LangChain and LangGraph Notes

Table of Contents

LangChain Expression Language (LCEL) Fundamentals

What is LCEL?

LangChain Expression Language (LCEL) is a declarative way to compose chains of components in LangChain. It uses the **Runnable Protocol**, which defines standardized methods that all components must implement.

Key Components:

- **Prompts**: Templates for generating inputs
- **Models**: LLMs that process the prompts
- **Output Parsers**: Format and structure the model outputs
- **Tools**: External functions the model can call

The Runnable Protocol

Every component in LangChain implements the Runnable interface with these standard methods:

```
# Synchronous methods
result = runnable.invoke(input)           # Single input
results = runnable.batch([input1, input2])  # Multiple inputs
for chunk in runnable.stream(input):      # Streaming output
    print(chunk)

# Asynchronous methods
result = await runnable.ainvoke(input)
results = await runnable.abatch([input1, input2])
async for chunk in runnable.astream(input):
    print(chunk)
```

The Pipe Operator (|) and Operator Overloading

The | symbol in LangChain uses **operator overloading** to create intuitive data pipelines:

```
# Native Python bitwise OR
print(5 | 3)  # Output: 7 (bitwise OR)
```

```
# LangChain overloaded behavior
chain = prompt | model | parser  # Creates a RunnableSequence
```

How it works:

1. Python classes can override the `__or__` method to customize the `|` operator
2. LangChain's `Runnable` class implements `__or__` to create `RunnableSequence` objects
3. This makes `prompt | model` equivalent to `RunnableSequence(prompt, model)`

Basic Chain Example

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

# Setup components
prompt = ChatPromptTemplate.from_template("Tell me a joke about {topic}")
llm = AzureChatOpenAI(...)
parser = StrOutputParser()

# Create chain with pipe syntax
chain = prompt | llm | parser

# Execute
result = chain.invoke({"topic": "cats"})
```

Advanced LCEL Features

1. Parallel Execution

```
from langchain_core.runnables import RunnableParallel

parallel_chain = RunnableParallel(
    joke=joke_chain,
    fact=fact_chain
)
result = parallel_chain.invoke({"topic": "space"})
# Returns: {"joke": "...", "fact": "..."}
```

2. Fallbacks

```
# Add backup models
chain_with_fallback = primary_llm.with_fallbacks([backup_llm])
```

3. Conditional Logic

```
def route_by_topic(input_data):
    if "weather" in input_data["topic"]:
        return weather_chain
    return general_chain

conditional_chain = RunnableLambda(route_by_topic)
```

RAG (Retrieval-Augmented Generation) Pattern

```
from langchain_community.vectorstores import DocArrayInMemorySearch
from langchain_huggingface import HuggingFaceEmbeddings

# Setup vector store
embeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2")
vectorstore = DocArrayInMemorySearch.from_texts(documents, embedding=embeddings)
retriever = vectorstore.as_retriever()

# RAG chain
rag_chain = RunnableMap({
    "context": lambda x: retriever.get_relevant_documents(x["question"]),
    "question": lambda x: x["question"]
}) | prompt | llm | parser
```

LangGraph Concepts: Threads, State, and Persistence

What is LangGraph?

LangGraph is a framework for building **stateful, multi-actor applications** with LLMs. It extends LangChain with:

- **State management** across conversation turns
- **Multi-step workflows** with decision points
- **Human-in-the-loop** capabilities
- **Persistence** and resumption

Core Concepts Explained with Analogies

1. THREAD = Individual Conversations

Think of your phone's messaging app:

- Each contact has their own conversation thread
- Messages are isolated between different people

- You can switch between conversations without mixing them up

```
# Different users, different threads
config_sarah = {"configurable": {"thread_id": "user_sarah"}}
config_john = {"configurable": {"thread_id": "user_john"}}
```

2. STATE = Conversation Memory

Like your chat history:

- Remembers what was said before
- Maintains context across messages
- Can store additional metadata

```
class AgentState(TypedDict):
    messages: Annotated[list, add_messages]  # Conversation history
    user_id: str                             # Custom field
    budget: float                            # Custom field
    preferences: dict                        # Custom field
```

3. PERSISTENCE = Saving Conversations

Like your phone saving messages even after closing the app:

- **MemorySaver**: Keeps state in RAM (lost when program stops)
- **SqliteSaver**: Saves to database (survives restarts)

```
from langgraph.checkpoint.memory import MemorySaver

memory = MemorySaver()
agent = workflow.compile(checkpointer=memory)
```

4. STREAMING = Real-time Updates

Like seeing "..." when someone is typing:

- Watch the agent's thinking process step-by-step
- See tool calls as they happen
- Get responses as they're generated

```
for event in agent.stream(input_data, config):
    print(f"Step: {event}")
```

Building a Basic LangGraph Agent

Step 1: Define Tools

