AI Workflow Inheritance

Control flow specialization example

Christoph Bussler 5 min read · Just now

Integrating agentic workflows into existing business logic presents significant architectural challenges. Workflow inheritance simplifies this by separating concerns — effectively modularizing different workflow aspects.

This blog reviews the concept of workflow aspects, explores the concept of workflow inheritance, and provides a concrete example of control flow inheritance within the LangGraph framework.

Workflow aspects

Understanding Workflow Aspects

Building structured agentic workflows draws from the foundational "workflow perspectives" introduced here: *Workflow Management — Modeling Concepts, Architecture and Implementation* (https://www.real-programmer.com/publication/books/WorkflowManagementJablonskiBussler.pdf). These Workflow Aspects decouple the "what", "how", "when", "who" and the data of a business process:

- **Functional aspect (what)**: Defines the high-level tasks to be accomplished (e.g., "Review Mortgage Application").

- **Implementation aspect (how)**: Specifies the technical execution of a task (e.g., using an LLM to extract data or a Python function to call the DocuSign API).
- **Control Flow aspect (when)**: Establishes the execution order like sequential, parallel, or conditional (e.g., “Signing occurs only after approval”).
- **Data aspect (data)**: Manages the state, inputs, and outputs of the workflow (e.g., passing a JSON application object between nodes).
- **Organizational aspect (who)**: Identifies the actor performing the task, typically a human, an agent, or a system.

Applying Aspects to LangGraph

In LangGraph (https://www.langchain.com/langgraph), these abstract concepts translate directly into graph components. The following example demonstrates how to implement and identify these aspects within a graph definition:

```
# ================================================================================
# === Top level workflow (start of inheritance tree)
# ================================================================================

class TopLevelWorkflow:
   def __init__(self):
       self._langgraph: Optional[StateGraph] = None
       self._langgraph_compiled: Optional[CompiledStateGraph] = None
       self._functional_aspect_graph()
       self._functional_aspect_nodes()
       self._controlflow_aspect()
       self._compile()

   # ================================================================================
   # === Data and dataflow aspect: input and output data
   # ================================================================================

   class InputState(TypedDict):
```

```
    name: str

class OutputState(TypedDict):
    summary: str

class _OverallState(TypedDict):
    name: str
    state_node_1: Optional[str]
    state_node_2: Optional[str]

# ================================================================================
# === Functional aspect: graph and nodes
# ================================================================================

def _node_start(
        self,
        input_state: InputState) -> _OverallState:
    node_return: dict = self._node_start_implementation(input_state)
    return node_return

def _node_1(self, overall_state: _OverallState) -> _OverallState:
    node_return: dict = self._node_1_implementation(overall_state)
    return node_return

def _node_2(
        self,
        overall_state: _OverallState) -> _OverallState:
    node_return: dict = self._node_2_implementation(overall_state)
    return node_return

def _node_end(
        self,
        overall_state: _OverallState) -> OutputState:
    node_return: dict = self._node_end_implementation(overall_state)
    return node_return

def _functional_aspect_graph(self) -> None:
    self._langgraph = StateGraph(
        TopLevelWorkflow._OverallState,
        input_schema=TopLevelWorkflow.InputState,
        output_schema=TopLevelWorkflow.OutputState)

def _functional_aspect_nodes(self) -> None:
    self._langgraph.set_entry_point("node_start")
    self._langgraph.set_finish_point("node_end")

    self._langgraph.add_node("node_start", self._node_start)
    self._langgraph.add_node("node_end", self._node_end)
    self._langgraph.add_node("node_1", self._node_1)
    self._langgraph.add_node("node_2", self._node_2)
```

```
    # ============================================================================
    # === Implementation aspect: logic for node implementation
    # ============================================================================

    def _node_start_implementation(
            self,
            input_state: InputState) -> dict:
        print("node_start (input_state): ", input_state)
        # Initialize the internal state to a defined set of values (None in
        # this case)
        return {"name": input_state["name"],
                "state_node_1": None,
                "state_node_2": None}

    def _node_1_implementation(
            self,
            overall_state: _OverallState) -> dict:
        print("node_1 (overall_state): ", overall_state)
        return {"state_node_1": "node_1 completed"}

    def _node_2_implementation(
            self,
            overall_state: _OverallState) -> dict:
        print("node_2 (overall_state): ", overall_state)
        return {"state_node_2": "node_2 completed"}

    def _node_end_implementation(
            self,
            overall_state: _OverallState) -> dict:
        print("node_end (overall_state): ", overall_state)
        summary: str = (overall_state['name'] +
                        ": " +
                        overall_state['state_node_1'] +
                        "; " +
                        overall_state['state_node_2'])
        return {"summary": summary}

    # ============================================================================
    # === Control flow aspect
    # ============================================================================

    # Control flow aspect is not implemented and must be specified by
    # an inheriting workflow for execution

    def _controlflow_aspect(self) -> None:
        raise NotImplementedError("Control flow aspect not implemented")

    # ============================================================================
    # LangGraph workflow functions
```

```
    # ========================================================================

    def _compile(self) -> None:
        self._langgraph_compiled = self._langgraph.compile()

    def compiled_workflow(self) -> CompiledStateGraph:
        return self._langgraph_compiled

    def invoke(
            self,
            input_state: InputState) -> OutputState:
        output_state: TopLevelWorkflow.OutputState = (
            self._langgraph_compiled.invoke(input=input_state))
        return output_state
```

