00_langgraph_workflow_vs_agents

October 3, 2025

1 LangGraph: Workflows vs Agents - Complete Guide

1.1 Overview

LangGraph offers two fundamental approaches to building AI systems: **Workflows** and **Agents**. Understanding when to use each is critical for building efficient, maintainable AI applications.

1.1.1 The Fundamental Difference

Aspect	Workflows	Agents
Control Flow	Developer-defined	LLM-determined
Predictability	High - predetermined paths	Lower - dynamic decisions
Complexity	Simpler to debug	More complex reasoning
Use Case	Known process steps	Open-ended problems
LLM Role	Embedded in predefined steps	Orchestrates its own actions

---

1.2 Core Concepts

1.2.1 What is a Workflow?

A **workflow** is a graph where: - The control flow is **predetermined** by the developer - LLMs operate **within** the defined structure - Paths are **known at design time** - The system follows **explicit routing logic**

```
Input → LLM Call →  Gate →        LLM Call → Output
        (fixed)     (conditional) (fixed)
```

1.2.2 What is an Agent?

An **agent** is a system where: - The LLM **decides** the control flow - Tools are available but **usage is dynamic** - The system **adapts** to environmental feedback - Paths emerge from **agent decisions**

```
Input → Agent decides → Tool/Action → Feedback → Agent decides → ...
        (dynamic)       (variable)    (loop)     (dynamic)
```

1.3 Workflows Explained

1.3.1 Key Characteristics

1. **Predetermined Paths**: You define all possible routes
2. **Embedded Intelligence**: LLMs enhance specific steps
3. **Explicit Control**: Clear start, end, and decision points
4. **Verifiable Stages**: Each step can be tested independently

1.3.2 Workflow Patterns

1. Prompt Chaining Sequential LLM calls where each processes the previous output.

When to Use: - Translation with quality checks - Content generation with refinement - Multi-step verification processes

Example Structure:

```
def workflow():
    draft = llm_generate(input)
    if quality_check(draft) == "fail":
        draft = llm_improve(draft, feedback)
    final = llm_polish(draft)
    return final
```

2. Parallelization Multiple independent LLM calls executing simultaneously.

When to Use: - Processing multiple data sources - Generating varied content types - Independent analysis tasks

Example Structure:

```
def workflow(topic):
    # Parallel execution
    summary_future = llm_summarize(topic)
    analysis_future = llm_analyze(topic)
    keywords_future = llm_extract_keywords(topic)

    # Aggregate results
    return combine(summary_future.result(),
                   analysis_future.result(),
                   keywords_future.result())
```

3. Routing Directing inputs to specialized handlers based on classification.

When to Use: - Customer service categorization - Document type processing - Multi-intent handling

Example Structure:

```
def workflow(query):
    category = llm_classify(query)
```

```python
    if category == "technical":
        return technical_handler(query)
    elif category == "billing":
        return billing_handler(query)
    else:
        return general_handler(query)
```

4. Orchestrator-Worker Orchestrator breaks down tasks and delegates to workers.

When to Use: - Report generation with sections - Complex document processing - Multi-component analysis

Example Structure:

```python
def workflow(task):
    # Orchestrator plans
    subtasks = orchestrator_plan(task)

    # Workers execute in parallel
    results = [worker_execute(st) for st in subtasks]

    # Orchestrator synthesizes
    return orchestrator_synthesize(results)
```

5. Evaluator-Optimizer Generate-evaluate-refine loop until quality criteria met.

When to Use: - Content meeting specific criteria - Iterative refinement tasks - Quality-gated outputs

Example Structure:

```python
def workflow(requirements):
    output = generator(requirements)

    while True:
        evaluation = evaluator(output, requirements)
        if evaluation.passed:
            break
        output = optimizer(output, evaluation.feedback)

    return output
```

1.4 Agents Explained

1.4.1 Key Characteristics

1. **Autonomous Decision-Making**: Agent chooses tools and actions
2. **Dynamic Tool Usage**: Tools selected based on context
3. **Feedback Loops**: Continuous observation-action cycles
4. **Adaptive Behavior**: Responds to environmental changes