```
from langchain_core.tools import tool

@tool
def get_weather(city: str) -> str:
    """Get weather information for a city."""
    return f"Weather in {city}: Sunny, 75°F"

@tool
def get_travel_budget(destination: str, days: int) -> str:
    """Calculate travel budget estimate."""
    cost_per_day = {"paris": 200, "tokyo": 180, "bali":
80}.get(destination.lower(), 150)
    total = cost_per_day * days
    return f"Estimated budget for {destination}: ${total} (${cost_per_day}/day)"
```

Step 2: Create the Graph

```
from langgraph.graph import StateGraph, MessagesState, START, END
from langgraph.prebuilt import ToolNode

def should_continue(state):
    """Decide whether to use tools or end."""
    last_message = state["messages"][-1]
    if hasattr(last_message, "tool_calls") and last_message.tool_calls:
        return "tools"
    return END

def call_model(state):
    """Call the AI model."""
    response = llm.bind_tools(tools).invoke(state["messages"])
    return {"messages": [response]}

# Build the graph
workflow = StateGraph(MessagesState)
workflow.add_node("agent", call_model)
workflow.add_node("tools", ToolNode(tools))

workflow.add_edge(START, "agent")
workflow.add_conditional_edges("agent", should_continue, ["tools", END])
workflow.add_edge("tools", "agent")

agent = workflow.compile()
```

Advanced LangGraph Features

Human-in-the-Loop (HITL)

Pause execution for human approval before taking actions:

```
# Create agent that pauses before using tools
agent_with_approval = workflow.compile(
    checkpointer=memory,
    interrupt_before=["tools"]  # Pause here for approval
)

# Execute with interruption
response = agent_with_approval.invoke(input_data, config)
current_state = agent_with_approval.get_state(config)

if current_state.next:
    # Show what agent wants to do
    last_message = current_state.values['messages'][-1]
    print(f"Agent wants to use: {last_message.tool_calls}")

    # Get human approval
    approval = input("Approve? (y/n): ")
    if approval.lower() == 'y':
        # Continue execution
        agent_with_approval.stream(None, config)
```

State Modification

Change what the agent is about to do:

```
# Modify tool call arguments
modified_message = AIMessage(
    content="",
    tool_calls=[{
        'name': 'get_weather',
        'args': {'city': 'Paris'},  # Changed from original request
        'id': 'modified_call'
    }]
)

agent.update_state(config, {"messages": [modified_message]})
```

Time Travel with Checkpoints

Access any previous point in the conversation:

```
# Get conversation history
history = list(agent.get_state_history(config))
```

```
# Go back to an earlier checkpoint
old_checkpoint = history[2]  # Third most recent
response = agent.invoke(new_input, config=old_checkpoint.config)
```

Comparison: Threads vs User Management Systems

Feature	LangGraph Threads	Traditional UMS	Hybrid Approach
Setup	Simple: Just thread_id	Complex: DB + Auth	Medium complexity
Isolation	Built-in per thread	Manual implementation	Best of both
Persistence	Via checkpointers	Custom database	Combined storage
Scalability	High (10k+ threads)	DB-dependent	Optimized
Security	Basic isolation	Full auth system	Enhanced security
Use Case	Agent workflows	User accounts	Enterprise apps

Advanced LangGraph Features

Workaround for Limited Models (e.g., Amazon Bedrock Nova Lite)

Some models don't support native tool calling. Use **prompt engineering** as a workaround:

```
def parse_tool_call(response_text):
    """Parse tool calls from model text output."""
    try:
        # Try JSON parsing first
        parsed = json.loads(response_text)
        if "tool" in parsed:
            return {
                "name": parsed["tool"],
                "args": parsed["args"],
                "id": "parsed_call"
            }
    except json.JSONDecodeError:
        # Fallback to regex parsing
        import re
        match = re.search(r"Action: (\w+) \[(.+)\]", response_text)
        if match:
            return {
                "name": match.group(1),
                "args": {"query": match.group(2)},
                "id": "regex_call"
            }
    return None

def call_model_with_parsing(state):
    """Enhanced model calling with tool parsing."""
```

```
    system_prompt = """
    You are an agent that uses tools. For tool calls, respond in JSON:
    {"tool": "tool_name", "args": {"key": "value"}}
    Otherwise, provide a direct answer.
    """

    messages = [SystemMessage(content=system_prompt)] + state["messages"]
    response = llm.invoke(messages)

    # Parse for tool calls
    tool_call = parse_tool_call(response.content)
    if tool_call:
        response.tool_calls = [tool_call]

    return {"messages": [response]}
```

Custom State Beyond Messages

Extend state to track additional information:

```
class CustomAgentState(TypedDict):
    messages: Annotated[list, add_messages]
    user_preferences: dict
    conversation_summary: str
    last_tool_used: Optional[str]
    session_start: datetime
    interaction_count: int

def update_interaction_count(state):
    """Track number of interactions."""
    return {"interaction_count": state.get("interaction_count", 0) + 1}

