Guide Pratique des Frameworks et SDK Agentiques

PydanticAI + Logfire | LangGraph | n8n | Spring AI | LangChain4j

Phase : Axe 2 — Agentic AI Shift
Contexte : Programme de Formation Tekkod
Destinataire : Alexandro Disla
Domaine : Secteur Bancaire — Credit Scoring & Risk Assessment

Février 2026

Table des Matières

1. Introduction et Positionnement

Ce document constitue le volet pratique du cursus Agentic AI de Tekkod. Après avoir acquis les fondations théoriques (scaling laws, architecture Transformer, protocole MCP, RAG avancé), il est temps de passer à la maîtrise concrète des outils. L'objectif est de comprendre chaque framework suffisamment pour écrire du code de production, pas simplement copier-coller des exemples.

Nous passons en revue cinq frameworks couvrant deux écosystèmes (Python et Java) et trois niveaux d'abstraction (framework d'agents typés, orchestration par graphe d'état, automatisation workflow). Chaque section suit la même structure : philosophie du framework, architecture interne, exemples de code avec contexte bancaire, et limites.

1.1 Les Trois Niveaux d’Outillage

Niveau	Framework Python	Framework Java	Description
Agent Framework(type-safe)	PydanticAI	Spring AILangChain4j	Définir des agents avec validation structurée, tools typés, et observabilité intégrée
Orchestration(graphe d’état)	LangGraph	LangGraph.js(ou LangChain4j Agentic)	Composer des workflows multi-étapes avec boucles, conditions, et persistance d’état
WorkflowAutomation	n8n	n8n	Automatiser des processus métier avec interface visuelle et 400+ connecteurs

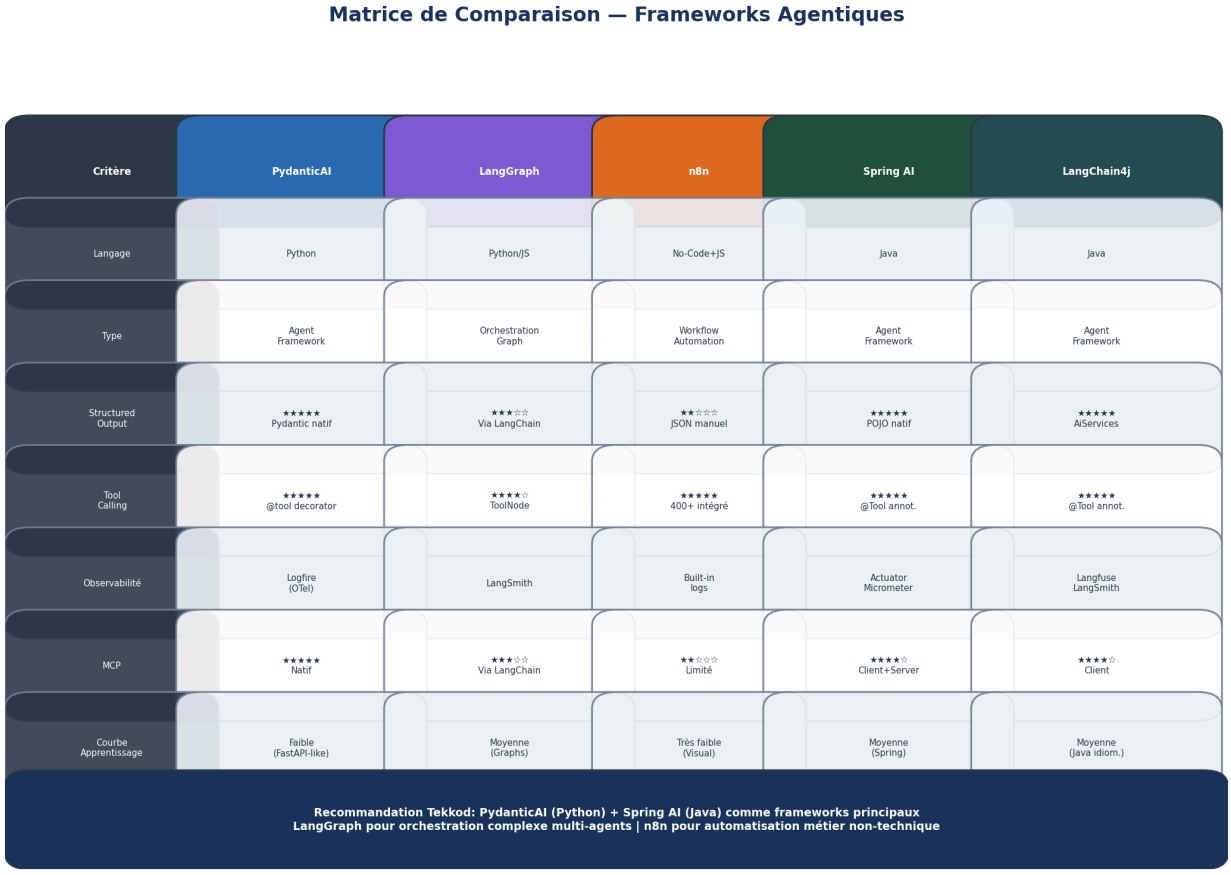

Figure 1 — Matrice de comparaison des frameworks agentiques

2. PydanticAI — Le Framework d'Agents Type-Safe

2.1 Philosophie et Positionnement

PydanticAI est construit par la même équipe que Pydantic (la couche de validation utilisée par l'OpenAI SDK, l'Anthropic SDK, LangChain, FastAPI, et des dizaines d'autres bibliothèques). Leur objectif déclaré : apporter le « FastAPI feeling » au développement d'agents IA.

La philosophie repose sur quatre piliers fondamentaux. Le premier est le type safety intégral : chaque sortie d'agent est validée par un modèle Pydantic, chaque dépendance injectée est typée, et l'IDE peut auto-compléter l'intégralité du code. Le deuxième pilier est l'agnosticisme modèle : PydanticAI supporte plus de 30 providers (OpenAI, Anthropic, Gemini, DeepSeek, Grok, Cohere, Mistral, Perplexity, Ollama, et bien d'autres). Le troisième est l'observabilité native via Logfire. Le quatrième est l'intégration protocoles standards : MCP, Agent2Agent (A2A), et divers standards de streaming UI.

