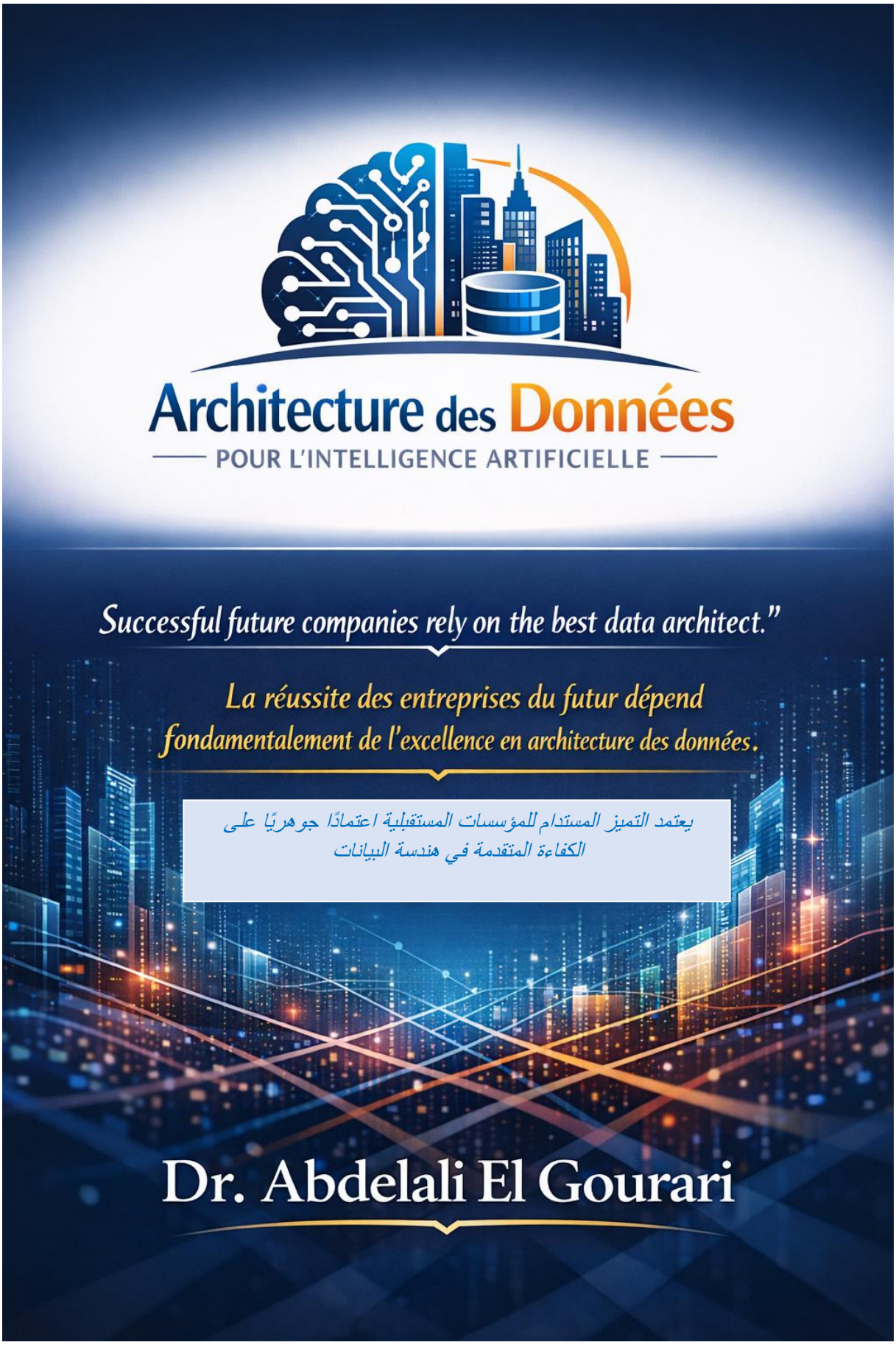


Plan

Architecture des Données pour l'Intelligence Artificielle

1. Informations Générales

- **Pré-requis :**
  - ✓ Bases en programmation Python
  - ✓ Notions de bases de données relationnelles
  - ✓ Concepts fondamentaux en Machine Learning

2. Objectifs Pédagogiques Généraux

À l'issue de ce cours, l'étudiant sera capable de :

1. Concevoir une architecture de données adaptée aux systèmes d'IA modernes.
2. Implémenter des pipelines ETL/ELT robustes.
3. Déployer une architecture RAG (Retrieval Augmented Generation).
4. Préparer des datasets pour le fine-tuning de modèles.
5. Mettre en place des pipelines Big Data et temps réel.
6. Assurer la gouvernance, la qualité et la conformité des données.
7. Concevoir et déployer un système IA complet en production.

3. Organisation du Cours par Modules

MODULE 1 : Bootcamp Python Data & Setup

Objectifs spécifiques

- Mettre en place un environnement professionnel.
- Manipuler des données avec Python.
- Comprendre les bases de la conteneurisation.

