CreditGuard AI

Document de Conception

Agent Intelligent d'Évaluation de Risque Crédit

Implémentation Dual-Stack : Python (FastAPI + PydanticAI) | Java (Spring Boot + Spring AI)

Phase : Axe 2 — Agentic AI Shift
Programme de Formation Tekkod
Février 2026

Table des Matières

1. Vue d'Ensemble du Projet

1.1 Objectif

CreditGuard AI est un agent intelligent d'évaluation de risque crédit conçu comme projet capstone de l'Axe 2 (Agentic AI) du programme Tekkod. L'objectif est de construire un système qui reçoit une demande de crédit, interroge les données clients et produits, calcule un score de risque, et retourne une recommandation structurée — le tout orchestré par un agent IA.

Le projet est implémenté en dual-stack : Python (FastAPI + PydanticAI) et Java (Spring Boot + Spring AI). Les deux stacks partagent la même base MySQL et exposent les mêmes endpoints REST. L'objectif pédagogique est de maîtriser les deux écosystèmes agentiques sur un cas métier réaliste.

1.2 Contexte Métier

Le système modélise le processus d'évaluation de crédit bancaire : un client soumet une demande pour un produit de prêt (hypothèque, auto, personnel), l'agent récupère le profil du client, analyse l'historique de crédit, calcule un score de risque (300-850, échelle FICO), détermine le niveau de risque (LOW, MEDIUM, HIGH, CRITICAL), ajuste le taux d'intérêt en fonction du risque, et produit une recommandation d'approbation ou de rejet.

1.3 Lien avec l'Axe 4 (ORM / Credit Scoring)

Ce projet réutilise directement le schéma de données du credit scoring développé dans l'Axe 4 (clients, loan_products, credit_applications). La différence : l'Axe 4 se concentre sur l'accès aux données via ORM et l'analyse en DataFrame ; l'Axe 2 ajoute une couche d'intelligence agentique par-dessus.