2.2 Architecture Interne

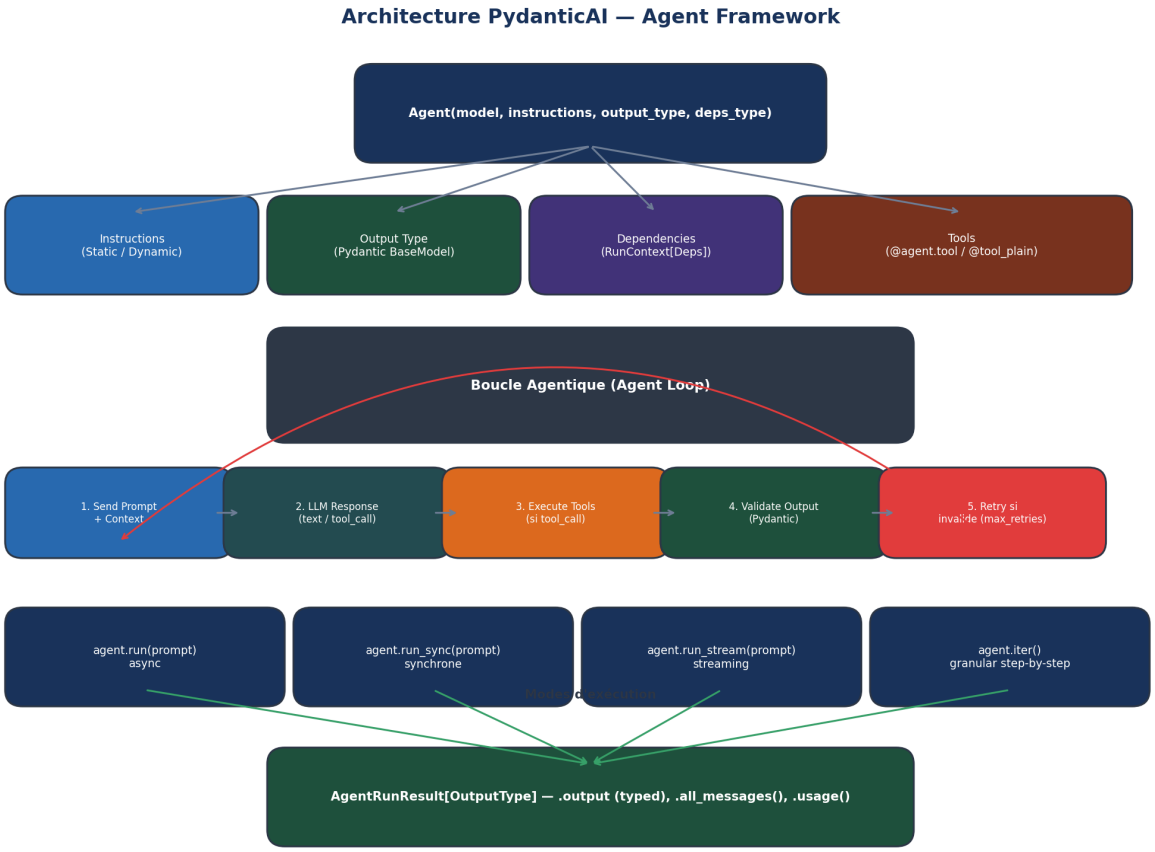

Figure 2 — Architecture interne de PydanticAI

L'architecture se décompose en cinq composants interconnectés :

Agent — Le point d'entrée central

Un Agent est défini par quatre éléments : le modèle LLM à utiliser, les instructions système (statiques ou dynamiques via une fonction), le type de sortie attendu (un BaseModel Pydantic qui sera validé automatiquement), et le type de dépendances (un dataclass injecté dans les tools via RunContext). C'est une unité autonome de raisonnement IA.

Tools — Les capacités d'action

Les tools sont des fonctions Python décorées que le LLM peut invoquer. PydanticAI offre deux décorateurs : @agent.tool pour les fonctions qui ont besoin du RunContext (accès aux dépendances injectées), et @agent.tool_plain pour les fonctions autonomes. Le LLM voit la signature de la fonction et sa docstring, puis décide seul s'il doit l'appeler.

RunContext[Deps] — L'injection de dépendances

RunContext est le mécanisme d'injection de dépendances de PydanticAI. On définit un dataclass de dépendances (connexion DB, client HTTP, etc.), et chaque tool qui en a besoin reçoit un RunContext[Deps] en premier paramètre. Cela permet de tester les agents avec des mocks facilement.

Boucle Agentique — Le cycle d'exécution

Quand on appelle agent.run(), PydanticAI entre dans une boucle : il envoie le prompt au LLM, reçoit la réponse (texte ou tool_call), exécute les tools si demandé, renvoie les résultats au LLM, et répète jusqu'à obtenir une réponse finale. Cette réponse est alors validée par le modèle Pydantic de sortie.

Structured Output — La validation garantie

Si le LLM retourne une réponse qui ne valide pas le schéma Pydantic, PydanticAI retry automatiquement (configurable via max_retries) en renvoyant les erreurs de validation au LLM pour qu'il corrige.

