final_answer"]
reasoning = final_state["reasoning"]
steps = final_state["steps"]
docs = final_state["documents"]

print("=== WORKFLOW EXECUTION ===")
for step in steps:
    print(step)

print("\n=== DOCUMENTS USED ===")
for doc in docs:
    print(f"• {doc}")

print("\n=== REASONING ===")
print(reasoning)

print("\n=== FINAL ANSWER ===")
print(answer)
```

🔧 Advanced Customization

1. Add Conditional Routing

```python
```

```python
def should_revalidate(state: AgentState) -> str:
    """Decide if we need to regenerate answer"""
    if "NO" in state["steps"][-1]:
        return "regenerate"
    return "end"

workflow.add_conditional_edges(
    "validate",
    should_revalidate,
    {
        "regenerate": "answer",
        "end": END
    }
)
```

2. Add Parallel Execution

```python
from langgraph.graph import StateGraph

# Create branches that execute in parallel
workflow.add_node("branch_a", func_a)
workflow.add_node("branch_b", func_b)

workflow.add_edge("start", "branch_a")
workflow.add_edge("start", "branch_b")

# Both branches merge
workflow.add_edge(["branch_a", "branch_b"], "merge_node")
```

3. Add Error Handling

```python
```

```python
def retrieve_documents_safe(state: AgentState) -> AgentState:
    """Retrieve with error handling"""
    try:
        query = state["input"].lower()
        docs = []
        for topic, content in KNOWLEDGE_BASE.items():
            if topic in query:
                docs.extend(content)
        state["documents"] = docs
    except Exception as e:
        state["documents"] = [f"Error: {str(e)}"]

    state["steps"].append("✓ Retrieved documents")
    return state
```

4. Add Logging

python

```python
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def retrieve_documents(state: AgentState) -> AgentState:
    """Retrieve with logging"""
    logger.info(f"Retrieving for query: {state['input']}")

    # ... retrieval logic ...

    logger.info(f"Found {len(state['documents'])} documents")
    return state
```

5. Custom Model Selection

python

```python
def generate_reasoning(state: AgentState) -> AgentState:
    """Use different models for different cases"""

    if len(state["input"]) < 50:
        model = "gpt-4o-mini"  # Simple queries
    else:
        model = "gpt-4"        # Complex queries

    response = client.messages.create(
        model=model,
        max_tokens=200,
        messages=[...]
    )

    return state
```

6. Add Metrics & Monitoring

python

```python
import time

def retrieve_documents(state: AgentState) -> AgentState:
    """Retrieve with metrics"""
    start = time.time()

    # ... retrieval logic ...

    duration = time.time() - start
    state["steps"].append(f"✓ Retrieved documents ({duration:.2f}s)")

    return state
```

🐛 Troubleshooting

Problem 1: API Key Not Found

Error:

```
AuthenticationError: Error code: 401
```

Solution:

```bash
# Verify key is set
echo $OPENAI_API_KEY

# If empty, set it
export OPENAI_API_KEY="sk-proj-..."

# Or create .env file
echo "OPENAI_API_KEY=sk-proj-..." > .env
```

Problem 2: Rate Limiting

Error:

```
RateLimitError: Error code: 429
```

Solution:

```python
import time

def retry_with_backoff(func, max_retries=3):
    for attempt in range(max_retries):
        try:
            return func()
        except Exception as e:
            if attempt < max_retries - 1:
                wait_time = 2 ** attempt
                print(f"Rate limited. Waiting {wait_time}s...")
                time.sleep(wait_time)
            else:
                raise
```

Problem 3: State Not Updating

Wrong:

```python
```

```python
state["documents"] = []  # This might not persist
```

Correct:

```python
state["documents"] = retrieved_docs  # Ensure full assignment
return state  # Always return state
```

Problem 4: Nodes Not Executing

Check:

```python
# Verify edges are connected
print(graph.get_graph().nodes)
print(graph.get_graph().edges)

# Visualize
graph.get_graph().draw_mermaid()
```

Problem 5: High API Costs

Solution:

```python
# Use cheaper model
model = "gpt-4o-mini"  # 5x cheaper than gpt-4

# Reduce tokens
max_tokens = 100  # Instead of 1000

# Cache responses
from functools import lru_cache

@lru_cache(maxsize=128)
def get_answer(query: str):
    # Only call API once per unique query
    pass
```

📊 Performance Optimization

1. Batch Processing

python

```
def run_batch(queries: list) -> list:
    """Process multiple queries efficiently"""
    results = []
    for query in queries:
        initial_state = AgentState(
            input=query,
            steps=[],
            documents=[],
            reasoning="",
            final_answer=""
        )
        result = graph.invoke(initial_state)
        results.append(result)
    return results
```

2. Caching

python

```
from functools import lru_cache

@lru_cache(maxsize=256)
def cached_retrieve(query: str) -> tuple:
    """Cache document retrieval"""
    docs = []
    for topic, content in KNOWLEDGE_BASE.items():
        if topic in query.lower():
            docs.extend(content)
    return tuple(docs)
```

3. Async Execution

python

```python
import asyncio

async def run_agent_async(query: str):
    """Non-blocking execution"""
    loop = asyncio.get_event_loop()
    result = await loop.run_in_executor(None, run_agent, query)
    return result

# Usage
results = asyncio.gather(
    run_agent_async("Query 1"),
    run_agent_async("Query 2"),
    run_agent_async("Query 3")
)
```

🎓 Learning Path

1. **Understand State** - How data flows through nodes
2. **Build Simple Graph** - 2-3 nodes sequentially
3. **Add Conditional Logic** - Route based on decisions
4. **Integrate Tools** - Call external APIs/databases
5. **Add Error Handling** - Handle failures gracefully
6. **Optimize Performance** - Cache, batch, async
7. **Deploy to Production** - Docker, serverless, cloud

📚 Additional Resources

- **LangGraph Docs**: https://langchain-ai.github.io/langgraph/
- **OpenAI API Docs**: https://platform.openai.com/docs
- **Chain-of-Thought Papers**: https://arxiv.org/abs/2201.11903
- **Agentic AI Patterns**: https://github.com/langchain-ai/langgraph

---

✅ Checklist for Production

- [ ] API keys securely stored (use .env)
- [ ] Error handling for all nodes
- [ ] Logging at each step
- [ ] Rate limiting implemented
- [ ] Cost monitoring enabled
- [ ] Tests for each node function
- [ ] Documentation for team
- [ ] Performance benchmarks
- [ ] Security audit completed
- [ ] Monitoring/alerting setup

Happy building! 🚀