GenAI-Friendly Architecture Pattern

Building AI-Ready Systems Today

Hugues Dtankouo

Senior Python Developer - Gen AI

linkedin.com/in/dtankouo

November 1, 2025

Abstract

The AI landscape evolves rapidly. Model Context Protocol just arrived. New orchestration tools emerge constantly. In 6 months, something else will reshape the field. This pace creates two challenges:

For teams still deciding: Which orchestrator to choose? LangGraph? Temporal? Airflow? Something new? Investing months in the wrong tool means wasted time and resources. Result: paralysis.

For teams already committed: Deep integration with one tool creates coupling. When the next evolution arrives, migration becomes prohibitively expensive. Result: lock-in.

This document presents a **GenAI-Friendly Architecture Pattern** that solves both problems. It enables you to build working systems today without committing to any orchestrator, maintain maximum flexibility to adopt any AI orchestration tool later, migrate between tools without rewriting business logic, and experiment freely without fear of wrong choices.

The pattern is simple: structure your business logic as pure functions with standardized interfaces. These functions work standalone today and integrate seamlessly with any orchestration platform tomorrow. The approach works for both new projects and existing systems, requiring no special tools or frameworks.

Contents

1 The Context: Why This Pattern Matters Now **3**
- 1.1 The Rapid Evolution Problem . . . 3
- 1.2 Two Common Responses . . . 3
- 1.3 The Pattern's Answer . . . 3

2 The Pattern **3**
- 2.1 Core Principle . . . 3
- 2.2 Define Your State . . . 3
- 2.3 Create Business Functions . . . 4
- 2.4 Complexity Lives in Business Classes . . . 5

3 Maximum Flexibility: Use It Anywhere **5**
- 3.1 Today: No Dependencies . . . 5
- 3.2 Today: With FastAPI . . . 5
- 3.3 Tomorrow: Any Orchestrator . . . 6
- 3.4 Functions as Tools . . . 7

4 Technology Independence: Your Insurance Policy **7**
- 4.1 The Core Benefit . . . 7
- 4.2 What Changes, What Stays . . . 8
- 4.3 Real-World Impact . . . 8

5 Key Benefits **8**

6 Bonus: Revealing Business Coverage **8**
- 6.1 The Hidden Problem . . . 9
- 6.2 What Functions Reveal . . . 9
- 6.3 Why This Matters . . . 9

7 Implementation Guide **9**
- 7.1 Step 1: Identify Business Capabilities . . . 9
- 7.2 Step 2: Define State . . . 10
- 7.3 Step 3: Create Functions (Keep Them Simple) . . . 10
- 7.4 Step 4: Apply Pre-Encapsulation . . . 10
- 7.5 Step 5: Start Simple, Add Orchestration Later . . . 10

8 Conclusion **11**

1 The Context: Why This Pattern Matters Now

1.1 The Rapid Evolution Problem

Generative AI and workflow orchestration evolve faster than traditional enterprise technology. What's cutting-edge today might be legacy in 18 months. This creates legitimate concerns:

- **Choosing wrong**: Betting on a tool that becomes obsolete
- **Training overhead**: Onboarding teams to frameworks that won't last
- **Migration cost**: Rebuilding everything when better options emerge

1.2 Two Common Responses

Response 1: Wait and See

Teams freeze, analyzing options endlessly. Meanwhile, competitors ship AI features.

Response 2: All-In Commitment

Teams deeply integrate with one framework. Business logic becomes inseparable from orchestrator primitives. Future migration becomes nearly impossible.

1.3 The Pattern's Answer

Act now. Keep maximum flexibility.

Structure your code to be orchestrator-independent. Your functions work today without any framework. When you choose an orchestrator (or need to change), integration is trivial.

This isn't about avoiding decisions. It's about **keeping your options open** while moving forward.