2.3 Exemples de Code — Contexte Bancaire

2.3.1 Agent Simple — Évaluation de Risque

Cet exemple définit un agent qui évalue le risque d'un client bancaire et retourne un résultat structuré validé par Pydantic.

```
from dataclasses import dataclass
from pydantic import BaseModel, Field
from pydantic_ai import Agent, RunContext



# --- Dépendances injectées ---
# Le dataclass sert de conteneur pour tout ce dont les tools ont besoin.
# En production, ce serait une vraie connexion DB, ici simplifié.
@dataclass
class BankDeps:
```

```
    db_connection: str  # URL de connexion à la base credit_scoring_db
    risk_threshold: float = 650.0  # Score minimum pour approbation


# --- Modèle de sortie ---
# Tout ce que l'agent retourne DOIT valider ce schéma.
# Si le LLM retourne un JSON invalide, PydanticAI retry automatiquement.
class RiskAssessment(BaseModel):
    client_name: str = Field(description="Nom complet du client")
    credit_score: int = Field(ge=300, le=850, description="Échelle FICO")
    risk_level: str = Field(description="LOW, MEDIUM, HIGH, ou CRITICAL")
    recommended_rate: float = Field(description="Taux d'intérêt recommandé en %")
    approved: bool = Field(description="Approbation oui/non")
    reasoning: str = Field(description="Explication de la décision")


# --- Définition de l'Agent ---
# model: quel LLM utiliser (agnostique, on peut changer sans toucher le code)
# output_type: force la sortie à valider ce BaseModel
# deps_type: type des dépendances injectées dans les tools
risk_agent = Agent(
    'anthropic:claude-sonnet-4-5',
    output_type=RiskAssessment,
    deps_type=BankDeps,
    instructions=(
        'Tu es un analyste de risque crédit bancaire. '
        'Évalue le risque du client en utilisant les outils disponibles. '
        'Score >= 750: LOW risk, 650-749: MEDIUM, 550-649: HIGH, <550: CRITICAL. '
        'Recommande un taux = base_rate + prime de risque (0-5% selon le niveau).'
    ),
)


# --- Tools ---
# @risk_agent.tool : le tool reçoit RunContext[BankDeps] en premier paramètre
# Le LLM voit le nom de la fonction et la docstring pour décider de l'appeler.
@risk_agent.tool
async def check_credit_history(ctx: RunContext[BankDeps], client_id: int) -> str:
    """Vérifie l'historique de crédit du client dans la base de données."""
    # En production : query SQLAlchemy sur credit_scoring_db
    # ctx.deps.db_connection donne accès à la connexion injectée
    return f"Client {client_id}: score 720, historique 8 ans, revenu 85000 EUR"


@risk_agent.tool
async def get_product_rates(ctx: RunContext[BankDeps], product_code: str) -> str:
```

```
    """Récupère le taux de base et conditions du produit de prêt."""
    return f"Produit {product_code}: base_rate 3.5%, max_amount 500000, term 240 mois"


# --- Exécution ---
async def evaluate_client():
    deps = BankDeps(db_connection='mysql://analyst:analyst@mysql:3306/credit_scoring_db')
    result = await risk_agent.run(
        'Evalue le risque du client ID 42 pour un prêt MORTGAGE',
        deps=deps
    )

    # result.output est typé RiskAssessment (auto-complete IDE)
    assessment = result.output
    print(f'Score: {assessment.credit_score}')
    print(f'Risque: {assessment.risk_level}')
    print(f'Approuvé: {assessment.approved}')
    print(f'Taux: {assessment.recommended_rate}%')
    print(f'Raison: {assessment.reasoning}')

    # Métadonnées d'exécution
    print(f'Tokens utilisés: {result.usage()}')
    print(f'Messages échangés: {len(result.all_messages())}')
```

2.3.2 Instructions Dynamiques

Les instructions peuvent être des fonctions qui s'adaptent au contexte d'exécution.

```
# Les instructions dynamiques reçoivent RunContext et peuvent
# adapter le comportement de l'agent selon le contexte.
@risk_agent.instructions
def dynamic_instructions(ctx: RunContext[BankDeps]) -> str:
    threshold = ctx.deps.risk_threshold
    return (
        f'Seuil d'approbation actuel: {threshold}. '
        f'Les demandes avec un score inférieur à {threshold} '
        f'doivent être flagées pour révision manuelle.'
    )
```

2.3.3 Human-in-the-Loop — Approbation de Tool

PydanticAI permet de demander une approbation humaine avant d'exécuter certains tools sensibles.

```
from pydantic_ai.tools import ToolDefinition

# prepare_tool est un callback appelé AVANT que le tool ne soit proposé au LLM.
```

```
# Si on veut conditionner l'accès, on peut modifier la définition ou la retirer.
@risk_agent.tool(prepare=approve_high_risk_tools)
async def override_risk_level(
    ctx: RunContext[BankDeps], client_id: int, new_level: str
) -> str:
    """Force le niveau de risque d'un client (nécessite approbation)."""
    return f"Risque du client {client_id} mis à jour vers {new_level}"

async def approve_high_risk_tools(
    ctx: RunContext[BankDeps], tool_def: ToolDefinition
) -> ToolDefinition | None:
    # Logique d'approbation (en production: webhook, UI, Slack, etc.)
    response = input(f'Approuver l'outil {tool_def.name}? [y/n]: ')
    return tool_def if response.lower() == 'y' else None
```

2.3.4 Modes d'Exécution

Méthode	Usage	Description
agent.run()	Async (production)	Appel asynchrone, await le résultat complet
agent.run_sync()	Synchrone (scripts)	Bloquant, pour tests et scripts simples
agent.run_stream()	Streaming (UI)	Stream la réponse token par token
agent.iter()	Granulaire (debug)	Itérer sur chaque étape de la boucle agentique

2.4 Fonctionnalités Avancées

MCP (Model Context Protocol)

PydanticAI intègre nativement MCP. On peut connecter un agent à des serveurs MCP externes (Google Drive, Slack, bases SQL) sans écrire de code d'intégration custom. L'agent voit les tools exposés par le serveur MCP exactement comme ses tools locaux.

Agent2Agent (A2A)

Le protocole A2A permet à deux agents de communiquer entre eux, même s'ils sont construits avec des frameworks différents. Par exemple, un agent PydanticAI de scoring pourrait déléguer la vérification de conformité à un agent Google ADK via A2A.

Durable Execution

Pour les workflows longs (approbations asynchrones, pipelines multi-étapes), PydanticAI supporte la persistance de l'état d'exécution. Si le processus crash ou redémarre, l'agent reprend exactement où il en était.

Graph Support

Pour les applications complexes où le flux de contrôle standard devient spaghetti, PydanticAI offre un module de graphes définis par type hints. C'est un mini-LangGraph intégré au framework.

3. Pydantic Logfire — Observabilité AI + Application

3.1 Pourquoi Logfire Existe

Le problème que Logfire résout est précis : les bugs en production IA viennent rarement du LLM seul. Ils se cachent dans les coutures — une requête SQL lente qui retarde la récupération du contexte, un timeout API pendant un tool call, un search vectoriel inefficace. On a besoin de visibilité sur toute la stack, pas juste les appels LLM.

Logfire est une plateforme d'observabilité construite sur OpenTelemetry (OTel) qui trace l'ensemble de l'application : appels LLM, raisonnement des agents, requêtes DB, appels API, recherches vectorielles, logique métier. Le tout dans un dashboard unifié.