1.1 Environnement professionnel

- Installation de VS Code
- Git (versioning)
- Docker (conteneurisation)
- Conda (gestion d'environnements)

1.2 Fondamentaux Python

- Types de base : int, str, list, dict
- Manipulation de structures de données

1.3 Premiers pas avec Pandas

- Création et exploration d'un DataFrame
- Nettoyage et exportation des données

1.4 Introduction à Polars

- Performance et comparaison avec Pandas

1.5 Versioning avec Git

- Création de commits
- Navigation dans l'historique

1.6 Conteneurisation avec Docker

- Création d'une image
- Exécution d'un conteneur

Livrable : *Environnement fonctionnel + premier pipeline de nettoyage de données.*

MODULE 2 : Fondamentaux — Des Données Structurées aux Données IA

Objectifs spécifiques

- Comprendre les différents types de données en IA.
- Maîtriser les architectures ETL/ELT.
- Concevoir une architecture moderne de stockage.

2.1 Typologie des Données en IA

- Données structurées (SQL, CSV, DataFrames)
- Données semi-structurées (JSON, XML, logs)
- Données non structurées (PDF, images, texte libre)
- Données vectorielles (embeddings, recherche sémantique)

2.2 ETL vs ELT — Architectures de Pipeline

- ETL traditionnel (transformation en Python)
- ELT moderne (transformation en SQL)
- Critères de choix et compromis
- Hybridation ETL/ELT pour l'IA

2.3 Stockage Moderne : Data Lake → Lakehouse

- Data Lake (stockage objet type S3)
- Limites des architectures classiques
- Lakehouse : Delta Lake, Iceberg, Hudi

- Architecture Bronze – Silver – Gold

2.4 Préparation des Données pour l'IA

- Feature engineering
- Format de fine-tuning (instruction-input-output)
- Validation qualité et traçabilité

Livrable : *Pipeline complet de transformation vers un dataset ML exploitable.*

MODULE 3 : Architecture RAG — Retrieval Augmented Generation

Objectifs spécifiques

- Comprendre les limites des LLMs seuls.
- Implémenter une architecture RAG complète.
- Évaluer les performances d'un système RAG.

3.1 Fondamentaux du RAG

- Limites des LLMs
- Architecture haut niveau
- Chaîne : ingestion → embedding → retrieval → génération

3.2 Préparation des Documents

- Parsing (PDF, Word, HTML)
- Stratégies de chunking (fixe, récursif, sémantique)
- Métadonnées et contextualisation

3.3 Bases de Données Vectorielles

- Recherche vectorielle (similarité cosinus)
- Comparaison : Chroma, Qdrant, Weaviate, Pinecone, pgvector
- Recherche hybride (vectorielle + BM25)
- Reranking

3.4 Implémentation Complète

- Pipeline d'ingestion
- API de recherche (FastAPI)
- Intégration LLM (OpenAI / Anthropic)
- Évaluation : hit rate, MRR

Livrable : *Système RAG complet (ingestion + API + interface).*

MODULE 4 : Fine-tuning et Feature Stores

Objectifs spécifiques

- Choisir entre RAG et fine-tuning.
- Préparer un dataset d'entraînement de qualité.
- Mettre en place un Feature Store.
- Assurer la reproductibilité des expériences.

4.1 Stratégies d'Entraînement

- RAG vs Fine-tuning
- Supervised Fine-Tuning (SFT)
- RLHF (aperçu)

4.2 Préparation des Datasets

- Formats : Alpaca, ShareGPT, ChatML
- Création d'exemples d'entraînement
- Augmentation synthétique
- Gestion du déséquilibre

4.3 Feature Stores

- Concept et architecture
- Offline vs Online features
- Implémentation avec Feast
- Intégration avec modèles LLM

4.4 MLOps Léger

- Versioning avec DVC
- Tracking avec MLflow
- Reproductibilité

Livrable : *Dataset validé + Feature Store opérationnel.*

MODULE 5 : Pipelines à Grande Échelle

Objectifs spécifiques

- Concevoir des pipelines Big Data.
- Orchestrer des workflows complexes.
- Implémenter du streaming temps réel.
- Déployer sur le cloud.

5.1 Big Data avec Spark

- Cas d'usage
- RDD vs DataFrame API
- Spark SQL
- Optimisation (partitionnement, caching)

5.2 Orchestration

- Apache Airflow (DAGs as code)
- Gestion des dépendances
- Monitoring
- Alternatives (Dagster, Prefect)

5.3 Streaming et Temps Réel

- Batch vs Streaming
- Apache Kafka
- Spark Structured Streaming
- Lambda vs Kappa

5.4 Cloud Data Platforms

- AWS (S3, Glue, Athena, Bedrock)
- GCP (BigQuery, Vertex AI)
- Azure (Synapse, OpenAI Service)
- Optimisation des coûts

Livrable : *Pipeline Spark orchestré et déployé sur le cloud.*

MODULE 6 : Gouvernance, Qualité et Sécurité

Objectifs spécifiques

- Garantir la qualité des données.
- Assurer la conformité réglementaire.
- Comprendre les architectures organisationnelles modernes.