1.4.2 Agent Architecture

```python
# Core Agent Loop
def agent(input):
    state = initialize(input)

    while not is_complete(state):
        # Agent decides next action
        action = llm_decide(state, available_tools)

        # Execute action
        result = execute_tool(action)

        # Update state with feedback
        state = update_state(state, result)

    return state.final_answer
```

1.4.3 Agent Components

1. Tools Functions the agent can invoke:

```python
@tool
def search_database(query: str) -> str:
    """Search internal database for information."""
    return database.search(query)

@tool
def calculate(expression: str) -> float:
    """Evaluate mathematical expression."""
    return eval(expression)
```

2. Memory State that persists across agent decisions:

```python
class AgentState(TypedDict):
    messages: Annotated[list[AnyMessage], operator.add]
    context: dict
    tool_results: list
    iterations: int
```

3. Decision Logic The agent's reasoning process:

```python
def agent_node(state):
    # Agent sees state and available tools
    response = llm_with_tools.invoke(state["messages"])

    # Agent decides: answer or use tool
    if response.tool_calls:
        return {"action": "use_tool", "tool_calls": response.tool_calls}
```

```python
    else:
        return {"action": "respond", "response": response}
```

1.5 Decision Framework

1.5.1 Choose Workflows When:

Process is well-defined - Steps are known in advance - Decision points are clear - Quality gates are explicit

Determinism is important - Need consistent execution paths - Debugging requires clarity - Compliance requires auditability

Subtasks are independent - Can parallelize operations - Each step is verifiable - Clear input-output contracts

Examples: - Document translation pipeline - Report generation with sections - Multi-step content validation - Structured data processing

1.5.2 Choose Agents When:

Problem is open-ended - Solution path unknown upfront - Requires exploration - Multiple valid approaches

Tool selection is dynamic - Agent must choose appropriate tools - Context determines actions - Adaptive behavior needed

Iterative refinement required - Feedback loops essential - Self-correction needed - Learning from attempts

Examples: - Research assistants - Code debugging - Complex problem solving - Interactive troubleshooting

---

1.6 Architecture Patterns

1.6.1 Hybrid Approach: Workflows with Agent Nodes

Combine both paradigms for optimal results:

```python
def hybrid_system(input):
    # Workflow structure
    classified = routing_llm(input)

    if classified == "simple":
        # Use workflow for simple cases
        return simple_workflow(input)
    else:
        # Use agent for complex cases
        return agent_system(input)
```

1.6.2 Multi-Agent Workflows

Orchestrate multiple specialized agents:

```python
def multi_agent_workflow(task):
    # Orchestrator (workflow) coordinates agents
    plan = orchestrator_plan(task)

    results = []
    for subtask in plan:
        # Each subtask handled by specialized agent
        agent = select_agent(subtask.type)
        result = agent.execute(subtask)
        results.append(result)

    return synthesize(results)
```

1.7 Real-World Use Cases

1.7.1 Workflow Use Cases

1. Content Localization Pipeline

```
Input (EN) → Translate (LLM) → Cultural Check (LLM) →
Quality Gate → Refinement (if needed) → Output (ES)
```

Why Workflow: Fixed stages, clear quality criteria, verifiable steps

2. Financial Report Generation

```
Data → Section Analysis (Parallel LLMs) →
Orchestrator Combines → Executive Summary → Final Report
```

Why Workflow: Known report structure, parallel processing, deterministic output

3. Resume Screening System

```
Resume → Extract Info (LLM) → Category Router →
[Technical/Creative/Management] Handler → Ranking
```

Why Workflow: Clear categorization, specialized processing, audit trail

1.7.2 Agent Use Cases

1. Research Assistant

```
Question → Agent decides: [Search/Read/Synthesize] →
Evaluates completeness → More research or Answer
```

Why Agent: Unknown information needs, dynamic tool selection, iterative refinement

2. Code Debugger

```
Bug Report → Agent: [Read code/Run tests/Check logs] →
Hypothesis → [Test/Verify] → Solution or iterate
```

Why Agent: Unpredictable debugging path, adaptive strategy, tool choice depends on findings

