Pattern Architectural GenAI-Friendly

Construire des Systèmes Prêts pour l'IA Aujourd'hui

Hugues Dtankouo

Senior Python Developer - Gen AI

linkedin.com/in/dtankouo

1er novembre 2025

Résumé

Le paysage de l'IA évolue rapidement. Le Model Context Protocol vient d'arriver. De nouveaux outils d'orchestration émergent constamment. Dans 6 mois, quelque chose d'autre va remodeler le domaine. Ce rythme crée deux défis :

Pour les équipes qui décident encore : Quel orchestrateur choisir ? LangGraph ? Temporal ? Airflow ? Quelque chose de nouveau ? Investir des mois dans le mauvais outil signifie du temps et des ressources perdus. Résultat : paralysie.

Pour les équipes déjà engagées : Une intégration profonde avec un outil crée un couplage. Quand la prochaine évolution arrive, la migration devient prohibitivement coûteuse. Résultat : enfermement.

Ce document présente un **Pattern Architectural GenAI-Friendly** qui résout ces deux problèmes. Il vous permet de construire des systèmes fonctionnels aujourd'hui sans vous engager envers un orchestrateur, de maintenir une flexibilité maximale pour adopter n'importe quel outil d'orchestration IA plus tard, de migrer entre outils sans réécrire la logique métier, et d'expérimenter librement sans crainte de mauvais choix.

Le pattern est simple : structurez votre logique métier comme des fonctions pures avec des interfaces standardisées. Ces fonctions fonctionnent de manière autonome aujourd'hui et s'intègrent parfaitement avec n'importe quelle plateforme d'orchestration demain. L'approche fonctionne aussi bien pour les nouveaux projets que pour les systèmes existants, sans nécessiter d'outils ou de frameworks spéciaux.

Table des matières

1 Le Contexte : Pourquoi ce Pattern est Important Maintenant **3**
- 1.1 Le Problème de l'Évolution Rapide . . . 3
- 1.2 Deux Réponses Courantes . . . 3
- 1.3 La Réponse du Pattern . . . 3

2 Le Pattern **3**
- 2.1 Principe Central . . . 3
- 2.2 Définissez Votre État . . . 3
- 2.3 Créez des Fonctions Métier . . . 4
- 2.4 La Complexité Vit dans les Classes Métier . . . 5

3 Flexibilité Maximale : Utilisez-le N'importe Où **5**
- 3.1 Aujourd'hui : Sans Dépendances . . . 5
- 3.2 Aujourd'hui : Avec FastAPI . . . 6
- 3.3 Demain : N'importe Quel Orchestrateur . . . 6
- 3.4 Fonctions comme Outils . . . 7

4 Indépendance Technologique : Votre Police d'Assurance **7**
- 4.1 Le Bénéfice Central . . . 7
- 4.2 Ce Qui Change, Ce Qui Reste . . . 8
- 4.3 Impact Réel . . . 8

5 Bénéfices Clés **8**

6 Bonus : Révéler la Couverture Métier **9**
- 6.1 Le Problème Caché . . . 9
- 6.2 Ce Que les Fonctions Révèlent . . . 9
- 6.3 Pourquoi C'est Important . . . 9

7 Guide d'Implémentation **10**
- 7.1 Étape 1 : Identifier les Capacités Métier . . . 10
- 7.2 Étape 2 : Définir l'État . . . 10
- 7.3 Étape 3 : Créer les Fonctions (Gardez-les Simples) . . . 10
- 7.4 Étape 4 : Appliquer la Pré-Encapsulation . . . 10
- 7.5 Étape 5 : Commencer Simple, Ajouter l'Orchestration Plus Tard . . . 11

8 Conclusion **11**

1 Le Contexte : Pourquoi ce Pattern est Important Maintenant

1.1 Le Problème de l'Évolution Rapide

L'IA générative et l'orchestration de workflows évoluent plus vite que les technologies d'entreprise traditionnelles. Ce qui est à la pointe aujourd'hui pourrait être obsolète dans 18 mois. Cela crée des préoccupations légitimes :

— **Choisir le mauvais outil** : Parier sur un outil qui devient obsolète
— **Coût de formation** : Former les équipes sur des frameworks qui ne dureront pas
— **Coût de migration** : Tout reconstruire quand de meilleures options émergent

1.2 Deux Réponses Courantes

Réponse 1 : Attendre et Voir

Les équipes se figent, analysant les options sans fin. Pendant ce temps, les concurrents déploient des fonctionnalités IA.