6.1 Qualité des Données

- Dimensions : complétude, cohérence, fraîcheur
- Tests automatisés (Great Expectations)
- Data profiling
- Data lineage

6.2 Sécurité et Conformité

- RGPD : anonymisation, pseudonymisation
- Chiffrement (at rest / in transit)
- RBAC / ABAC
- Audit

6.3 Data Mesh

- Limites des architectures centralisées
- Domain-oriented ownership
- Data as a Product
- Self-serve platform

Livrable : *Framework de qualité et documentation de conformité.*

MODULE 6 : Projet Final — Système IA Complet

Objectifs : Concevoir et implémenter un système IA de bout en bout.

Phases du Projet

1. **Cahier des charges**
   - ❖ Étude de cas réel (e-commerce, santé, finance)
2. **Architecture**
   - ❖ Conception complète de l'architecture data
3. **Implémentation**
   - ❖ Pipeline : ingestion → RAG → API
4. **Déploiement**
   - ❖ Conteneurisation et mise en production
5. **Documentation**
   - ❖ Architecture Decision Records (ADRs)
6. **Présentation orale**
   - ❖ Soutenance et démonstration technique

Livrable final : *Système IA complet opérationnel et documenté.*

MODULE 1 : Bootcamp Python Data & Setup

Partie 1 : L'environnement de travail professionnel

1.1 Pourquoi ces outils ?

Outil	À quoi ça sert ?	Pourquoi c'est indispensable ?
VS Code	Éditeur de code	Gratuit, extensions Python puissantes, debugger intégré, standard industriel
Git	Versioning du code	Sauvegarde l'historique, travail en équipe, déploiement automatisé
Docker Desktop	Conteneurisation	"Ça marche sur mon PC" → "Ça marche partout". Reproductibilité totale
Conda	Gestion environnements	Isole les projets (évite conflits de versions), gère Python + librairies système

Analogie : Imaginez que vous cuisinez.

> Conda = cuisine séparée par projet (pas de mélange sucre/sel).
> Git = carnet de recettes qui mémorise chaque version.
> Docker = lunchbox qui garde le repas identique partout où vous allez.

1.2 Installation et premier pas

Étape 1 : Vérifier l'installation Python

```
# Dans VS Code, terminal (Ctrl+J), tapez : python --version
# Résultat attendu : Python 3.11.x ou 3.12.x
```

Étape 2 : Créer un environnement Conda (isolation totale)

```
# TERMINAL - Ligne 1
conda create -n cours_data_ia python=3.11
```

Explication : Crée un environnement nommé `cours_data_ia` avec Python 3.11. C'est comme créer une pièce séparée dans votre maison où vous installerez UNIQUEMENT les outils de ce cours. Si vous cassez quelque chose ici, le reste de votre ordinateur est protégé.

```
# TERMINAL - Ligne 2
conda activate cours_data_ia
```

Explication : Entre dans la pièce. Votre terminal indique maintenant `(cours_data_ia)` au début. Tout ce que vous installerez maintenant restera dans cette pièce.

Dans VS Code → Sélectionner le bon interpréteur

Dans VS Code :

- Ctrl + Shift + P
- Tapez : Python: Select Interpreter
- Choisissez : Python 3.11 (cours_data_ia)

```
# TERMINAL - Ligne 3
conda install pandas polars pyarrow jupyter

pip install pandas polars pyarrow jupyter
```

Explication : Installe 4 outils essentiels :

- **`pandas`** : manipulation de données (Excel en Python)
- **`polars`** : pandas ultra-rapide (même usage, 10x plus vite)
- **`pyarrow`** : format de données moderne (utilisé par polars et Spark)
- **`jupyter`** : notebooks interactifs (pour expérimenter)

1.3 Premier contact avec Python

Créez un fichier `test_environment.py` dans VS Code :

```
# LIGNE 1 : Importation des librairies
import pandas as pd       # 'pd' = alias conventionnel pour pandas
import polars as pl       # 'pl' = alias pour polars
import sys                # Module système de Python (infos sur l'environnement)

# LIGNE 5 : Vérification des versions (évite les bugs de compatibilité)
print("=" * 50)  # Affiche 50 fois le caractère "=" (séparateur visuel)
print("VÉRIFICATION DE L'ENVIRONNEMENT")
print("=" * 50)

# LIGNE 10 : f-string = formatage de texte moderne en Python
# {sys.version} = insère la version de Python automatiquement
print(f"Python version : {sys.version}")

# LIGNE 13 : Vérification que pandas est bien installé
# pd.__version__ = attribut interne de pandas contenant sa version
print(f"Pandas version : {pd.__version__}")

# LIGNE 16 : Vérification polars
print(f"Polars version : {pl.__version__}")

# LIGNE 19 : Message de succès
print("\n Environnement prêt ! Vous pouvez commencer le cours.")
```

Explication:

> **`import`** charge des fonctionnalités externes. **`as`** crée un raccourci. Au lieu de taper **`pandas.DataFrame`**, vous taperez **`pd.DataFrame`**

`"=" * 50` multiplie la chaîne "=" 50 fois. C'est du Python pur, pas spécifique aux données
`f"..."` = f-string (format string). Permet d'insérer des variables directement dans le texte avec `{}`
`__version__` = convention Python pour les métadonnées internes d'un package