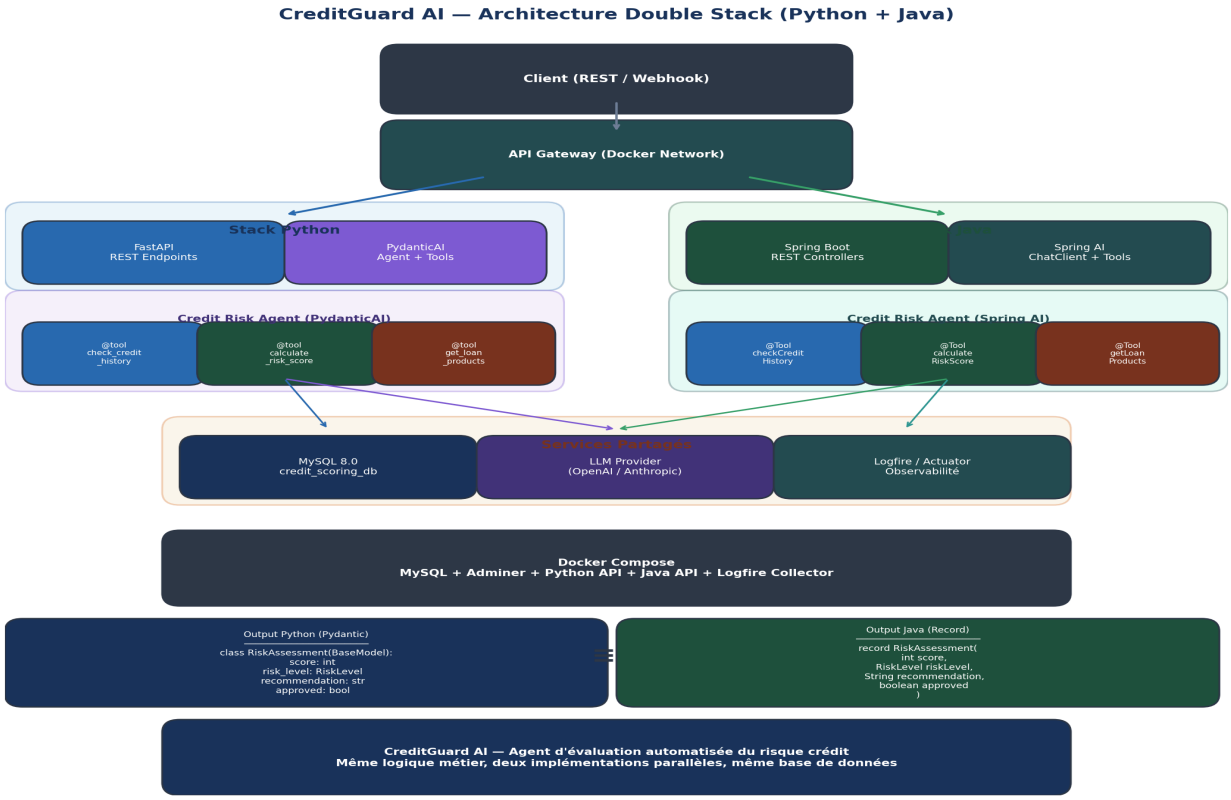

Figure 1 — Architecture CreditGuard AI — Dual-Stack

2. Architecture Technique

2.1 Composants

Composant	Technologie	Rôle
Python API	FastAPI + PydanticAI	API REST + agent de scoring Python

Java API	Spring Boot + Spring AI	API REST + agent de scoring Java
Base de données	MySQL 8.0	Stockage partagé (clients, produits, demandes, évaluations)
Admin DB	Adminer	Interface web d'administration MySQL
LLM Provider	Anthropic / OpenAI (configurable)	Moteur de raisonnement de l'agent
Observabilité Python	Pydantic Logfire	Traces, métriques, coûts tokens
Observabilité Java	Spring Actuator + Micrometer	Health, metrics, tracing

2.2 Schéma de Base de Données

Le schéma reprend et étend celui de l'Axe 4. Quatre tables principales, plus une table de résultats d'agent :

Table	Description	Clés
customers	Profils clients (revenu, emploi, historique)	PK: id
loan_products	Produits de prêt (taux base, plafond, durée)	PK: id, UNIQUE: code
credit_history	Historique de crédit par client (prêts passés, défauts)	PK: id, FK: customer_id
loan_applications	Demandes de crédit en cours d'évaluation	PK: id, FK: customer_id, product_id
risk_assessments	Résultats des évaluations par l'agent IA	PK: id, FK: application_id

Table risk_assessments (nouveau par rapport à l'Axe 4)

Cette table capture les résultats structurés retournés par l'agent IA : le score calculé, le niveau de risque, le taux recommandé, la décision (approved/rejected/review), le raisonnement textuel de l'agent, le provider LLM utilisé, le nombre de tokens consommés, et le timestamp. C'est la table d'audit qui permet de tracer chaque décision de l'agent.

3. Conception de l'Agent de Scoring

3.1 Structured Output

Le coeur de l'agent est le modèle de sortie structuré. Les deux stacks partagent exactement le même schéma :

Champ	Type	Contraintes	Description
client_name	string	Non vide	Nom complet du client
credit_score	integer	300 - 850	Score calculé (échelle FICO)
risk_level	enum	LOW, MEDIUM, HIGH, CRITICAL	Niveau de risque déduit du score
recommended_rate	float	> 0	Taux d'intérêt recommandé (base + prime)
approved	boolean	-	Décision d'approbation
reasoning	string	Non vide	Explication de la décision par l'agent

3.1.1 Implémentation Python (Pydantic BaseModel)

```
class RiskAssessment(BaseModel):
    client_name: str = Field(description="Nom complet du client")
    credit_score: int = Field(ge=300, le=850, description="Score FICO")
    risk_level: Literal['LOW', 'MEDIUM', 'HIGH', 'CRITICAL']
    recommended_rate: float = Field(gt=0, description="Taux en %")
    approved: bool = Field(description="Decision d'approbation")
    reasoning: str = Field(description="Explication de la decision")
```