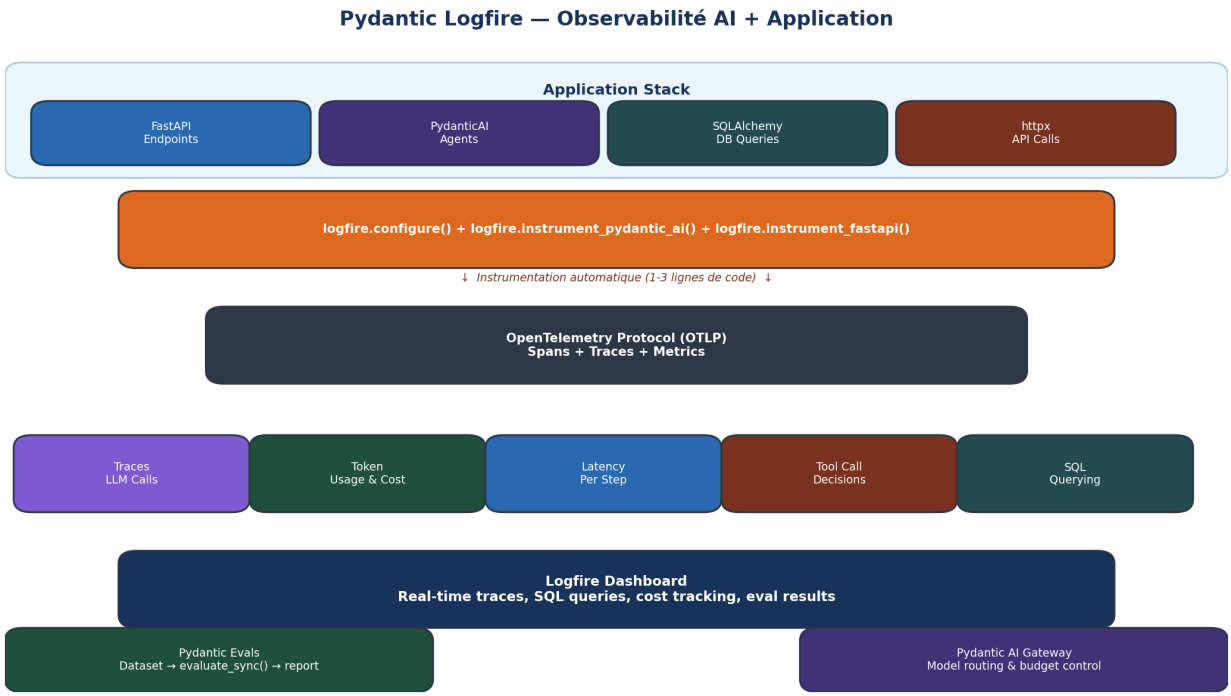

Figure 3 — Architecture d'observabilité Logfire

3.2 Intégration en 3 Lignes

```
import logfire

# 1. Configurer Logfire (crée un projet sur logfire.pydantic.dev)
logfire.configure()
```

```
# 2. Instrumenter PydanticAI (trace chaque appel LLM, tool call, retry)
logfire.instrument_pydantic_ai()

# 3. Instrumenter le reste de la stack (optionnel mais recommandé)
logfire.instrument_fastapi()     # Trace chaque endpoint FastAPI
logfire.instrument_sqlalchemy()   # Trace chaque query SQL
logfire.instrument_httpx()        # Trace chaque appel HTTP sortant
```

C'est tout. À partir de là, chaque exécution d'agent apparaît dans le dashboard Logfire avec un arbre de traces complèt : le prompt envoyé, la réponse LLM, les tool calls déclenchés, les requêtes SQL exécutées, les validations Pydantic, les retries, et les coûts en tokens.

3.3 Concepts Fondamentaux

Concept	Définition	Exemple Bancaire
Span	Unité atomique de télémétrie. Correspond à une étape.	Un appel à check_credit_history() = 1 span
Trace	Arbre de spans parent/enfant.	Agent run complet : prompt → tool → LLM → output
Metric	Valeur agrégée calculée sur les spans.	Coût total en tokens par heure
Log	Enregistrement horodaté d'un événement.	Alerte: client score < 300 détecté

3.4 Exemple Bancaire Complet avec Logfire

```
from pydantic_ai import Agent, RunContext
from dataclasses import dataclass
from pydantic import BaseModel
import logfire

# Configuration Logfire
logfire.configure()               # Connecte au dashboard
logfire.instrument_pydantic_ai()   # Instrumente tous les agents
logfire.instrument_sqlite3()        # Instrumente les queries SQL


@dataclass
class SupportDeps:
    db_conn: str
    client_id: int
```

```
class SupportOutput(BaseModel):
    risk_level: str
    summary: str
    block_card: bool


support_agent = Agent(
    'gateway/openai:gpt-4o',  # Gateway pour routing multi-modèles
    deps_type=SupportDeps,
    output_type=SupportOutput,
    instructions='Tu es un agent support bancaire. Évalue la situation du client.'
)


@support_agent.tool
async def check_recent_transactions(ctx: RunContext[SupportDeps]) -> str:
    """Vérifie les transactions récentes du client."""
    # Chaque appel est automatiquement tracé dans Logfire
    # avec durée, arguments, résultat, et contexte parent
    return "3 transactions suspectes détectées en 2h"


# Chaque run apparaît dans Logfire avec l'arbre complet :
# └ Agent Run (durée totale, tokens, coût)
#   ├ System Prompt
#   ├ User Message
#   ├ LLM Response (tool_call: check_recent_transactions)
#   ├ Tool Execution: check_recent_transactions (150ms)
#   ├ LLM Response (structured output)
#   └ Pydantic Validation: SupportOutput (pass)
```

3.5 Pydantic Evals — Tests Systématiques

Pydantic Evals est un framework open-source de tests pour systèmes IA, intégré à Logfire. On définit des datasets de cas de test, des évaluateurs (exact match, LLM-as-judge, custom), et on exécute les évaluations de manière reproductible. Les résultats sont visualisés dans Logfire pour suivre la performance dans le temps.

3.6 Pydantic AI Gateway

Le Gateway est un composant de routing qui distribue les appels LLM vers différents providers selon des règles configurées : budget (switcher vers un modèle moins cher quand le budget mensuel est atteint), latence (router vers le modèle le plus rapide disponible), ou fallback (basculer vers un modèle de secours si le principal est down). Combiné avec Logfire, on voit en temps réel les coûts par provider et les décisions de routing.