Pour exécuter : `python test_environment.py`

Partie 2 : Les structures de données fondamentales

2.1 Rappel : Les types Python de base

Avant de manipuler des données massives, maîtrisez ces 4 types :

```
# Type 1 - NOMBRE (integer)
age = 29
# 'age' est le nom de la boîte, 29 est la valeur entière (int)

# Type 2 - TEXTE (string, toujours entre guillemets)
nom = "Abdelali El Gourari"
# Les guillemets délimitent le texte. "29" (texte) ≠ 29 (nombre)

# Type 3 - LISTE (collection ordonnée, modifiable)
notes = [15, 18, 12, 14]
# Crochets [] = liste. Index commence à 0 : notes[0] = 15, notes[1] = 18

# Type 4 - DICTIONNAIRE (clé-valeur, comme un formulaire)
etudiant = {
    "nom": "Ali Mouhou",        # Clé "nom", valeur "Ali Mouhou"
    "age": 25,                  # Clé "age", valeur 25
    "notes": [15, 18, 12, 14]   # Clé "notes", valeur = une liste
}
# Accès : etudiant["nom"] retourne "Ali Mouhou"

# Vérification
print(f"Type de 'age' : {type(age)}")           # <class 'int'>
print(f"Type de 'nom' : {type(nom)}")           # <class 'str'>
print(f"Type de 'notes' : {type(notes)}")       # <class 'list'>
print(f"Type de 'etudiant' : {type(etudiant)}") # <class 'dict'>

# Accès aux données
print(f"\nPremière note : {notes[0]}")          # Index 0 = premier élément
print(f"Nom de l'étudiant : {etudiant["nom"]}") # Accès par clé
```

Pourquoi c'est crucial ? : Pandas et Polars utilisent ces structures en interne. Un DataFrame n'est qu'une **liste de dictionnaires optimisée**.

2.2 Premier vrai traitement : Pandas (le classique)

Créez `premier_pipeline.py` :

```
# Importation
import pandas as pd  # On importe la librairie de manipulation de données

# CRÉATION DE DONNÉES BRUTES (simulation d'un fichier CSV)
```



```python
# Liste de dictionnaires = structure typique des données tabulaires
donnees_brutes = [
    {"client_id": 1, "nom": "Alice", "achat": 150.0, "ville": "Paris"},
    {"client_id": 2, "nom": "Bob", "achat": 0.0, "ville": "Lyon"},
                # 0 = problème
    {"client_id": 3, "nom": "Charlie", "achat": 230.5, "ville": None},
                # None = donnée manquante
    {"client_id": 4, "nom": "Diana", "achat": -50.0, "ville": "Paris"},
                # Négatif = erreur
    {"client_id": 5, "nom": "Eve", "achat": 89.0, "ville": "Marseille"},
]

# CRÉATION DU DATAFRAME (tableau de données)
# pd.DataFrame() convertit la liste de dict en tableau structuré
df = pd.DataFrame(donnees_brutes)

# EXPLORATION RAPIDE (indispensable en data)
print("=" * 50)
print("DONNÉES BRUTES")
print("=" * 50)
print(df)   # Affiche le tableau complet
print(f"\nDimensions : {df.shape}")   # shape = (nb_lignes, nb_colonnes)
# Résultat : (5, 4) = 5 clients, 4 colonnes

# DÉTECTION DES PROBLÈMES
print("\n" + "=" * 50)
print("ANALYSE DE QUALITÉ")
print("=" * 50)

# df.isnull() = tableau booléen (True si valeur manquante)
# .sum() compte les True par colonne
print(f"Valeurs manquantes :\n{df.isnull().sum()}")

# df.describe() = statistiques descriptives automatiques
print(f"\nStatistiques :\n{df.describe()}")

# NETTOYAGE DES DONNÉES (Data Cleaning)

# Étape 1 - Supprimer les achats négatifs (incohérents)
# df['achat'] sélectionne la colonne
# >= 0 crée un masque booléen (lignes à garder)
df_clean = df[df['achat'] >= 0].copy()   # .copy() évite les warnings
# Explication : On garde uniquement les lignes où achat >= 0

# Étape 2 - Remplacer les valeurs manquantes
# .fillna() = "fill NA (Not Available)"
df_clean['ville'] = df_clean['ville'].fillna("Inconnue")
# Charlie avait ville=None, maintenant ville="Inconnue"

# Étape 3 - Créer une nouvelle colonne (feature engineering)
# Calcul du montant TTC (TVA 20%)
df_clean['achat_ttc'] = df_clean['achat'] * 1.20
# '*' multiplie chaque valeur de la colonne par 1.20

# RÉSULTAT FINAL
print("\n" + "=" * 50)
print("DONNÉES NETTOYÉES")
print("=" * 50)
print(df_clean)
```



```
# EXPORT (pour le module suivant)
# Parquet = format moderne, compressé, rapide (remplace CSV)
df_clean.to_parquet("clients_nettoyes.parquet", index=False)
# index=False = ne pas sauvegarder la colonne d'index (0,1,2,3)
print("\n Fichier sauvegardé : clients_nettoyes.parquet")
```

Explications détaillées des opérations clés :

Concept	Explication
DataFrame	Structure tabulaire 2D (lignes = observations, colonnes = variables). C'est Excel en Python.
isnull()	Détecte les "trous" dans les données. Critique : les modèles IA détestent les valeurs manquantes.
Filtrage	`df[condition]` garde uniquement les lignes où condition = True. Ici, on élimine les erreurs de saisie (négatifs).
fillna()	Stratégie de "imputation" (remplissage). Alternative : supprimer la ligne, ou remplacer par la moyenne.
Feature Engineering	Création de nouvelles informations à partir des existantes. Ici, on ajoute la TVA. Essentiel pour l'IA.
Parquet	Format colonne-orienté (vs ligne pour CSV). 10x plus rapide, 5x moins volumineux. Standard Big Data.