Réponse 2 : Engagement Total

Les équipes s'intègrent profondément avec un framework. La logique métier devient inséparable des primitives de l'orchestrateur. La migration future devient presque impossible.

1.3 La Réponse du Pattern

Agissez maintenant. Gardez une flexibilité maximale.

Structurez votre code pour être indépendant de l'orchestrateur. Vos fonctions fonctionnent aujourd'hui sans aucun framework. Quand vous choisissez un orchestrateur (ou devez en changer), l'intégration est triviale.

Ce n'est pas une question d'éviter les décisions. C'est une question de **garder vos options ouvertes** tout en avançant.

2 Le Pattern

2.1 Principe Central

Structurez les capacités métier comme des fonctions pures qui :

— Ont des noms clairs ayant un sens métier
— Acceptent un objet d'état en entrée
— Retournent un objet d'état mis à jour en sortie
— Restent complètement indépendantes de toute technologie d'orchestration
— Délèguent la complexité aux classes métier (gardez les fonctions simples)

2.2 Définissez Votre État

```python
from typing import TypedDict

class OrderWorkflowState(TypedDict):
    """État pour un workflow de traitement de commande"""
    # Entrées
    order_id: str | None
```

```
    customer_data: dict | None
    items: list[dict] | None

    # Résultats du traitement
    validation_result: dict | None
    pricing: dict | None
    inventory_check: dict | None

    # Sorties finales
    approved: bool | None
    invoice: dict | None
    tracking_info: dict | None

    # Suivi du workflow
    current_step: str
    errors: list[str]
```

Listing 1 – Définition de l'état avec TypedDict

2.3 Créez des Fonctions Métier

```
def validate_order(state: OrderWorkflowState) -> OrderWorkflowState:
    """
    Valide la commande selon les règles métier.

    Cette fonction orchestre la logique de validation mais ne
    l'implémente pas. La complexité vit dans la classe OrderValidator.
    """
    validator = get_order_validator(state["order_id"])

    # Orchestre les étapes de validation
    validator.check_customer_credit()
    validator.verify_items_availability()
    validator.apply_business_rules()
    validator.check_regulatory_compliance()

    # Met à jour l'état
    state["validation_result"] = validator.get_summary()
    state["current_step"] = "validated"

    return state
```

Listing 2 – Exemple de fonction métier - validation de commande

```
def calculate_pricing(state: OrderWorkflowState) -> OrderWorkflowState:
    """Calcule le prix de la commande."""
    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 – Exemple de fonction métier - calcul de prix

2.4 La Complexité Vit dans les Classes Métier

```python
class OrderValidator:
    """
    Votre logique métier complexe - inchangée.
    C'est ici que vit et évolue la vraie complexité.
    """
    def __init__(self):
        self.rules_engine = RulesEngine()
        self.ml_models = {}
        self.cache = {}
        # Peut inclure des clients LLM, agents, APIs, etc.

    def check_customer_credit(self):
        # 100+ lignes de logique complexe
        # Peut inclure des modèles ML, appels LLM, systèmes multi-agents
        pass

    def verify_items_availability(self):
        # Logique d'inventaire complexe
        pass

    def get_summary(self):
        # Retourne seulement ce qui doit être dans l'état
        return {
            "valid": self.is_valid,
            "score": self.risk_score,
            "flags": self.compliance_flags
        }
```

Listing 4 – Classe métier encapsulant la complexité

Insight clé : Chaque fonction peut contenir n'importe quoi - code déterministe, appels LLM, orchestrations multi-agents. L'interface reste simple : `state → state`.

3 Flexibilité Maximale : Utilisez-le N'importe Où

3.1 Aujourd'hui : Sans Dépendances

```python
# Aucun framework nécessaire
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 – Exécution directe sans framework

3.2 Aujourd'hui : Avec 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 – Intégration avec FastAPI

3.3 Demain : N'importe Quel Orchestrateur

Les mêmes fonctions fonctionnent avec n'importe quelle plateforme d'orchestration. Les exemples incluent LangGraph, Apache Airflow, Prefect, Temporal, et d'autres.

```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 – Exemple : intégration LangGraph