4. LangGraph — Orchestration par Graphe d'État

4.1 Philosophie et Positionnement

LangGraph est un framework d'orchestration bas niveau pour workflows stateful de longue durée. Contrairement à PydanticAI qui fournit un agent clé-en-main, LangGraph donne les primitives pour construire soi-même l'architecture de raisonnement.

LangGraph modélise toute application IA comme un graphe dirigé où les nœuds sont des fonctions (appel LLM, exécution tool, logique métier), les arêtes sont des transitions (fixes ou conditionnelles), et l'état est un dictionnaire typé partagé entre tous les nœuds. Ce modèle permet naturellement les boucles (retry, self-correction), les branchements conditionnels (routing), et l'interruption pour approbation humaine.

LangGraph 1.0 stable a été publié en octobre 2025. Le framework est inspiré de Pregel (Google) et Apache Beam, avec une interface publique inspirée de NetworkX. Il est MIT-licensed et peut être utilisé sans LangChain.

4.2 Concepts Fondamentaux

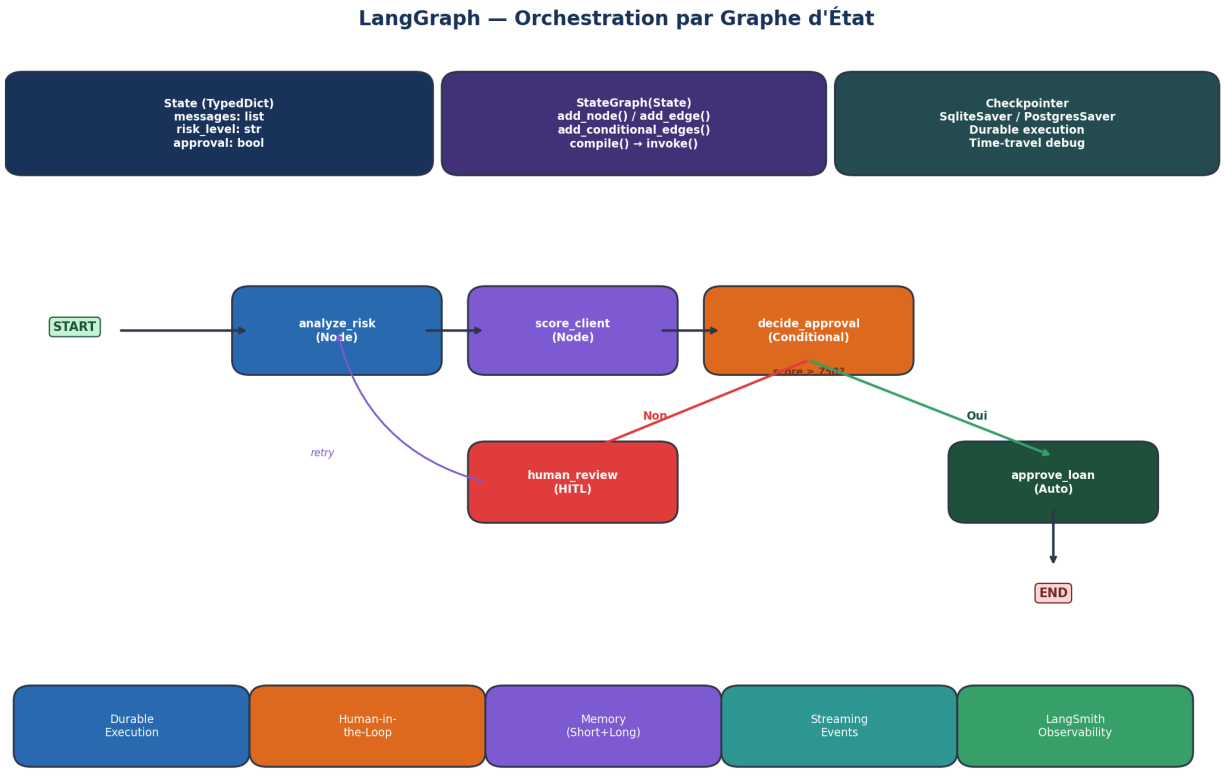

Figure 4 — LangGraph : orchestration par graphe d'état

Concept	Description	Analogue Bancaire
State	TypedDict partagé par tous les nœuds	Dossier de crédit en cours d'évaluation
Node	Fonction qui lit l'état, fait une action, retourne un update	analyze_risk(), score_client()
Edge	Transition fixe entre deux nœuds	Après analyse → toujours scorer
Conditional Edge	Transition dépendant de l'état	Si score >= 750 → auto-approve, sinon → review
Checkpointer	Persistance de l'état (SQLite, PostgreSQL)	Sauvegarder l'état entre les étapes
Human-in-the-Loop	Interruption pour approbation	Validation manuelle d'un crédit > 500K EUR

4.3 Exemple Bancaire — Pipeline de Scoring

```
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END
from langgraph.checkpoint.sqlite import SqliteSaver


# --- État partagé ---
# Tous les nœuds lisent et écrivent dans ce dictionnaire typé.
class CreditState(TypedDict):
    client_id: int
    client_data: dict       # données client chargées
    credit_score: int        # score calculé
    risk_level: str          # LOW / MEDIUM / HIGH / CRITICAL
    approved: bool | None    # décision finale
    messages: list           # historique LLM


# --- Nœuds ---
# Chaque nœud est une fonction pure : reçoit l'état, retourne un update.
def load_client(state: CreditState) -> dict:
    """Nœud 1 : Charger les données client depuis la DB."""
    client_id = state['client_id']
    # En production : query SQLAlchemy
    data = {'name': 'Jean Dupont', 'income': 85000, 'history_years': 8}
    return {'client_data': data}


def calculate_score(state: CreditState) -> dict:
    """Nœud 2 : Calculer le score de crédit."""
```

```
    data = state['client_data']
    # Logique de scoring simplifiée
    score = min(850, 300 + data['income'] // 200 + data['history_years'] * 20)
    level = ('LOW' if score >= 750 else 'MEDIUM' if score >= 650
             else 'HIGH' if score >= 550 else 'CRITICAL')
    return {'credit_score': score, 'risk_level': level}


def auto_approve(state: CreditState) -> dict:
    """Nœud 3a : Approbation automatique (score élevé)."""
    return {'approved': True}


def manual_review(state: CreditState) -> dict:
    """Nœud 3b : Révision manuelle (score insuffisant)."""
    # En production : envoie une notification, attend une réponse humaine
    return {'approved': None}  # En attente


# --- Logique de routage conditionnel ---
def should_auto_approve(state: CreditState) -> str:
    """Décide la branche suivante selon le score."""
    if state['credit_score'] >= 750:
        return 'auto_approve'
    return 'manual_review'


# --- Construction du graphe ---
workflow = StateGraph(CreditState)

# Ajouter les nœuds
workflow.add_node('load_client', load_client)
workflow.add_node('calculate_score', calculate_score)
workflow.add_node('auto_approve', auto_approve)
workflow.add_node('manual_review', manual_review)

# Arêtes fixes
workflow.add_edge(START, 'load_client')
workflow.add_edge('load_client', 'calculate_score')

# Arête conditionnelle : branchement après le scoring
workflow.add_conditional_edges(
    'calculate_score',
    should_auto_approve,
    {'auto_approve': 'auto_approve', 'manual_review': 'manual_review'}
)
```

```
# Terminaisons
workflow.add_edge('auto_approve', END)
workflow.add_edge('manual_review', END)

# Compiler avec checkpointer pour persistance
checkpointer = SqliteSaver.from_conn_string(':memory:')
app = workflow.compile(checkpointer=checkpointer)

# Exécution
result = app.invoke(
    {'client_id': 42, 'messages': []},
    config={'configurable': {'thread_id': 'credit-42'}}
)
print(f'Score: {result["credit_score"]}, Approuvé: {result["approved"]}')
```