2.3 Passer à la vitesse supérieure : Polars (le moderne)

Même traitement, **10x** plus rapide (utile pour gros volumes) :

```
# Importation
import polars as pl  # Polars = concurrent moderne de pandas, écrit en Rust

# Mêmes données brutes
donnees_brutes = [
    {"client_id": 1, "nom": "Alice", "achat": 150.0, "ville": "Paris"},
    {"client_id": 2, "nom": "Bob", "achat": 0.0, "ville": "Lyon"},
    {"client_id": 3, "nom": "Charlie", "achat": 230.5, "ville": None},
    {"client_id": 4, "nom": "Diana", "achat": -50.0, "ville": "Paris"},
    {"client_id": 5, "nom": "Eve", "achat": 89.0, "ville": "Marseille"},
]

# CRÉATION DU DATAFRAME POLARS
# pl.DataFrame = même concept, implémentation différente
df = pl.DataFrame(donnees_brutes)

# SYNTAXE POLARS - Méthodes enchaînées (fluent API)
```



```
# L'avantage : Polars optimise l'exécution automatiquement
df_clean = (
    df
    # Filtrage (même logique que pandas)
    .filter(pl.col("achat") >= 0)   # pl.col() sélectionne une colonne

    # Remplacement des nulls
    .with_columns(
        pl.col("ville").fill_null("Inconnue")
        # with_columns = ajoute/modifie colonnes
    )

        # Création de la colonne TTC
    .with_columns(
        (pl.col("achat") * 1.20).alias("achat_ttc")
        # .alias() nomme la nouvelle colonne
    )
)

# AFFICHAGE
print("Résultat avec Polars :")
print(df_clean)

# Export identique
df_clean.write_parquet("clients_nettoyes_polars.parquet")
print("\n Export Polars réussi")
```

Différences clés Pandas vs Polars :

Aspect	Pandas	Polars
Syntaxe	`df[df['col'] > 0]`	`df.filter(pl.col("col") > 0)`
Performance	1x (référence)	10-50x plus rapide
Mémoire	Copie les données souvent	"Lazy evaluation" (optimisé)
Quand l'utiliser	< 100k lignes, prototypage	> 1M lignes, production

Partie 3 : Git — Sauvegarder son travail comme un pro

3.1 Pourquoi Git est non-négociable ?

Imaginez : vous travaillez 3h sur un pipeline, vous supprimez une ligne par erreur, tout casse. **Sans Git** : vous pleurez. **Avec Git** : vous revenez en arrière en 10 secondes.

3.2 Les commandes essentielles (à taper dans le terminal)

```
# ÉTAPE 1 : Initialiser un repository (une fois par projet)
git init
```



```
# Crée un dossier caché .git qui mémorise TOUT l'historique

# ÉTAPE 2 : Voir l'état actuel
git status
# Affiche : fichiers modifiés, fichiers nouveaux, fichiers prêts à être sauvegardés

# ÉTAPE 3 : Ajouter des fichiers à la "zone de staging" (préparation)
git add test_environment.py
git add premier_pipeline.py
# Ou pour tout ajouter : git add .

# ÉTAPE 4 : Créer un "commit" (point de sauvegarde nommé)
git commit -m "Module 1 : Setup environnement et premier pipeline"
# -m = message décrivant ce que vous avez fait
# Règle d'or : message clair, au présent, impératif

# ÉTAPE 5 : Voir l'historique
git log --oneline
# Affiche : a1b2c3d Module 1 : Setup environnement et premier pipeline

# ÉTAPE 6 (si besoin) : Revenir en arrière
git checkout a1b2c3d   # Retourne au commit a1b2c3d

# Ou annuler les modifications non commitées :
git restore premier_pipeline.py
```

ÉTAPE 7 : Vérifier la branche

```
git branch
```

Normalement :

```
* main
```

ou

```
* master
```

ÉTAPE 8 : Vérifier le remote

```
git remote -v
```

Si rien ne s'affiche → vous devez connecter le dépôt distant.

Si le remote n'est pas encore configuré

1. Créez un repository vide sur GitHub (ex : architectures-donnees)
2. Puis ajoutez le remote :

```
git remote add origin https://github.com/VOTRE_USER/architectures-donnees.git
```

ÉTAPE 9 : Push vers GitHub

Si c'est la première fois :

```
git push -u origin main
```

ou si votre branche est master :

```
git push -u origin master
```

➔ -u permet de lier la branche locale au remote (une seule fois).