3.1.2 Implémentation Java (Record)

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

3.2 Tools de l'Agent

L'agent dispose de trois tools pour accéder aux données bancaires. Le LLM décide seul lesquels appeler et dans quel ordre :

Tool	Arguments	Retourne	Description
check_credit_history	customer_id: int	Profil client + historique	Charge le profil du client depuis la base, incluant revenu, emploi, historique de crédit, et prêts passés

calculate_risk_score	customer_id: int,amount: float	Score numérique + facteurs	Calcule le score de risque basé sur les données client et le montant demandé
get_loan_products	product_code: str	Détails du produit	Récupère les conditions du produit de prêt (taux base, plafond, durée)

3.3 Instructions Système

Les instructions de l'agent définissent son rôle, ses règles de décision, et ses contraintes métier. Le même prompt est utilisé dans les deux stacks :

```
# Instructions système de l'agent CreditGuard
Tu es un analyste de risque credit bancaire. Ton role est d'evaluer
les demandes de credit en utilisant les outils disponibles.

Regles de scoring :
- Score >= 750 : risque LOW, approbation automatique
- Score 650-749 : risque MEDIUM, approbation avec taux ajuste
- Score 550-649 : risque HIGH, revision manuelle recommandee
- Score < 550 : risque CRITICAL, rejet recommande

Calcul du taux recommande :
- Taux = base_rate du produit + prime de risque
- Prime : LOW = 0%, MEDIUM = 1.5%, HIGH = 3%, CRITICAL = 5%

Toujours appeler check_credit_history en premier, puis get_loan_products,
puis calculate_risk_score. Explique ton raisonnement dans le champ reasoning.
```

3.4 Boucle Agentique

Le flux d'exécution de l'agent suit un cycle standard :

1. Réception de la demande (client ID + product code + montant). 2. Le LLM reçoit les instructions système et le message utilisateur. 3. Le LLM décide d'appeler check_credit_history (tool call). 4. Le tool exécute la query SQL et retourne les données client. 5. Le LLM appelle get_loan_products pour les conditions du produit. 6. Le LLM appelle calculate_risk_score avec les données collectées. 7. Le LLM synthétise et retourne un RiskAssessment structuré. 8. PydanticAI/Spring AI valide la sortie et retourne le résultat typé.

4. Endpoints REST

Endpoint	Méthode	Description	Réponse
/api/assess-credit	POST	Soumettre une demande d'évaluation de risque	RiskAssessment (JSON structuré)
/api/risk-history/{customer_id}	GET	Historique des évaluations d'un client	Liste de RiskAssessment
/api/loan-recommendation	POST	Recommandation de produit de prêt	Recommandation structurée
/api/health	GET	Vérification de santé de l'API	Statut + version

4.1 Endpoint Principal : POST /api/assess-credit

Request Body :

```
{
  "customer_id": 42,
  "product_code": "MORTGAGE",
  "requested_amount": 250000.00
}
```

Response (200 OK) :

```
{
  "client_name": "Jean Dupont",
  "credit_score": 720,
  "risk_level": "MEDIUM",
  "recommended_rate": 5.0,
  "approved": true,
  "reasoning": "Le client presente un profil de risque moyen. Revenu annuel
    de 85 000 EUR avec 8 ans d'historique de credit stable. Le score de 720
    est dans la tranche MEDIUM, ce qui justifie un taux de base de 3.5% plus
    une prime de 1.5%. Approbation recommandee sous conditions."
}
```

5. Human-in-the-Loop

Le système implémente un pattern d'approbation humaine pour les cas borderline. Les règles de routage sont les suivantes :

Score	Risque	Action
>= 750	LOW	Approbation automatique par l'agent

650 - 749	MEDIUM	Approbation par l'agent, notification à l'analyste
550 - 649	HIGH	Agent propose, analyste humain approuve/rejette
< 550	CRITICAL	Rejet automatique, notification au responsable

5.1 Implémentation Python (PydanticAI)

En PydanticAI, le Human-in-the-Loop se fait via le callback prepare des tools. Avant d'exécuter un tool sensible (comme l'approbation finale), le callback peut interrompre l'exécution et demander une validation humaine. En production, cette validation passerait par un webhook Slack ou une interface web.