4.4 Quand Utiliser LangGraph vs PydanticAI

Critère	PydanticAI	LangGraph
Complexité du workflow	Simple à moyen (agent + tools)	Complexe (multi-branches, boucles)
Contrôle du flux	Automatique (boucle agentique)	Manuel (chaque transition définie)
Multi-agents	Possible mais basique	Natif (sous-graphes, swarms)
Human-in-the-Loop	Via prepare callback	Natif (interrupt + resume)
Persistance	Durable execution (recent)	Checkpointer (mature)
Observabilité	Logfire (natif)	LangSmith (natif)
Cas d'usage bancaire	Un agent de scoring	Pipeline scoring multi-étapes avec approbations

5. n8n — Automatisation de Workflows

5.1 Positionnement

n8n est fondamentalement différent des autres outils de cette revue. Ce n'est pas un SDK programmatique — c'est une plateforme d'automatisation de workflows avec une interface visuelle, qui a ajouté des capacités IA natives. n8n n'a pas de SDK Python ou Java au sens classique.

Ce qui rend n8n pertinent pour Tekkod : il permet d'automatiser les processus métier des clients sans écrire de code custom. Un client bancaire qui veut automatiser l'envoi de notifications

quand un scoring change, ou router les demandes de crédit vers le bon analyste, peut le faire en n8n sans mobiliser un développeur.

5.2 Capacités Programmatiques

Capacité	Détail
REST API	API publique pour créer, exécuter, modifier des workflows programmatiquement
Code Node	Nœud spécial pour écrire du JavaScript ou Python (via Pyodide/WASM) dans un workflow
Custom Nodes	Possibilité de développer des nœuds custom en TypeScript (npm packages)
Webhooks	Exposer des endpoints HTTP pour déclencher des workflows
AI Nodes (LangChain)	Nœuds natifs basés sur LangChain pour agents IA dans les workflows
Self-hosting	Docker, Kubernetes, contrôle total des données

5.3 Exemple Bancaire — Workflow d'Alerte

Un workflow n8n typique pour le scoring bancaire, décrit en pseudo-structure JSON (pas un SDK programmatique, mais la manière dont n8n modélise les workflows) :

```
// Structure conceptuelle d'un workflow n8n
// (créé visuellement ou via l'API REST)
{
  "name": "Credit Score Alert Pipeline",
  "nodes": [
    {
      "type": "n8n-nodes-base.scheduleTrigger",
      "name": "Every 15 Minutes",
      "parameters": { "rule": { "interval": [{ "field": "minutes", "minutesInterval": 15
}] } }
    },
    {
      "type": "n8n-nodes-base.mySql",
      "name": "Check Score Changes",
      "parameters": {
        "operation": "executeQuery",
        "query": "SELECT * FROM credit_applications WHERE credit_score < 550 AND status =
'PENDING'"
      }
    },
    {
      "type": "n8n-nodes-base.if",
```

```
      "name": "Has Critical Scores?",
      "parameters": { "conditions": { "number": [{ "value1": "={{ $json.length }}",
"operation": "larger", "value2": 0 }] } }
    },
    {
      "type": "@n8n/n8n-nodes-langchain.agent",
      "name": "AI Risk Analyzer",
      "parameters": { "text": "Analyze these critical credit applications: {{ $json }}" }
    },
    {
      "type": "n8n-nodes-base.slack",
      "name": "Alert Risk Team",
      "parameters": { "channel": "#risk-alerts", "text": "{{ $json.output }}" }
    }
  ]
}
```

5.4 Position dans l'Ecosystème Tekkod

n8n est complémentaire aux frameworks programmatiques. La stratégie recommandée pour Tekkod : utiliser PydanticAI/Spring AI pour les agents de scoring complexes (logique métier, validation typée, observabilité), et n8n pour l'orchestration des processus métier autour de ces agents (notifications, routing, intégration CRM, scheduling). n8n peut appeler les agents via webhooks/REST.

6. Spring AI — L'Équivalent Java de PydanticAI

6.1 Positionnement

Spring AI est le framework officiel de l'écosystème Spring pour l'intégration IA. La version 1.0 GA a été publiée en mai 2025. C'est l'équivalent le plus direct de PydanticAI dans le monde Java, avec la même philosophie : type safety, model agnostique, structured output, tool calling, et observabilité intégrée.

La différence majeure : Spring AI s'appuie sur l'écosystème Spring (auto-configuration, injection de dépendances, Actuator, Spring Data), tandis que PydanticAI s'appuie sur Pydantic (validation, sérialisation, type hints). Les deux suivent la même logique architecturale, adaptée aux idiomes de leur langage.