The initial example leaves the control flow unspecified; later sections demonstrate how specialized subclasses define this aspect via workflow inheritance. Note that in LangGraph, a “node” is functionally equivalent to a “workflow step”.

While implementation and terminology vary across platforms, these core workflow aspects remain universal. LangGraph manages execution through the Pregel algorithm (https://docs.langchain.com/oss/python/langgraph/pregel), whereas other systems may utilize Petri Nets (https://en.wikipedia.org/wiki/Petri_net), dependency-based resolution, or other execution semantics.

Workflow inheritance

Introduction

Workflow inheritance enables the subclassing of workflows, where specific aspects are inherited, extended, or overridden as introducted here *Process Inheritance* (https://dl.acm.org/doi/10.5555/646090.680396). Just as in

object-oriented programming, a workflow type can serve as a superclass for multiple specialized subclasses across various layers.

Subclasses inherit all specifications from their parent types while maintaining the ability to overwrite existing logic, add new requirements or implement intentionally left undefined implementations. Effectively designing these hierarchies is an iterative engineering process to determine the optimal placement for each workflow aspect within the hierarchy.

Benefits of workflow inheritance

Workflow inheritance provides several strategic advantages for managing complex agentic systems:

- **Exploration and testing**: Evaluate alternative implementations within distinct subclasses, e.g., comparing sequential vs. concurrent node execution.
- **Versioning**: Manage evolving requirements or bug fixes by introducing new sub-workflows as versioned iterations.
- **Process variants**: Maintain a core organizational process while allowing individual business units to specialize logic for their unique needs.
- **Correctness and optimization**: Use a base workflow to establish a "gold standard" for data correctness, then optimize subclasses for latency or resource utilization without losing the baseline.
- **Specification clarity**: Isolate specific perspectives to simplify design and debugging. Explicitly representing differences between layers improves both documentation and system explainability.

- **Continuous development**: Develop and refine new logic within a subclass before promoting it to a primary production level.

Example

The following section demonstrates control flow inheritance. Two subclasses build upon the earlier workflow `TopLevelWorkflow` to define sequential and concurrent execution patterns. Sequential execution establishes the baseline for correctness. Parallel execution optimizes non-functional aspects such as throughput, latency, and resource utilization while maintaining results identical to the sequential version.

Control flow workflow inheritance example

The `TopLevelWorkflow` is subclassed twice to define specific control flows. One subclass implements sequential node execution and the other implements concurrent execution.

Sequential control flow

The following subclass overrides the `TopLevelGraph` control flow to execute nodes sequentially.

