Reference Card: Docling
Yogesh Haribhau Kulkarni

LangGraph: Installation & Core Concepts

```
# Installation
pip install langgraph langchain-core langchain-openai

# Core Imports
from langgraph.graph import StateGraph, END, START
from typing import TypedDict, Annotated
from langchain_core.messages import HumanMessage, AIMessage
import operator

# State Definition - The data structure passed between nodes
class GraphState(TypedDict):
    messages: Annotated[list, operator.add]  # operator.add appends to list
    count: int
    data: dict

# Alternative: Using channels for state management
from langgraph.graph import MessagesState
class MyState(MessagesState):
    custom_field: str  # Inherits 'messages' field automatically
```

Building Basic Graphs

```
# Create StateGraph
workflow = StateGraph(GraphState)

# Define Node Functions - must take state and return dict with updates
def node_1(state: GraphState):
    return {"count": state["count"] + 1, "messages": [AIMessage(content="Step 1")]}

def node_2(state: GraphState):
    return {"count": state["count"] * 2}

# Add Nodes
workflow.add_node("process", node_1)
workflow.add_node("transform", node_2)

# Add Edges - Define flow between nodes
workflow.add_edge(START, "process")  // START is entry point
workflow.add_edge("process", "transform")
workflow.add_edge("transform", END)  // END terminates graph

# Compile Graph
app = workflow.compile()

# Invoke Graph
result = app.invoke({"count": 1, "messages": []})
print(result)  // {"count": 4, "messages": [...]}
```

Conditional Edges & Routing

```
# Conditional Routing Function - returns next node name
def route_decision(state: GraphState) -> str:
    if state["count"] > 10:
        return "high_path"
    return "low_path"

# Add Conditional Edge
workflow.add_conditional_edges(
    "process",  // Source node
    route_decision,  // Function that returns next node
    {
        "high_path": "handler_high",  // Mapping of return values to nodes
        "low_path": "handler_low"
    }
)

# Multiple Output Routing
def multi_route(state: GraphState) -> list[str]:
    return ["path_a", "path_b"]  // Execute both paths in parallel

workflow.add_conditional_edges("start", multi_route)

# Direct to END
def maybe_end(state: GraphState) -> str:
    return END if state["count"] > 100 else "continue"
```

Checkpointing & Memory

```
# In-Memory Checkpointer
from langgraph.checkpoint.memory import MemorySaver
memory = MemorySaver()
app = workflow.compile(checkpointer=memory)

# SQLite Checkpointer (Persistent)
from langgraph.checkpoint.sqlite import SqliteSaver
with SqliteSaver.from_conn_string("checkpoints.db") as checkpointer:
    app = workflow.compile(checkpointer=checkpointer)

# Using with Thread ID for conversation memory
config = {"configurable": {"thread_id": "user_123"}}
result = app.invoke({"messages": [HumanMessage(content="Hello")]}, config)

# Stream with checkpointing
for chunk in app.stream(input_data, config):
    print(chunk)

# Get State at checkpoint
state = app.get_state(config)
print(state.values)  // Current state values
print(state.next)  // Next nodes to execute
```

Human-in-the-Loop & Interrupts

```
# Compile with interrupt before specific nodes
app = workflow.compile(
    checkpointer=memory,
    interrupt_before=["human_review"]  // Pause before this node
)

# Invoke - will stop at interrupt
config = {"configurable": {"thread_id": "conv_1"}}
result = app.invoke(input_data, config)

# Check if interrupted
state = app.get_state(config)
print(state.next)  // Shows ["human_review"] if interrupted

# Resume after human review - update state if needed
app.update_state(config, {"approved": True})
result = app.invoke(None, config)  // Continue from checkpoint

# Interrupt after nodes
app = workflow.compile(interrupt_after=["needs_approval"])

# Manual breakpoint in node
from langgraph.types import interrupt
def node_with_breakpoint(state):
    decision = interrupt("Review this decision")  // Pause and wait
    return {"result": decision}
```

Subgraphs & Composition

```
# Define Subgraph
subgraph_builder = StateGraph(GraphState)
subgraph_builder.add_node("sub_node_1", func1)
subgraph_builder.add_node("sub_node_2", func2)
subgraph_builder.add_edge(START, "sub_node_1")
subgraph_builder.add_edge("sub_node_1", "sub_node_2")
subgraph_builder.add_edge("sub_node_2", END)
subgraph = subgraph_builder.compile()

# Add Subgraph as Node
workflow = StateGraph(GraphState)
workflow.add_node("preprocessing", preprocess_func)
workflow.add_node("subgraph_step", subgraph)  // Use compiled graph as node
workflow.add_node("postprocessing", postprocess_func)
workflow.add_edge(START, "preprocessing")
workflow.add_edge("preprocessing", "subgraph_step")
workflow.add_edge("subgraph_step", "postprocessing")
workflow.add_edge("postprocessing", END)
app = workflow.compile()
```

