🚀 Complete Data Cleaning Agent Tutorial with LangGraph

Table of Contents

---

1. Understanding the Fundamentals {#fundamentals}

What is LangGraph?

LangGraph is a library for building stateful, multi-actor applications with LLMs. Think of it as a way to create a "workflow" where different AI agents can work together, remember what they've done, and make decisions based on previous steps.

Key Concepts:

- **Nodes**: Individual functions/agents that perform specific tasks
- **Edges**: Connections between nodes that determine the flow
- **State**: Shared memory that all nodes can read and modify
- **Conditional Edges**: Smart routing based on conditions

Why This Matters for Data Cleaning:

Data cleaning isn't a single step—it's a process:

1. **Analyze** → Understand the data
2. **Plan** → Decide what needs cleaning
3. **Execute** → Apply the cleaning
4. **Validate** → Check if it worked
5. **Iterate** → Repeat if needed

---

2. Why LangGraph for Data Cleaning? {#why-langgraph}

Traditional Approach Problems:

```python
# This is what you had - single LLM call
def clean_data(df):
    # One big prompt asking LLM to do everything
    # No memory, no validation, expensive
    return llm.generate_cleaning_code(df)
```

LangGraph Approach Benefits:

- **Memory**: Agents remember what they've tried
- **Validation**: Each step can be checked
- **Cost Efficiency**: Only call LLM when needed
- **Modularity**: Easy to add new cleaning strategies
- **Error Recovery**: Can retry failed steps

---

3. Building Our Agent Architecture {#architecture}

Our Agent Workflow:

```mermaid
graph TD
    A[Data Ingestion] --> B[Schema Analyzer]
    B --> C[Quality Assessor]
    C --> D[Cleaning Planner]
    D --> E[Code Generator]
    E --> F[Executor]
    F --> G[Validator]
    G --> H{Quality Check}
    H -->|Pass| I[Complete]
    H -->|Fail| D
    G --> J[Memory Store]
```

Agent Roles:

1. **Schema Analyzer**: Understands data structure (NO LLM needed)
2. **Quality Assessor**: Identifies issues (SMART LLM usage)
3. **Cleaning Planner**: Creates strategy (LLM + rules)

4. **Code Generator**: Writes Python code (LLM)
5. **Executor**: Runs code safely (NO LLM)
6. **Validator**: Checks results (NO LLM + optional LLM)

---

4. Implementation Guide {#implementation}

Step 1: Define Our State

```python
from typing import TypedDict, List, Dict, Any
from langgraph.graph import StateGraph

class DataCleaningState(TypedDict):
    # Data
    original_data: pd.DataFrame
    current_data: pd.DataFrame

    # Analysis
    schema_info: Dict[str, Any]
    quality_issues: List[Dict[str, Any]]

    # Planning
    cleaning_plan: List[Dict[str, Any]]
    current_step: int

    # Execution
    generated_code: str
    execution_result: Dict[str, Any]

    # Memory & Validation
    previous_attempts: List[Dict[str, Any]]
    validation_results: Dict[str, Any]
    is_complete: bool
```

Step 2: Create Smart Nodes

Each node has a specific purpose and uses LLM only when necessary.

Step 3: Memory System

Our agents will remember:

- What they've tried before
- What worked and what didn't
- Quality improvements over time

Step 4: Cost Optimization

- Cache repeated analyses
- Use smaller models for simple tasks
- Only call LLM when logic can't handle it

---

5. Cost Optimization Strategies {#cost-optimization}

1. Intelligent LLM Usage

```python
# Instead of always using LLM:
def should_use_llm(issue_type: str) -> bool:
    # Use rules for simple cases
    simple_cases = ['missing_values', 'duplicate_rows', 'whitespace']
    return issue_type not in simple_cases
```

2. Caching System

```python
# Cache expensive operations
@lru_cache(maxsize=100)
def analyze_column_pattern(column_data_hash: str):
    # Expensive LLM analysis only once per pattern
    pass
```

3. Progressive Enhancement

- Start with rule-based cleaning
- Use LLM only for complex cases
- Learn from user feedback

---

6. Advanced Features {#advanced-features}

Memory-Driven Learning

Our agent will remember:

- Which cleaning strategies work best for different data types
- Common patterns in your datasets
- User preferences and feedback

Adaptive Planning

- Start with conservative cleaning
- Progressively apply more aggressive techniques
- Always validate before proceeding

Interactive Validation

- Show before/after comparisons
- Ask for user confirmation on major changes
- Learn from user decisions

What We're Building Next

I'm going to create a complete implementation that demonstrates:

1. **Smart Architecture**: Each agent has a clear purpose
2. **Memory System**: Agents learn from experience
3. **Cost Efficiency**: Minimal LLM calls with maximum value
4. **User Experience**: Clear progress and validation
5. **Extensibility**: Easy to add new cleaning strategies

The result will be a data cleaning agent that's:

- 10x more cost-effective than your current approach
- More reliable through validation loops
- Smarter through memory and learning
- More transparent in its decision-making

Ready to dive into the code? Let's build something amazing! 🚀