# Use in workflow
workflow.add_node("counter", update_interaction_count)
```

Error Handling and Retries

```
def safe_tool_call(state):
    """Tool calling with error handling."""
    try:
        return call_tools(state)
    except Exception as e:
        error_message = HumanMessage(content=f"Tool error: {str(e)}")
        return {"messages": [error_message]}

def should_retry(state):
    """Check if we should retry after an error."""
    last_message = state["messages"][-1]
    if "Tool error" in last_message.content:
```

```
        retry_count = state.get("retry_count", 0)
        if retry_count < 3:
            return "retry"
    return END

# Add retry logic to workflow
workflow.add_node("safe_tools", safe_tool_call)
workflow.add_conditional_edges("safe_tools", should_retry, ["retry", END])
```

Practical Implementation Patterns

Pattern 1: Multi-Step Travel Planning Agent

```
class TravelPlannerState(TypedDict):
    messages: Annotated[list, add_messages]
    destination: Optional[str]
    budget: Optional[float]
    dates: Optional[dict]
    preferences: dict
    plan_status: str

def extract_travel_info(state):
    """Extract travel details from conversation."""
    # Implementation to parse destination, budget, dates
    pass

def search_flights(state):
    """Search for flights based on extracted info."""
    pass

def find_hotels(state):
    """Find accommodations."""
    pass

def create_itinerary(state):
    """Generate day-by-day itinerary."""
    pass

# Build comprehensive travel workflow
travel_workflow = StateGraph(TravelPlannerState)
travel_workflow.add_node("extract_info", extract_travel_info)
travel_workflow.add_node("search_flights", search_flights)
travel_workflow.add_node("find_hotels", find_hotels)
travel_workflow.add_node("create_itinerary", create_itinerary)

# Add conditional logic for workflow steps
```

Pattern 2: Customer Support with Escalation

```
class SupportState(TypedDict):
    messages: Annotated[list, add_messages]
    issue_category: str
    severity_level: int
    resolution_attempts: int
    escalated: bool
    customer_info: dict

def classify_issue(state):
    """Classify customer issue and set severity."""
    pass

def attempt_resolution(state):
    """Try to resolve the issue."""
    pass

def should_escalate(state):
    """Decide if issue should be escalated to human agent."""
    if state["severity_level"] > 7 or state["resolution_attempts"] > 2:
        return "escalate"
    return "retry"

def escalate_to_human(state):
    """Transfer to human agent."""
    return {"escalated": True, "messages": [
        HumanMessage(content="Escalating to human agent...")
    ]}
```

Pattern 3: Content Generation Pipeline

```
def generate_outline(state):
    """Create content outline."""
    pass

def research_topic(state):
    """Gather information on the topic."""
    pass

def write_draft(state):
    """Generate initial content draft."""
    pass

def review_and_edit(state):
    """Review and improve content."""
    pass

def format_output(state):
    """Format for final presentation."""
    pass
```

```
# Chain content creation steps with human review points
content_workflow = StateGraph(ContentState)
# ... add nodes and edges with review checkpoints
```

Best Practices

1. State Design

- Keep state minimal but comprehensive
- Use typing for better error detection
- Include metadata for debugging

2. Error Handling

- Always handle tool failures gracefully
- Implement retry logic for transient errors
- Log state transitions for debugging

3. Performance Optimization

- Use batch operations when possible
- Cache expensive computations in state
- Implement proper cleanup for long-running threads

4. Security Considerations

- Validate all inputs before tool execution
- Implement approval workflows for sensitive operations
- Use proper authentication for multi-user systems

5. Testing Strategies

- Test individual nodes in isolation
- Use mock tools for unit testing
- Implement integration tests with real workflows

Common Pitfalls and Solutions

Pitfall 1: State Bloat

Problem: State becomes too large and slows down processing. **Solution**: Regularly clean up old data, use state compression.

Pitfall 2: Infinite Loops

Problem: Agent gets stuck in reasoning loops. **Solution**: Implement step counters and loop detection.

Pitfall 3: Tool Call Failures

Problem: External APIs fail and break workflow. **Solution**: Implement robust error handling and fallbacks.

Conclusion

LangChain and LangGraph provide powerful frameworks for building sophisticated AI applications:

- **LangChain (LCEL)** excels at composing simple, stateless chains of operations
- **LangGraph** adds state management and complex workflow capabilities
- Both frameworks emphasize modularity, reusability, and developer experience

The key to success is understanding when to use each tool:

- Use **LCEL** for straightforward prompt → model → output chains
- Use **LangGraph** when you need memory, multi-step reasoning, or human interaction

These frameworks continue to evolve rapidly, with new features and improvements being added regularly. Stay updated with the official documentation and community resources for the latest capabilities.