5.2 Implémentation Java (Spring AI)

Spring AI utilise le pattern Advisor pour intercepter la chaîne d'exécution. Un ReviewAdvisor custom peut bloquer l'appel et router vers une file d'attente de révision lorsque le score est dans la zone borderline (550-749). L'analyste humain consulte la proposition de l'agent et valide ou modifie la décision.

6. Environnement Docker

6.1 Services

Service	Image	Port	Rôle
mysql	mysql:8.0	3306	Base partagée credit_scoring_db
adminer	adminer:latest	8080	Administration web MySQL
python-api	Build local (Dockerfile.python)	8000	FastAPI + PydanticAI
java-api	Build local (Dockerfile.java)	8081	Spring Boot + Spring AI

6.2 Réseau

Tous les services partagent un réseau Docker bridge. Les deux APIs accèdent à MySQL via le hostname 'mysql' (port 3306 interne). Les APIs sont exposées sur des ports différents (8000 pour Python, 8081 pour Java) pour permettre la comparaison côte à côte.

6.3 Volumes

Le volume mysql_data persiste les données entre les redémarrages. Les volumes de code (./python:/app pour Python, ./java:/app pour Java) permettent le hot-reload pendant le développement. Un volume partagé ./data/seed.sql contient les données de test initiales.

7. Observabilité

7.1 Stack Python : Logfire

Logfire trace l'intégralité de la chaîne d'exécution : l'appel HTTP entrant (FastAPI), le prompt envoyé au LLM, chaque tool call (avec durée et résultat), la réponse LLM, la validation Pydantic, et les queries SQL (SQLAlchemy). Le dashboard unifié permet de diagnostiquer les problèmes en un coup d'oeil : un tool qui met 3 secondes à répondre, un LLM qui retourne un JSON invalide, ou un coût qui explose.

7.2 Stack Java : Spring Actuator

Spring Actuator expose automatiquement les endpoints de monitoring : /actuator/health (santé de l'application et de MySQL), /actuator/metrics (temps de réponse, throughput, mémoire), et /actuator/info (version, build). L'intégration Micrometer permet d'ajouter des métriques custom pour le scoring (nombre d'évaluations, distribution des scores, taux d'approbation).

7.3 Métriques Clés à Suivre

Métrique	Type	Seuil d'Alerte
Temps de réponse moyen	Latence	> 5 secondes
Coût tokens / évaluation	Coût	> 0.10 USD / call
Taux d'erreur de validation	Qualité	> 5% de retries
Taux d'approbation	Métier	< 30% ou > 90% (à investiguer)
Distribution des scores	Métier	Dérive vs baseline

8. Tests et Évaluation

8.1 Golden Dataset

Un jeu de données de référence (golden dataset) est construit avec des cas connus : des clients au score élevé qui doivent être approuvés, des clients à score bas qui doivent être rejetés, des cas borderline, et des anomalies (haut revenu / score bas). Ce dataset sert de benchmark pour vérifier que l'agent prend les bonnes décisions de manière reproductible.

8.2 Pydantic Evals (Python)

Le framework Pydantic Evals permet de définir des cas de test, de les exécuter contre l'agent, et d'évaluer les résultats avec des critères précis : le score est-il dans la bonne tranche, le risk_level est-il cohérent avec le score, le taux recommandé respecte-t-il la formule base_rate + prime. Les résultats sont visualisés dans Logfire pour suivre la performance dans le temps.