2 The Pattern

2.1 Core Principle

Structure business capabilities as pure functions that:

- Have clear business-meaningful names
- Accept a state object as input
- Return an updated state object as output
- Remain completely independent of any orchestration technology
- Delegate complexity to business classes (keep functions simple)

2.2 Define Your State

```python
from typing import TypedDict

class OrderWorkflowState(TypedDict):
    """State for an order processing workflow"""
    # Inputs
    order_id: str | None
    customer_data: dict | None
    items: list[dict] | None
```

```python

    # Processing results
    validation_result: dict | None
    pricing: dict | None
    inventory_check: dict | None

    # Final outputs
    approved: bool | None
    invoice: dict | None
    tracking_info: dict | None

    # Workflow tracking
    current_step: str
    errors: list[str]
```

Listing 1: State definition using TypedDict

2.3 Create Business Functions

```python
def validate_order(state: OrderWorkflowState) -> OrderWorkflowState:
    """
    Validate order against business rules.

    This function orchestrates validation logic but doesn't
    implement it. Complexity lives in the OrderValidator class.
    """
    validator = get_order_validator(state["order_id"])

    # Orchestrate validation steps
    validator.check_customer_credit()
    validator.verify_items_availability()
    validator.apply_business_rules()
    validator.check_regulatory_compliance()

    # Update state
    state["validation_result"] = validator.get_summary()
    state["current_step"] = "validated"

    return state
```

Listing 2: Business function example - order validation

```python
def calculate_pricing(state: OrderWorkflowState) -> OrderWorkflowState:
    """Calculate order pricing."""
    pricer = get_pricing_engine(state["customer_data"])

    pricing_result = pricer.calculate(
        items=state["items"],
        customer_tier=state["customer_data"]["tier"]
    )

    state["pricing"] = pricing_result
    state["current_step"] = "priced"

    return state
```

Listing 3: Business function example - pricing calculation

2.4 Complexity Lives in Business Classes

```python
class OrderValidator:
    """
    Your complex business logic - unchanged.
    This is where real complexity lives and evolves.
    """
    def __init__(self):
        self.rules_engine = RulesEngine()
        self.ml_models = {}
        self.cache = {}
        # Could include LLM clients, agents, APIs, etc.

    def check_customer_credit(self):
        # 100+ lines of complex logic
        # Can include ML models, LLM calls, multi-agent systems
        pass

    def verify_items_availability(self):
        # Complex inventory logic
        pass

    def get_summary(self):
        # Return only what needs to be in state
        return {
            "valid": self.is_valid,
            "score": self.risk_score,
            "flags": self.compliance_flags
        }
```

Listing 4: Business class encapsulating complexity

Key insight: Each function can contain anything - deterministic code, LLM calls, multi-agent orchestrations. The interface stays simple: `state → state`.

3 Maximum Flexibility: Use It Anywhere

3.1 Today: No Dependencies

```python
# No framework needed
state = OrderWorkflowState(
    order_id="12345",
    customer_data={...},
    items=[...],
    current_step="initial",
    errors=[]
)

state = validate_order(state)
state = calculate_pricing(state)
state = approve_order(state)

print(f"Order approved: {state['approved']}")
```

Listing 5: Direct execution without any framework

3.2 Today: With FastAPI

```python
@app.post("/process-order")
def process_order(order_id: str, customer: dict, items: list[dict]):
    state = OrderWorkflowState(
        order_id=order_id,
        customer_data=customer,
        items=items,
        current_step="initial",
        errors=[]
    )

    state = validate_order(state)
    state = calculate_pricing(state)
    state = approve_order(state)

    return {
        "approved": state["approved"],
        "invoice": state["invoice"]
    }
```

Listing 6: Integration with FastAPI

3.3 Tomorrow: Any Orchestrator

The same functions work with any orchestration platform. Examples include LangGraph, Apache Airflow, Prefect, Temporal, and others.

```python
from langgraph.graph import StateGraph

graph = StateGraph(OrderWorkflowState)
graph.add_node("validate", validate_order)
graph.add_node("price", calculate_pricing)
graph.add_node("approve", approve_order)
graph.add_edge("validate", "price")
graph.add_edge("price", "approve")
```