```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 – Exemple : intégration Apache Airflow

3.4 Fonctions comme Outils

```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:
    """Wrapper d'outil MCP pour la validation de commande"""
    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 – Exemple : intégration Model Context Protocol (MCP)

Le pattern est vraiment universel : Mêmes fonctions, n'importe quelle intégration.

4 Indépendance Technologique : Votre Police d'Assurance

4.1 Le Bénéfice Central

Votre logique métier ne dépend jamais des primitives de l'orchestrateur. Cela signifie :

Aujourd'hui vous pouvez :

— Exécuter les fonctions directement (sans framework)
— Utiliser FastAPI pour les cas simples
— Expérimenter avec différents orchestrateurs sans engagement

Demain vous pouvez :

— Adopter n'importe quel outil d'orchestration (exemples : LangGraph, Airflow, Temporal, Prefect, etc.)
— Basculer entre outils au fur et à mesure que la technologie évolue
— Migrer sans réécrire la logique métier

Dans 2 ans vous pouvez :

— Passer à ce qui devient le nouveau standard
— Vos fonctions restent inchangées
— Seule la couche d'orchestration change

4.2 Ce Qui Change, Ce Qui Reste

Ce Qui Reste Constant

```python
# Vos fonctions métier - ne changent jamais
def validate_order(state) -> state: ...
def calculate_pricing(state) -> state: ...
def approve_order(state) -> state: ...

# Vos classes métier - ne changent jamais
class OrderValidator: ...
class PricingEngine: ...
```

Ce Qui Change (Facilement)

```python
# Couche d'orchestration - échangeable à tout moment
# Aujourd'hui : Appels directs
# Demain : Framework X
# L'année prochaine : Framework Y
```

4.3 Impact Réel

Scénario : Vous construisez avec un orchestrateur aujourd'hui. Une meilleure option émerge dans 12 mois.

Approche traditionnelle : Des mois de réécriture. Logique métier entremêlée avec les primitives du framework. Risque élevé.

Ce pattern : Changez uniquement la couche d'orchestration. Logique métier intacte. Des heures, pas des mois.

Ce n'est pas théorique. Les outils IA évoluent à cette vitesse. Votre architecture devrait correspondre à ce rythme.

5 Bénéfices Clés

Bénéfice	Description
Démarrage Immédiat	Construisez des systèmes fonctionnels aujourd'hui sans choisir d'orchestrateur
Zéro Enfermement	Changez d'orchestrateurs sans toucher à la logique métier
Expérimentation Sans Risque	Essayez différents outils sans engagement
À l'Épreuve du Futur	Prêt pour ce qui arrive ensuite dans l'évolution de l'IA
Focus Métier	Les fonctions correspondent aux capacités métier, pas aux abstractions techniques
Amélioration Progressive	Ajoutez l'orchestration quand nécessaire, pas avant

6 Bonus : Révéler la Couverture Métier

Au-delà de la flexibilité, ce pattern offre un bénéfice inattendu : **visibilité sur ce que votre système fait réellement**.

6.1 Le Problème Caché

Les bases de code traditionnelles cachent la couverture métier derrière la complexité technique. Un système de 50 000 lignes semble substantiel, mais quelles capacités métier fournit-il réellement ?

6.2 Ce Que les Fonctions Révèlent

Quand vous structurez le code comme des fonctions métier, la réalité devient visible :

```python
# IMPLÉMENTÉ
def validate_order(state): pass  # 5 000 lignes dans OrderValidator

# IMPLÉMENTÉ (mais fortement optimisé pour un cas particulier)
def calculate_pricing(state): pass  # 15 000 lignes au total
    # Incluant 12 000 lignes gérant le cas particulier de
    # tarification enterprise platinum
    # Ce seul cas particulier = 24% de toute la base de code