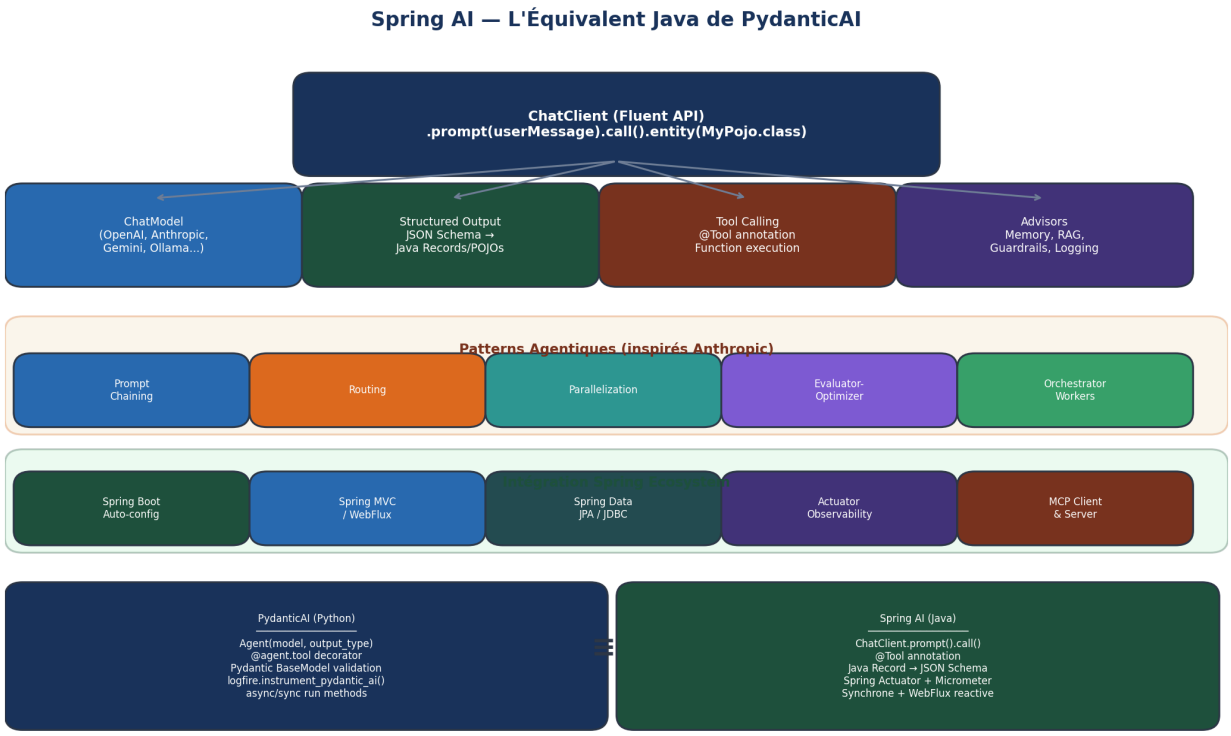

Figure 5 — Architecture Spring AI

6.2 Mapping des Concepts PydanticAI → Spring AI

Concept	PydanticAI (Python)	Spring AI (Java)
Point d'entrée	Agent(model, output_type)	ChatClient.builder(chatModel)
Structured Output	class MyOutput(BaseModel)	record MyOutput(...) via .entity()
Tools	@agent.tool decorator	@Tool annotation sur méthodes
Dépendances	RunContext[Deps] (dataclass)	Spring @Autowired / DI
Instructions	instructions= (str ou func)	.defaultSystem(prompt)
Observabilité	Logfire (instrument_pydantic_ai)	Actuator + Micrometer (auto)
Model switching	Agent('openai:gpt-4o')	spring.ai.openai.* dans properties
Retry	max_retries= sur Agent	StructuredOutputValidationAdvisor
MCP	Natif (MCPServerHTTP)	spring-ai-mcp-client/server
Patterns agentiques	Boucle agentique intégrée	Advisor chain + Recursive advisors

6.3 Exemples de Code — Contexte Bancaire

6.3.1 Agent Simple avec Structured Output

L'équivalent exact de l'agent PydanticAI de la section 2.3.1, en Java.

```
// --- Modèle de sortie (Java Record = Pydantic BaseModel) ---
// Spring AI génère automatiquement le JSON Schema depuis ce record
// et force le LLM à retourner un JSON conforme.
public record RiskAssessment(
    String clientName,
    int creditScore,         // Équivalent Field(ge=300, le=850)
    String riskLevel,         // LOW, MEDIUM, HIGH, CRITICAL
    double recommendedRate,
    boolean approved,
    String reasoning
) {}



// --- Service d'évaluation de risque ---
@Service
public class CreditRiskService {

    private final ChatClient chatClient;

    // Injection du ChatClient (configuré via application.properties)
    public CreditRiskService(ChatClient.Builder builder) {
        this.chatClient = builder
            .defaultSystem(
                "Tu es un analyste de risque crédit bancaire. "
                + "Évalue le risque du client en utilisant les outils disponibles. "
                + "Score >= 750: LOW, 650-749: MEDIUM, 550-649: HIGH, <550: CRITICAL."
            )
            .defaultTools(new CreditTools())  // Enregistrer les tools
            .build();
    }

    // L'équivalent de agent.run_sync(prompt)
    public RiskAssessment evaluateClient(int clientId) {
        return chatClient
            .prompt("Evalue le risque du client ID " + clientId + " pour un prêt
MORTGAGE")
            .call()
            .entity(RiskAssessment.class);  // Structured output vers Java Record
    }
}
```

6.3.2 Tool Calling avec @Tool

```
// --- Tools = méthodes annotées @Tool ---
```

```
// Le LLM voit le nom de la méthode et la description pour décider de l'appeler.
// Spring AI gère la sérialisation/désérialisation automatiquement.
public class CreditTools {

    @Tool(description = "Vérifie l'historique de crédit du client dans la base de
données")
    public String checkCreditHistory(int clientId) {
        // En production : JPA query sur credit_scoring_db
        return "Client " + clientId + ": score 720, historique 8 ans, revenu 85000 EUR";
    }

    @Tool(description = "Récupère le taux de base et conditions du produit de prêt")
    public String getProductRates(String productCode) {
        return "Produit " + productCode + ": base_rate 3.5%, max_amount 500000";
    }
}
```