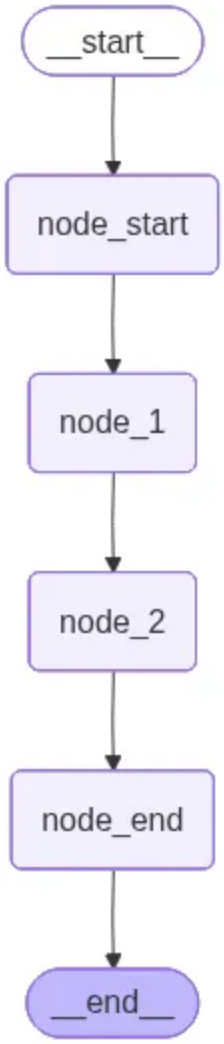
```
class SequentialWorkflow(TopLevelWorkflow):

    # ================================================================================
    # === Control flow aspect
    # ================================================================================

    def _controlflow_aspect(self) -> None:
        self._langgraph.add_edge("node_start", "node_1")
        self._langgraph.add_edge("node_1", "node_2")
        self._langgraph.add_edge("node_2", "node_end")
```

Subclassing improves design clarity by isolating individual workflow aspects.

The diagram below illustrates the complete structure.



Sequential control flow

Concurrent control flow

This subclass overrides the `TopLevelGraph` control flow to execute nodes concurrently.

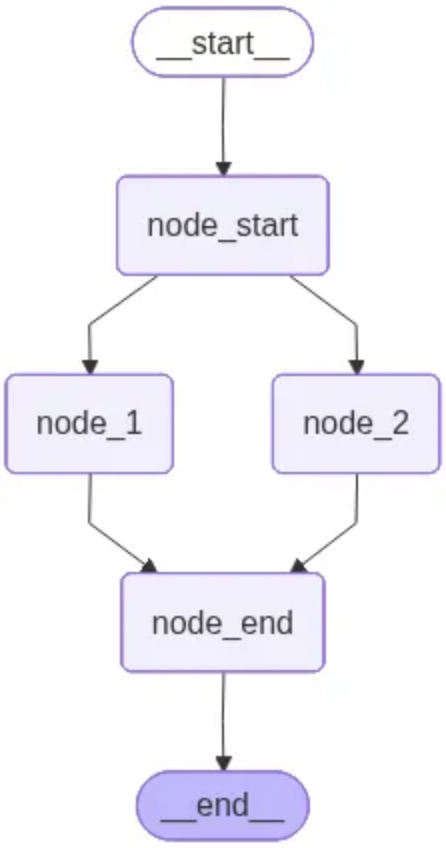
```
class ConcurrentWorkflow(TopLevelWorkflow):

    # ================================================================================
```

```
    # === Control flow aspect
    # ================================================================================

    def _controlflow_aspect(self) -> None:
        self._langgraph.add_edge("node_start", "node_1")
        self._langgraph.add_edge("node_start", "node_2")
        self._langgraph.add_edge("node_1", "node_end")
        self._langgraph.add_edge("node_2", "node_end")
```

The diagram below illustrates the complete structure.



Concurrent control flow

Independent LangGraph steps are concurrent rather than parallel. Review the execution semantics here: *LangGraph Execution Semantics* (https://chbussler.medium.com/langgraph-execution-semantics-c7dd89900ed4).

Discussion: data consistency

Concurrent implementations must yield the same data state as sequential versions. Sequential flows simplify modeling and debugging by avoiding resource synchronization. Concurrency improves efficiency and throughput without compromising correctness.

Organizations can adopt a design principle of developing sequential workflows before optimizing for concurrency.

Summary

Workflow inheritance through workflow aspects is a powerful way to specialize workflows. It enables tailored implementations for specific business contexts (like different departments) while supporting expressiveness and clarity in engineering and maintenance.

This blog demonstrates the workflow inheritance approach using LangGraph, though the principles of workflow subclassing apply to any workflow system.

AI | Agentic Workflow | Workflow | Inheritance | Workflow Aspects