3. Customer Support Bot

```
Query → Agent: [Search KB/Check account/Escalate] →
Response → Verify satisfaction → Follow-up or close
```

Why Agent: Varied customer needs, context-dependent actions, learning from interactions

---

1.8 Implementation Examples

1.8.1 Workflow Implementation (Evaluator-Optimizer)

```python
from langgraph.graph import StateGraph, START, END
from typing_extensions import TypedDict

class State(TypedDict):
    task: str
    output: str
    feedback: str
    iterations: int

def generator(state: State):
    result = llm.invoke(f"Generate: {state['task']}")
    return {"output": result.content, "iterations": state.get("iterations", 0) + 1}

def evaluator(state: State):
    evaluation = llm.invoke(f"Evaluate: {state['output']} for task: {state['task']}")
    return {"feedback": evaluation.content}

def should_continue(state: State):
    if "approved" in state["feedback"].lower() or state["iterations"] >= 3:
        return END
    return "generator"

# Build workflow
workflow = StateGraph(State)
workflow.add_node("generator", generator)
workflow.add_node("evaluator", evaluator)
workflow.add_edge(START, "generator")
workflow.add_edge("generator", "evaluator")
workflow.add_conditional_edges("evaluator", should_continue, ["generator", END])

app = workflow.compile()
```

1.8.2 Agent Implementation

```python
from langgraph.graph import StateGraph, MessagesState
from langchain_core.tools import tool

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

@tool
def calculate(expr: str) -> str:
    """Calculate mathematical expression."""
    return str(eval(expr))

tools = [search, calculate]
llm_with_tools = llm.bind_tools(tools)

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

def tool_node(state: MessagesState):
    results = []
    for tool_call in state["messages"][-1].tool_calls:
        tool = {t.name: t for t in tools}[tool_call["name"]]
        result = tool.invoke(tool_call["args"])
        results.append(ToolMessage(content=result, tool_call_id=tool_call["id"]))
    return {"messages": results}

def should_continue(state: MessagesState):
    if state["messages"][-1].tool_calls:
        return "tools"
    return END

# Build agent
agent_graph = StateGraph(MessagesState)
agent_graph.add_node("agent", agent_node)
agent_graph.add_node("tools", tool_node)
agent_graph.add_edge(START, "agent")
agent_graph.add_conditional_edges("agent", should_continue, ["tools", END])
agent_graph.add_edge("tools", "agent")

agent_app = agent_graph.compile()
```

1.9 Best Practices

1.9.1 Workflow Best Practices

1. **Keep Nodes Focused**
   - Each node does one thing well
   - Clear input-output contracts
   - Easy to test independently
2. **Use Type Hints**

```python
class State(TypedDict):
    input: str
    result: str
    metadata: dict
```

3. **Implement Quality Gates**
   - Validate outputs at each stage
   - Explicit pass/fail criteria
   - Feedback loops for failures
4. **Parallelize When Possible**
   - Identify independent operations
   - Use Send API for dynamic parallelization
   - Aggregate results efficiently
5. **Make Routing Explicit**
   - Clear decision logic
   - Enum types for routes
   - Comprehensive edge cases

1.9.2 Agent Best Practices

1. **Design Clear Tools**

```python
@tool
def well_designed_tool(param: str) -> str:
    """
    Clear description of what the tool does.

    Args:
        param: Specific parameter description

    Returns:
        Specific return value description
    """
    return result
```

2. **Implement Safety Limits**

```python
class AgentState(TypedDict):
    messages: list
    iterations: int
    max_iterations: int  # Prevent infinite loops
```

3. **Add Human-in-the-Loop**

```python
def should_continue(state):
    if state["iterations"] > 5:
        return "human_review"  # Escalate complex cases
    if state["messages"][-1].tool_calls:
        return "tools"
    return END
```

4. **Log Agent Decisions**

   - Track tool selections
   - Record reasoning
   - Monitor performance

5. **Handle Failures Gracefully**

```python
def tool_node(state):
    try:
        result = execute_tool(state)
        return {"messages": [result]}
    except Exception as e:
        return {"messages": [ToolMessage(
            content=f"Error: {str(e)}",
            tool_call_id=state["messages"][-1].tool_calls[0]["id"]
        )]}
```