6.3.3 Configuration (application.properties)

```
# Changer de modèle = changer cette ligne (zéro code modifié)
spring.ai.anthropic.api-key=${ANTHROPIC_API_KEY}
spring.ai.anthropic.chat.options.model=claude-sonnet-4-5

# Ou basculer vers OpenAI :
# spring.ai.openai.api-key=${OPENAI_API_KEY}
# spring.ai.openai.chat.options.model=gpt-4o

# Observabilité (auto-configuré par Spring Boot Actuator)
management.endpoints.web.exposure.include=health,metrics,info
management.tracing.enabled=true
```

6.4 Patterns Agentiques en Spring AI

Spring AI implémente les patterns agentiques décrits dans la publication d'Anthropic « Building Effective Agents ». Chaque pattern est une classe Java réutilisable :

Pattern	Description	Classe Spring AI
Prompt Chaining	Décomposer une tâche en étapes séquentielles	ChainWorkflow
Routing	Router vers un handler spécialisé selon la requête	RoutingWorkflow
Parallelization	Exécuter en parallèle puis agréger	ParallelizationWorkflow
Evaluator-Optimizer	Le modèle évalue et affine sa propre réponse	EvaluatorOptimizerWorkflow

Orchestrator Workers	Décomposition dynamique + workers spécialisés	OrchestratorWorkersWorkflow

7. LangChain4j — L'Alternative Communautaire Java

7.1 Positionnement

LangChain4j est l'adaptation Java de LangChain, développée par la communauté (pas par l'équipe LangChain officielle). Le projet a atteint la version 1.0 stable en mai 2025, avec plus de 21K étoiles GitHub. Il est soutenu par Red Hat et Microsoft.

La différence philosophique avec Spring AI : LangChain4j est framework-agnostique (fonctionne avec Spring Boot, Quarkus, ou standalone), tandis que Spring AI est natif Spring. LangChain4j adopte un pattern déclaratif inspiré de Spring Data JPA : on définit une interface, et le framework génère l'implémentation via proxy dynamique.

7.2 Le Pattern AiServices

Le cœur de LangChain4j est la classe AiServices, qui est probablement la classe la plus importante du framework. On définit une interface Java avec des annotations, et LangChain4j génère un proxy qui gère l'appel LLM, le parsing de la réponse, et le retour typé.

```
// --- Interface déclarative ---
// Le type de retour définit le structured output attendu.
// LangChain4j gère le JSON Schema, l'appel LLM, et le parsing.
interface CreditRiskAnalyst {

    @SystemMessage(
        "Tu es un analyste de risque crédit bancaire. "
        + "Évalue le risque et retourne un assessment structuré."
    )
    @UserMessage(
        "Evalue le risque du client: revenu={{income}} EUR, "
        + "historique={{historyYears}} ans, montant demande={{amount}} EUR"
    )
    RiskAssessment evaluate(
        @V("income") double income,
        @V("historyYears") int historyYears,
        @V("amount") double amount
    );
}


// --- Structured Output (même record que Spring AI) ---
```

```
record RiskAssessment(
    String clientName,
    int creditScore,
    String riskLevel,
    double recommendedRate,
    boolean approved,
    String reasoning
) {}


// --- Instanciation ---
ChatLanguageModel model = OpenAiChatModel.builder()
    .apiKey(System.getenv("OPENAI_API_KEY"))
    .modelName("gpt-4o")
    .build();

CreditRiskAnalyst analyst = AiServices.builder(CreditRiskAnalyst.class)
    .chatLanguageModel(model)
    .tools(new CreditTools())  // Même pattern que Spring AI
    .build();

// --- Utilisation ---
RiskAssessment result = analyst.evaluate(85000.0, 8, 250000.0);
System.out.println(result.riskLevel());   // "LOW"
System.out.println(result.approved());     // true
```

7.3 Orchestration Agentique (Module langchain4j-agentic)

Depuis la version 1.3.0, LangChain4j fournit un module dédié à l'orchestration d'agents : langchain4j-agentic. Ce module permet de composer des agents en workflows avec des patterns séquentiels, en boucle, conditionnels, parallèles, et le pattern superviseur (où un agent décide quels agents exécuter). Le module AgenticScope fournit le contrôle du contexte et la visibilité sur la chaîne d'appels.

7.4 Spring AI vs LangChain4j — Quel Choisir ?

Critère	Spring AI	LangChain4j
Écosystème	Natif Spring (Boot, Data, Security)	Agnostique (Spring, Quarkus, standalone)
Structured Output	Via ChatClient.entity()	Via AiServices interface return type
Tool Calling	@Tool annotation	@Tool annotation (identique)
Pattern d'agent	Advisor chain + patterns Anthropic	AiServices + langchain4j-agentic
Maturité	1.0 GA (mai 2025)	1.0 stable (mai 2025)

Communauté	Spring team (VMware/Broadcom)	Communauté open-source + Red Hat
Recommandation Tekkod	Si déjà Spring Boot	Si Quarkus ou standalone

8. Synthèse et Recommandations Tekkod

8.1 Stack Recommandée

Rôle	Python	Java
Agent Framework	PydanticAI	Spring AI
Structured Output	Pydantic BaseModel	Java Records + ChatClient.entity()
Tool Calling	@agent.tool	@Tool annotation
Observabilité	Logfire	Spring Actuator + Micrometer
Orchestration complexe	LangGraph	LangChain4j Agentic / Spring State Machine
Workflow automation	n8n (via webhooks)	n8n (via webhooks)
API Framework	FastAPI	Spring Boot
Containerisation	Docker	Docker

8.2 Progression d'Apprentissage

Semaines 1-2 : Maîtriser PydanticAI (agents, tools, structured output, Logfire) sur des cas bancaires simples. Semaine 3 : Reproduire les mêmes agents en Spring AI pour solidifier la compréhension cross-language. Semaines 4-5 : Ajouter LangGraph pour les workflows multi-étapes avec approbation humaine. Semaines 6-7 : Intégrer n8n pour l'automatisation des processus métier autour des agents. Semaines 8-10 : Projet capstone intégré (voir Document de Conception).

8.3 Références

PydanticAI : https://ai.pydantic.dev/

Pydantic Logfire : https://pydantic.dev/logfire

LangGraph : https://github.com/langchain-ai/langgraph

n8n : https://docs.n8n.io/

Spring AI : https://spring.io/projects/spring-ai

LangChain4j : https://docs.langchain4j.dev/

Anthropic — Building Effective Agents :
https://docs.anthropic.com/en/docs/build-with-claude/agentic