Listing 7: Example: LangGraph integration

```python
from airflow import DAG
from airflow.providers.standard.operators.python import PythonOperator
from datetime import datetime

with DAG(
    dag_id='order_processing',
    start_date=datetime(2025, 1, 1),
    schedule_interval=None
) as dag:
    validate = PythonOperator(
        task_id='validate',
        python_callable=validate_order
    )
    price = PythonOperator(
        task_id='price',
        python_callable=calculate_pricing
    )
    approve = PythonOperator(
        task_id='approve',
        python_callable=approve_order
```

```
    )
    validate >> price >> approve
```

Listing 8: Example: Apache Airflow integration

3.4 Functions as Tools

```python
from mcp.server.fastmcp import FastMCP

mcp = FastMCP(name="order_service")

@mcp.tool()
def validate_order_tool(
    order_id: str,
    customer_data: dict,
    items: list
) -> dict:
    """MCP tool wrapper for order validation"""
    state = OrderWorkflowState(
        order_id=order_id,
        customer_data=customer_data,
        items=items,
        current_step="initial",
        errors=[]
    )
    result = validate_order(state)
    return result["validation_result"]
```

Listing 9: Example: Model Context Protocol (MCP) integration

The pattern is truly universal: Same functions, any integration.

4 Technology Independence: Your Insurance Policy

4.1 The Core Benefit

Your business logic never depends on orchestrator primitives. This means:

Today you can:

- Run functions directly (no framework)
- Use FastAPI for simple cases
- Experiment with different orchestrators without commitment

Tomorrow you can:

- Adopt any orchestration tool (examples: LangGraph, Airflow, Temporal, Prefect, etc.)
- Switch between tools as technology evolves
- Migrate without rewriting business logic

In 2 years you can:

- Move to whatever becomes the new standard
- Your functions remain unchanged
- Only the orchestration layer changes

4.2 What Changes, What Stays

What Stays Constant

```python
# Your business functions - never change
def validate_order(state) -> state: ...
def calculate_pricing(state) -> state: ...
def approve_order(state) -> state: ...

# Your business classes - never change
class OrderValidator: ...
class PricingEngine: ...
```

What Changes (Easily)

```python
# Orchestration layer - swap anytime
# Today: Direct calls
# Tomorrow: Framework X
# Next year: Framework Y
```

4.3 Real-World Impact

Scenario: You build with one orchestrator today. A better option emerges in 12 months.

Traditional approach: Months of rewriting. Business logic entangled with framework primitives. High risk.

This pattern: Change orchestration layer only. Business logic untouched. Hours, not months.

This isn't theoretical. AI tools evolve this fast. Your architecture should match that pace.

5 Key Benefits

Benefit	Description
Start Immediately	Build working systems today without choosing an orchestrator
Zero Lock-In	Switch orchestrators without touching business logic
Risk-Free Experimentation	Try different tools without commitment
Future-Proof	Ready for whatever comes next in AI evolution
Business Focus	Functions map to business capabilities, not technical abstractions
Progressive Enhancement	Add orchestration when needed, not before

6 Bonus: Revealing Business Coverage

Beyond flexibility, this pattern provides an unexpected benefit: **visibility into what your system actually does**.

6.1 The Hidden Problem

Traditional codebases hide business coverage behind technical complexity. A 50,000-line system looks substantial, but what business capabilities does it actually provide?

6.2 What Functions Reveal

When you structure code as business functions, reality becomes visible:

```
# IMPLEMENTED
def validate_order(state): pass  # 5,000 lines in OrderValidator

# IMPLEMENTED (but heavily optimized for one edge case)
def calculate_pricing(state): pass  # 15,000 lines total
    # Including 12,000 lines handling enterprise platinum
    # pricing edge case
    # This single edge case = 24% of entire codebase

# MISSING
def check_inventory(state): pass
def calculate_shipping(state): pass
def calculate_taxes(state): pass
def process_payment(state): pass
def assess_risk(state): pass
def send_notifications(state): pass
def update_inventory(state): pass
def sync_with_partners(state): pass
def setup_returns_process(state): pass

# PARTIAL
def generate_invoice(state): pass  # 3,000 lines, basic cases only
```

Listing 10: E-commerce order processing - actual implementation status

Reality: 12 business functions identified. Only 2 fully implemented. 9 missing. 75% scope gap.