# MANQUANT
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

# PARTIEL
def generate_invoice(state): pass  # 3 000 lignes, cas de base
    seulement
```

Listing 10 – Traitement de commande e-commerce - statut d'implémentation réel

Réalité : 12 fonctions métier identifiées. Seulement 2 complètement implémentées. 9 manquantes. 75% d'écart de scope.

6.3 Pourquoi C'est Important

— **Pour les parties prenantes** : Voir ce qu'ils obtiennent réellement pour leur investissement
— **Pour les nouveaux développeurs** : Comprendre la couverture du système en minutes
— **Pour les architectes** : Identifier où sont allées les ressources (12K lignes sur un cas particulier vs fonctionnalités de base manquantes)
— **Pour la planification** : Prendre des décisions éclairées sur le scope vs la perfection

Cette transparence est puissante. Vous ne pouvez pas cacher un scope étroit derrière des nombres de lignes impressionnants.

7 Guide d'Implémentation

7.1 Étape 1 : Identifier les Capacités Métier

Listez les responsabilités métier de votre système :

— Quelles capacités devrait-il fournir ?
— Que listeraient les parties prenantes comme fonctions du système ?
— Quelles sont les frontières naturelles du workflow ?

7.2 Étape 2 : Définir l'État

Créez un TypedDict avec :

— Les entrées nécessaires pour démarrer
— Les résultats de chaque étape
— Les sorties finales
— Un suivi basique (étape actuelle, erreurs)

Gardez l'état minimal : Incluez seulement les données qui traversent les frontières des fonctions.

7.3 Étape 3 : Créer les Fonctions (Gardez-les Simples)

Une fonction par capacité métier :

— Nom métier clair
— Orchestre, n'implémente pas
— Typiquement 10-30 lignes

7.4 Étape 4 : Appliquer la Pré-Encapsulation

Quand la complexité grandit, extrayez vers des classes métier :

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

Listing 11 – Le fichier workflow reste propre

```python
# Fichier : business/order_validator.py
class OrderValidator:
    """Toute la complexité de validation vit ici"""
    def execute_validations(self, order_data):
        # 200+ lignes de logique complexe
        # Appels LLM, modèles ML, systèmes multi-agents - n'importe quoi
        self._validate_customer_credit(order_data)
        self._check_inventory_availability(order_data)
        # etc.
```

Listing 12 – La complexité vit dans les classes métier

Principe : Fichier workflow = carte de processus métier lisible. Complexité = classes métier séparées.

7.5 Étape 5 : Commencer Simple, Ajouter l'Orchestration Plus Tard

```python
# Phase 1 : Appels directs (fonctionne aujourd'hui)
state = initial_state()
state = step_1(state)
state = step_2(state)

# Phase 2 : Ajouter FastAPI (quand nécessaire)
@app.post("/workflow")
def endpoint(data):
    state = initial_state(data)
    state = step_1(state)
    return step_2(state)

# Phase 3 : Ajouter l'orchestrateur (quand prêt)
# Mêmes fonctions, maintenant orchestrées
graph = StateGraph(WorkflowState)
graph.add_node("step_1", step_1)
graph.add_node("step_2", step_2)
```

Listing 13 – Approche d'amélioration progressive

8 Conclusion

Le paysage GenAI évolue rapidement. Votre architecture devrait embrasser cela, pas le craindre.

Ce pattern vous donne :

— Liberté de commencer à construire aujourd'hui sans engagement envers un orchestrateur
— Flexibilité pour expérimenter avec n'importe quel outil sans risque
— Indépendance pour migrer au fur et à mesure que la technologie évolue
— Clarté sur ce que votre système fait réellement

L'approche est simple :

— Structurez la logique métier comme des fonctions pures
— Gardez les fonctions simples (orchestrez, n'implémentez pas)
— Déléguez la complexité aux classes métier
— Commencez sans frameworks, ajoutez l'orchestration quand prêt

Que vous soyez paralysé par le choix ou enfermé dans un outil, ce pattern offre une voie vers l'avant. Construisez des systèmes prêts pour l'IA aujourd'hui. Gardez une flexibilité maximale pour demain.

Votre logique métier survit à chaque changement technologique. Seule la couche d'orchestration change. C'est comme ça qu'on avance vite dans un domaine qui évolue rapidement.

À Propos de l'Auteur

Hugues Dtankouo est un développeur Fullstack Python Senior avec 7 ans d'expérience à résoudre des défis d'automatisation complexes dans la banque d'investissement et les marchés de capitaux à travers l'orchestration sophistiquée de l'IA générative de pointe.

Son expertise couvre des environnements hautement régulés dans la banque d'investissement, le secteur de l'énergie, le conseil Big Four, et l'assurance. Il se spécialise dans la construction

d'architectures IA prêtes pour la production qui transforment des défis stratégiques en solutions innovantes pour des organisations exigeant les plus hauts standards de performance.