8.3 Tests d'Intégration (Java)

Côté Java, les tests utilisent Spring Boot Test avec Testcontainers pour démarrer un MySQL isolé pendant les tests. Chaque cas du golden dataset est exécuté via l'API REST, et les assertions vérifient les mêmes critères que les Pydantic Evals. La comparaison des résultats entre les deux stacks permet de s'assurer que la logique métier est identique.

9. Structure de Projet

```
creditguard-ai/
|-- docker-compose.yml
|-- data/
|   |-- seed.sql                    # Donnees de test initiales
|   +-- golden_dataset.json          # Cas de test de reference
|
|-- python/
|   |-- Dockerfile
|   |-- pyproject.toml
|   |-- src/
|   |   |-- main.py                 # FastAPI app + endpoints
|   |   |-- agent.py                # PydanticAI Agent definition
|   |   |-- models.py               # SQLAlchemy models (from Axe 4)
|   |   |-- schemas.py              # Pydantic output schemas
|   |   |-- tools.py                # Agent tools (credit check, scoring)
|   |   |-- dependencies.py         # BankDeps dataclass
|   |   +-- observability.py        # Logfire configuration
|   +-- tests/
|       |-- test_agent.py           # Pydantic Evals
|       +-- test_api.py             # API integration tests
|
|-- java/
|   |-- Dockerfile
|   |-- pom.xml
|   |-- src/main/java/com/creditguard/
|   |   |-- CreditGuardApplication.java
|   |   |-- config/
|   |   |   +-- AiConfig.java       # ChatClient configuration
|   |   |-- model/
```

```
|   |   |   |-- Customer.java          # JPA entities (from Axe 4)
|   |   |   +-- RiskAssessment.java    # Output record
|   |   |-- tools/
|   |   |   +-- CreditTools.java       # @Tool methods
|   |   |-- service/
|   |   |   +-- CreditRiskService.java # ChatClient usage
|   |   +-- controller/
|   |       +-- AssessmentController.java
|   +-- src/test/java/com/creditguard/
|       +-- CreditGuardIntegrationTest.java
|
+-- notebooks/
    |-- exploration_python.ipynb
    +-- exploration_java.ipynb
```

10. Décisions de Conception

Décision	Choix	Justification
Agent Framework Python	PydanticAI	Type safety natif, Logfire, MCP, FastAPI-like
Agent Framework Java	Spring AI	Natif Spring Boot, ChatClient API, @Tool
LLM par défaut	Claude Sonnet (Anthropic)	Bon ratio coût/performance pour le raisonnement
Structured Output	Pydantic BaseModel / Java Record	Validation automatique des sorties agent
Base de données	MySQL 8.0 (partagée)	Continuité avec Axe 4, ENUM natif, DECIMAL précis
ORM Python	SQLAlchemy 2.0 (dans les tools)	Réutilisation des modèles Axe 4
ORM Java	JPA/Hibernate (dans les tools)	Réutilisation des entités Axe 4
Observabilité Python	Logfire	Instrumenté en 3 lignes, traces unifiées
Observabilité Java	Spring Actuator	Auto-configuré, standard Spring
API Framework Python	FastAPI	Async natif, Pydantic intégré, OpenAPI auto
API Framework Java	Spring Boot	Standard industrie, écosystème complet
Containerisation	Docker Compose	Environnement reproductible, même infra Axe 4
Human-in-the-Loop	Scores 550-749	Zone borderline nécessitant jugement humain

Tests IA	Golden dataset + Evals	Reproductibilité et suivi de performance

11. Prochaines Étapes

Ce document constitue la fondation conceptuelle du projet CreditGuard AI. L'implémentation suivra les phases définies dans le guide des frameworks pratiques : d'abord PydanticAI seul (semaines 1-2), puis Spring AI (semaine 3), puis l'intégration Docker Compose complète avec les deux stacks en parallèle.

Le passage de la conception à l'implémentation se fera en phase de pair programming, avec des code chunks explicités à la demande.