Tools & Function Calling

```
# Define Tools
from langchain_core.tools import tool

@tool
def search_tool(query: str) -> str:
    """Search for information"""
    return f"Results for: {query}"

tools = [search_tool]

# Create Tool Node
from langgraph.prebuilt import ToolNode
tool_node = ToolNode(tools)

# Agent with Tools
from langchain_openai import ChatOpenAI
model = ChatOpenAI(model="gpt-4").bind_tools(tools)

def call_model(state: MessagesState):
    response = model.invoke(state["messages"])
    return {"messages": [response]}

# Routing: agent -> tools or end
def should_continue(state: MessagesState) -> str:
    last_message = state["messages"][-1]
    if last_message.tool_calls:
        return "tools"
    return END

workflow = StateGraph(MessagesState)
workflow.add_node("agent", call_model)
workflow.add_node("tools", tool_node)
workflow.add_edge(START, "agent")
workflow.add_conditional_edges("agent", should_continue, {"tools": "tools", END: END})
workflow.add_edge("tools", "agent")
```

Streaming & Async Operations

```
# Stream Mode - "values" (full state), "updates" (changes), "messages"
for chunk in app.stream(input_data, stream_mode="values"):
    print(chunk)

for chunk in app.stream(input_data, stream_mode="updates"):
    print(chunk)  // Only shows updates per node

# Stream Events - granular control
for event in app.stream(input_data, stream_mode="events"):
    print(event["event"], event["data"])

# Async Execution
async def async_node(state):
    return {"result": "async_result"}

# Async invocation
import asyncio
result = await app.ainvoke(input_data, config)

# Async streaming
async for chunk in app.astream(input_data, config):
    print(chunk)

# Stream tokens from LLM in agent
async for event in app.astream_events(input_data, version="v1"):
    if event["event"] == "on_chat_model_stream":
        print(event["data"]["chunk"].content, end="")
```

Map-Reduce & Parallelization

```
# Parallel Execution - Send pattern
from langgraph.graph import Send

def map_node(state: GraphState):
    items = state["items"]
    # Send each item to worker node in parallel
    return [Send("worker", {"item": item}) for item in items]

def worker(state: dict):
    result = process_item(state["item"])
    return {"results": [result]}

def reduce_node(state: GraphState):
    all_results = state.get("results", [])
    return {"final": aggregate(all_results)}

workflow = StateGraph(GraphState)
workflow.add_node("map", map_node)
workflow.add_node("worker", worker)
workflow.add_node("reduce", reduce_node)
workflow.add_edge(START, "map")
workflow.add_conditional_edges("map", lambda s: [Send("worker", s)])
workflow.add_edge("worker", "reduce")
workflow.add_edge("reduce", END)

# Parallel branches merging
workflow.add_edge("start", "branch_a")
workflow.add_edge("start", "branch_b")
workflow.add_edge(["branch_a", "branch_b"], "merge")  // Wait for both
```

State Reducers & Advanced Patterns

```
# Custom Reducers
def merge_dicts(left, right):
    return {**left, **right}

class AdvancedState(TypedDict):
    data: Annotated[dict, merge_dicts]  // Custom merge logic
    counter: Annotated[int, operator.add]
    items: Annotated[list, operator.add]

# Preprocessing/Postprocessing state updates
from langgraph.graph import add

class StateWithDefault(TypedDict):
    value: Annotated[int, add]  // Uses langgraph.graph.add reducer

# Time Travel - Update past state
state_history = app.get_state_history(config)
for state in state_history:
    print(state.values, state.config)

# Fork from past state
past_config = list(state_history)[2].config
app.update_state(past_config, {"new_field": "value"})
new_result = app.invoke(None, past_config)

# Trim messages for context management
def trim_messages(state):
    return {"messages": state["messages"][-10:]}  // Keep last 10
```

Error Handling & Best Practices

```python
# Error handling in nodes
def safe_node(state: GraphState):
    try:
        result = risky_operation(state)
        return {"status": "success", "result": result}
    except Exception as e:
        return {"status": "error", "error": str(e)}

# Retry logic
from tenacity import retry, stop_after_attempt, wait_exponential

@retry(stop=stop_after_attempt(3), wait=wait_exponential(multiplier=1))
def node_with_retry(state):
    return call_external_api(state)
```

```python
# Best Practices
# 1. Keep state minimal - only data needed across nodes
# 2. Use TypedDict for type safety
# 3. Return partial updates, not full state
# 4. Use checkpointers for production deployments
# 5. Implement proper error boundaries

# Visualization
from IPython.display import Image, display
display(Image(app.get_graph().draw_mermaid_png()))

# Export graph structure
print(app.get_graph().to_json())
```

Temporary page!

LaTeX was unable to guess the total number of pages correctly. As there was some unprocessed data that should have been added to the final page this extra page has been added to receive it.
If you rerun the document (without altering it) this surplus page will go away, because LaTeX now knows how many pages to expect for this document.