Si Git demande authentification

GitHub n'accepte plus mot de passe simple.
Il faut :

- Soit utiliser GitHub Desktop

- Soit utiliser un **Personal Access Token**
- Soit configurer SSH

Vérification finale
Allez sur GitHub → vous devez voir vos fichiers :

- test_environment.py
- premier_pipeline.py

Workflow normal ensuite
Après modification :

```
git add .
git commit -m "Ajout preprocessing"
git push
```

➔ (plus besoin de -u)

Analogie : Git est comme un **système de sauvegarde dans un jeu vidéo**. Vous créez des points de sauvegarde (commits) avant les boss difficiles (changements risqués). Vous pouvez recharger n'importe quelle sauvegarde.

Partie 4 : Docker — L'environnement reproductible

4.1 Le problème résolu par Docker

"Mais ça marchait sur mon ordinateur !" — Phrase la plus entendue en Data.

*Docker encapsule votre application + ses dépendances dans une **boîte isolée** qui fonctionne identiquement sur tous les ordinateurs.*

4.2 Premier Dockerfile (fichier de configuration)

Créez un fichier nommé `Dockerfile` (sans extension) :

```
# Image de base = système d'exploitation minimal avec Python
FROM python:3.11-slim
# 'FROM' = on part de l'image officielle Python 3.11 (version légère 'slim')

# Définition du répertoire de travail dans le conteneur
WORKDIR /app
# Toutes les commandes suivantes s'exécuteront dans /app

# Copie du fichier de dépendances (on le crée juste après)
COPY requirements.txt .
# 'COPY' copie depuis votre PC vers le conteneur
# Le point '.' = répertoire actuel (/app)

# Installation des librairies Python
RUN pip install --no-cache-dir -r requirements.txt
# 'RUN' exécute une commande lors du BUILD de l'image
# --no-cache-dir = évite de stocker les fichiers temporaires (allège l'image)

# Copie de tout votre code source
COPY . .
```



```
# Copie tous les fichiers du dossier courant vers /app

# Commande par défaut lors du démarrage du conteneur
CMD ["python", "premier_pipeline.py"]
# 'CMD' = ce qui s'exécute quand vous lancez le conteneur
```

Créez `requirements.txt` (liste des dépendances) :

```
pandas==2.1.4
polars==0.20.3
pyarrow==14.0.1
```

4.3 Construction et exécution

```
# ÉTAPE 1 : Construire l'image (créer la boîte)
docker build -t mon-premier-pipeline:v1 .
# -t = tag (nom de l'image)
# . = utilise le Dockerfile dans le répertoire courant

# ÉTAPE 2 : Vérifier que l'image existe
docker images
# Affiche : mon-premier-pipeline | v1 | 850MB | il y a 2 minutes

# ÉTAPE 3 : Exécuter le conteneur (lancer la boîte)
docker run mon-premier-pipeline:v1
# Vous voyez le résultat de votre script Python s'afficher !

# ÉTAPE 4 : Exécuter avec un volume partagé (pour récupérer les fichiers générés)
docker run -v $(pwd)/output:/app/output mon-premier-pipeline:v1
# -v = monte un dossier local (output) dans le conteneur (/app/output)
# Les fichiers créés dans /app/output seront visibles dans ./output sur votre PC
```

Exercice d'évaluation du Module 1

Mission : Créer un pipeline complet qui :

1) **Lit** un fichier CSV `ventes.csv`
2) **Nettoie** : supprime les prix négatifs, remplit les catégories manquantes par "Non classé"
3) **Transforme** : calcule le prix total (quantité × prix unitaire) et applique une remise de 10% si quantité > 10
4) **Exporte** en Parquet compressé
5) **Versionne** avec Git (minimum 3 commits clairs)
6) **Containerise** avec Docker (l'image doit s'exécuter sans erreur)

Critères de réussite :

- Code Python exécutable sans erreur
- Commentaires expliquant chaque étape
- Historique Git propre (`git log` montre la progression)
- `docker run` produit le fichier Parquet attendu

Récapitulatif des concepts acquis

Concept	Définition simple	Pourquoi c'est important pour l'IA
Environnement Conda	Pièce isolée pour chaque projet	Évite les conflits entre versions de librairies ML
DataFrame	Tableau Excel programmable	Structure universelle pour préparer les données d'entraînement
Parquet	Format de fichier compressé, rapide	Réduit les coûts de stockage cloud, accélère le chargement des datasets
Git	Historique des modifications	Collaboration en équipe, reproductibilité des expériences ML
Docker	Boîte isolée reproductible	Déployer un modèle IA sur un serveur distant sans surprise