1.10 Common Pitfalls

1.10.1 Workflow Pitfalls

Over-complicating Simple Tasks

```python
# Don't do this for simple tasks
def overly_complex_workflow(text):
    analyzed = llm_analyze(text)
    categorized = llm_categorize(analyzed)
    processed = llm_process(categorized)
    return llm_format(processed)

# Just do this
def simple_approach(text):
    return llm.invoke(f"Process this: {text}")
```

Ignoring Error States - Always handle LLM failures - Provide fallback paths - Don't assume perfect execution

Tight Coupling - Keep nodes independent - Avoid hidden dependencies - Make state explicit

1.10.2 Agent Pitfalls

Infinite Loops

```
# Always add iteration limits
class State(TypedDict):
    iterations: int
    max_iterations: int  # Required!

def should_continue(state):
    if state["iterations"] >= state["max_iterations"]:
        return END  # Safety exit
    # ... rest of logic
```

Too Many Tools to start with - Limit to 5-10 tools per agent (Although langgraph can handle way too many but go iteratively) - Group related functions - Consider tool hierarchies

Unclear Tool Descriptions

```
# Bad
@tool
def process(data):
    """Process data."""  # Too vague!

# Good
@tool
def extract_email_addresses(text: str) -> list[str]:
    """Extract all email addresses from the given text.

    Args:
        text: Input text to search for email addresses

    Returns:
        List of email addresses found in the text
    """
```

No Observability - Implement logging - Track decision paths - Monitor tool usage - Measure performance

---

1.11 Performance Considerations

1.11.1 Workflow Optimization

1. **Minimize Sequential LLM Calls**
    - Combine prompts when possible
    - Use structured outputs to reduce steps
2. **Cache Intermediate Results**

```python
@lru_cache(maxsize=100)
def expensive_llm_call(input: str):
    return llm.invoke(input)
```

3. **Batch When Possible**
   - Process multiple inputs together
   - Use LLM batch APIs

1.11.2 Agent Optimization

1. **Limit Tool Calls**
   - Set max iterations
   - Encourage efficient tool use in prompts
2. **Use Streaming**

```python
for chunk in agent.stream(input, stream_mode="updates"):
    process_chunk(chunk)  # Handle results as they arrive
```

3. **Implement Caching**
   - Cache tool results
   - Store common patterns

1.12 Conclusion

1.12.1 Decision Checklist

Use Workflows if: - [ ] Process steps are known - [ ] Path is mostly deterministic - [ ] Need strong auditability - [ ] Independent subtasks exist - [ ] Quality gates are clear

Use Agents if: - [ ] Problem is exploratory - [ ] Tool choice is context-dependent - [ ] Need adaptive behavior - [ ] Feedback loops are essential - [ ] Path emerges from context

Use Hybrid if: - [ ] Some parts are structured, others exploratory - [ ] Want workflow reliability with agent flexibility - [ ] Different user types need different approaches

1.12.2 Next Steps

1. **Experiment**: Start with simple workflows, evolve to agents
2. **Measure**: Track performance, costs, and user satisfaction
3. **Iterate**: Refine based on real-world usage
4. **Scale**: Build on patterns that work

1.13 Attribution & License

This guide adapts content from the official LangGraph documentation and tutorials.

Source: LangGraph Official Documentation

Aparsoft's Adaptations: This guide has been enhanced by Aparsoft Private Limited with additional examples, explanations, decision frameworks, and production insights specific to our developer community.

1.14 Get Help & Connect

1.14.1 Learning & Community

- **YouTube:** @aparsoft-ai - Main tutorial channel
- **Github:** Join our community - Check our repos and examples (aparsoft-tutorial-resources)
- **GitHub Discussions:** Ask questions about the code
- **LinkedIn:** /company/aparsoft - Articles and tips
- **X (formerly Twitter):** @aparsoft - Tutorials and updates

---

Last Updated: October 2025
Version: 1.0

[ ]:

```

```