6.3 Why This Matters

- **For stakeholders**: See what they're actually getting for their investment
- **For new developers**: Understand system coverage in minutes
- **For architects**: Identify where resources went (12K lines on one edge case vs missing core features)
- **For planning**: Make informed decisions about scope vs perfection

This transparency is powerful. You can't hide narrow scope behind impressive line counts.

7 Implementation Guide

7.1 Step 1: Identify Business Capabilities

List your system's business responsibilities:

- What capabilities should it provide?
- What would stakeholders list as the system's functions?

- What are the natural workflow boundaries?

7.2 Step 2: Define State

Create a TypedDict with:

- Inputs needed to start
- Results from each step
- Final outputs
- Basic tracking (current step, errors)

Keep state minimal: Only include data that crosses function boundaries.

7.3 Step 3: Create Functions (Keep Them Simple)

One function per business capability:

- Clear business name
- Orchestrates, doesn't implement
- Typically 10-30 lines

7.4 Step 4: Apply Pre-Encapsulation

When complexity grows, extract to business classes:

```python
# File: workflows/order_workflow.py
def validate_order(state: OrderState) -> OrderState:
    """Validate order"""
    validator = OrderValidator()
    validator.execute_validations(state["order_data"])
    state["validation"] = validator.get_summary()
    return state
```

Listing 11: Workflow file stays clean

```python
# File: business/order_validator.py
class OrderValidator:
    """All validation complexity lives here"""
    def execute_validations(self, order_data):
        # 200+ lines of complex logic
        # LLM calls, ML models, multi-agent systems - anything
        self._validate_customer_credit(order_data)
        self._check_inventory_availability(order_data)
        # etc.
```

Listing 12: Complexity lives in business classes

Principle: Workflow file = readable business process map. Complexity = separate business classes.

7.5 Step 5: Start Simple, Add Orchestration Later

```python
# Phase 1: Direct calls (works today)
state = initial_state()
state = step_1(state)
```

```
state = step_2(state)

# Phase 2: Add FastAPI (when needed)
@app.post("/workflow")
def endpoint(data):
    state = initial_state(data)
    state = step_1(state)
    return step_2(state)

# Phase 3: Add orchestrator (when ready)
# Same functions, now orchestrated
graph = StateGraph(WorkflowState)
graph.add_node("step_1", step_1)
graph.add_node("step_2", step_2)
```

Listing 13: Progressive enhancement approach

8 Conclusion

The GenAI landscape evolves rapidly. Your architecture should embrace this, not fear it.

This pattern gives you:

- Freedom to start building today without orchestrator commitment
- Flexibility to experiment with any tool risk-free
- Independence to migrate as technology evolves
- Clarity about what your system actually does

The approach is simple:

- Structure business logic as pure functions
- Keep functions simple (orchestrate, don't implement)
- Delegate complexity to business classes
- Start without frameworks, add orchestration when ready

Whether you're paralyzed by choice or locked into one tool, this pattern offers a path forward. Build AI-ready systems today. Keep maximum flexibility for tomorrow.

Your business logic survives every technology shift. Only the orchestration layer changes. That's how you move fast in a rapidly evolving field.

About the Author

Hugues Dtankouo is a Senior Fullstack Python Developer with 7 years of experience solving complex automation challenges in investment banking and capital markets through sophisticated orchestration of cutting-edge generative AI.

His expertise spans highly regulated environments across investment banking, energy sector, Big Four consulting, and insurance. He specializes in building production-ready AI architectures that transform strategic challenges into innovative solutions for organizations demanding the highest performance standards.