MODULE 2 : Fondamentaux — Des Données Structurées aux Données IA

Partie 1 : Les 4 Types de Données en IA Moderne

1.1 Pourquoi cette classification est cruciale ?

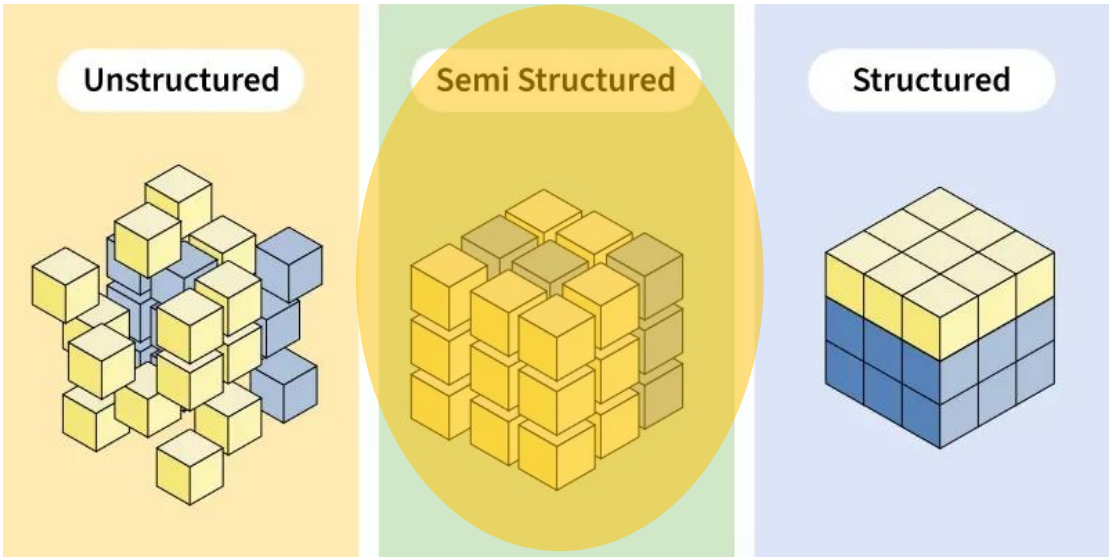
Dans l'IA moderne (LLMs, RAG, Fine-tuning), chaque type de donnée demande un traitement spécifique. ***Comprendre ces types*** = ***choisir les bonnes architectures***.

```
# Importations pour gérer tous les types de données
import pandas as pd             # Données structurées (tableaux)
import json                     # Données semi-structurées (JSON)
from PIL import Image           # Données non-structurées (images)
import numpy as np              # Calculs numériques (vecteurs)

# ============================================================
# TYPE 1 : DONNÉES STRUCTURÉES (Structured Data)
# ============================================================
```

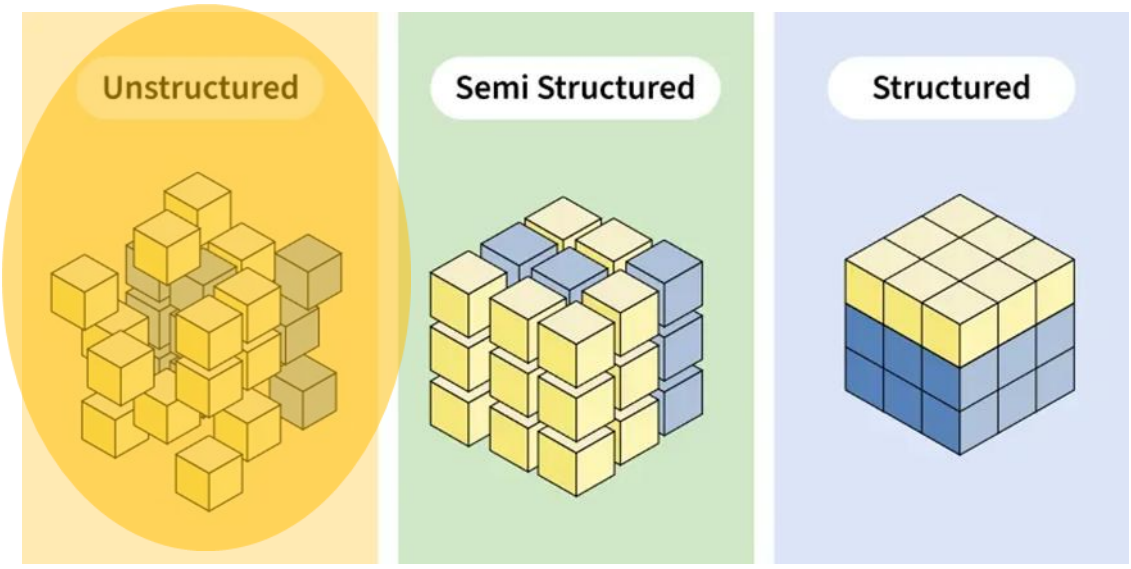

```
# Définition : Format rigide, schéma fixe, lignes et colonnes

# Exemples : Bases SQL, CSV Excel, tables de faits

# Usage IA : Features numériques, métadonnées de documents, scores

# Création d'une table client (comme en SQL)
client_structured = {
    "client_id": [1, 2, 3],             # Liste d'entiers (ID unique)
    "nom": ["Alice", "Bob", "Charlie"], # Liste de textes
    "age": [25, 30, 35],                # Liste d'entiers
    "ville": ["Paris", "Lyon", "Marseille"],
    "score_credit": [750, 680, 820]     # Feature pour ML
}

# Conversion en DataFrame Pandas (structure tabulaire)
df_clients = pd.DataFrame(client_structured)

# Exploration rapide
print("=" * 60)
print("TYPE 1 : DONNÉES STRUCTURÉES")
print("=" * 60)
print(df_clients)
print(f"\nSchéma (dtypes) :\n{df_clients.dtypes}")
# dtypes = data types : int64 (nombre), object (texte)

# ============================================================
# TYPE 2 : DONNÉES SEMI-STRUCTURÉES (Semi-structured)
```



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




```
# Définition : Schéma flexible, hiérarchique, sans structure fixe
# Exemples : JSON, XML, logs d'application, NoSQL
# Usage IA : Contexte RAG, traces de conversation, métadonnées riches

# Log d'application (typique des systèmes modernes)
log_semi_structured = {
    "timestamp": "2024-01-15T14:30:00Z",   # Format ISO 8601 standard
    "user_id": 1254,
    "action": "click",
    "page": "/produit/iphone17",
    "metadata": {                # OBJET IMBRIQUÉ (nested)
        "device": "mobile",
        "browser": "Chrome",
        "location": {
            "country": "Maroc",
            "city": "Marrakech"
        }
    },
    "tags": ["electronics", "apple", "high-value"]   # LISTE dans dict
}

# Sérialisation JSON (format universel d'échange)
json_string = json.dumps(log_semi_structured, indent=2, ensure_ascii=False)
# dumps = dump string : convertit dict Python → texte JSON
# indent=2 : formatage lisible (2 espaces d'indentation)
# ensure_ascii=False : autorise les accents (é, è)

print("\n" + "=" * 60)
print("TYPE 2 : DONNÉES SEMI-STRUCTURÉES (JSON)")
print("=" * 60)
print(json_string)

# Accès aux données hiérarchiques
print(f"\nAccès profond :
{log_semi_structured['metadata']['location']['city']}")
# On descend dans l'arbre : metadata → location → city
# Résultat : "Marrakech"

# ============================================================
# TYPE 3 : DONNÉES NON-STRUCTURÉES (Unstructured)
# ============================================================
```


```
# Définition : Pas de structure prédéfinie, contenu brut
# Exemples : PDF, images, vidéos, audio, texte libre
# Usage IA : Corpus d'entraînement LLM, documents pour RAG

# Simulation d'un document PDF (texte brut)
document_texte = """
CONTRAT DE PRESTATION DE SERVICES

ENTRE :
La Société TECH SOLUTIONS, SAS au capital de 50 000€
Ci-après dénommée "LE PRESTATAIRE"

ET :
M. Martin Dupont, entrepreneur individuel
Ci-après dénommé "LE CLIENT"

ARTICLE 1 : OBJET DU CONTRAT
LE PRESTATAIRE s'engage à réaliser pour LE CLIENT
un audit de sécurité informatique selon les normes ISO 27001.

ARTICLE 2 : DURÉE
Le présent contrat est conclu pour une durée de 12 mois
à compter du 1er janvier 2024.
"""

# Traitement basique du texte (préparation pour NLP)
mots = document_texte.lower().split()  # Minuscules + découpage
# lower() : normalise (évite "Contrat" ≠ "contrat")
# split() : découpe par espaces (liste de mots)

print("\n" + "=" * 60)
print("TYPE 3 : DONNÉES NON-STRUCTURÉES (Texte)")
print("=" * 60)
print(f"Extrait : {document_texte[:100]}...")  # 100 premiers caractères
print(f"Nombre de mots : {len(mots)}")

# Simulation image (on ne charge pas de vrai fichier pour l'exemple)
# Dans la vraie vie : img = Image.open("photo.jpg")
print(f"\n[Image simulée : 1920x1080 pixels, 3 canaux RGB]")

============================================================
# TYPE 4 : DONNÉES VECTORIELLES (Embeddings)
```



```
# ============================================================
# Définition : Représentation mathématique dense du sens (sémantique)
# Format : Vecteur de N nombres (768, 1024, 1536 dimensions typiquement)
# Usage IA : Recherche sémantique RAG, similarité entre documents

# Simulation d'un embedding (1536 dimensions = OpenAI)
# Dans la vraie vie : généré par un modèle (sentence-transformers, OpenAI)
embedding_document = np.random.randn(1536)   # 1536 nombres aléatoires
# random.randn = distribution normale (moyenne 0, écart-type 1)
# C'est la forme standard des embeddings pré-entraînés

print("\n" + "=" * 60)
print("TYPE 4 : DONNÉES VECTORIELLES (Embeddings)")
print("=" * 60)
print(f"Dimensions : {embedding_document.shape}")   # (1536,) = vecteur 1D
print(f"Premières valeurs : {embedding_document[:5]}")   # 5 premiers nombres
print(f"Norme (taille) : {np.linalg.norm(embedding_document):.2f}")
# Norme = "longueur" du vecteur (toujours ~1 pour embeddings normalisés)

# Calcul de similarité (cosine similarity)
# Deux documents similaires = vecteurs proches dans l'espace
embedding_doc2 = np.random.randn(1536)
similarity = np.dot(embedding_document, embedding_doc2) / (
    np.linalg.norm(embedding_document) * np.linalg.norm(embedding_doc2)
)
# dot product = produit scalaire (mesure alignement)
# Divisé par les normes = normalisation (résultat entre -1 et 1)

print(f"Similarité cosinus : {similarity:.3f}")
# 1.0 = identique, 0.0 = orthogonal, -1.0 = opposé
```

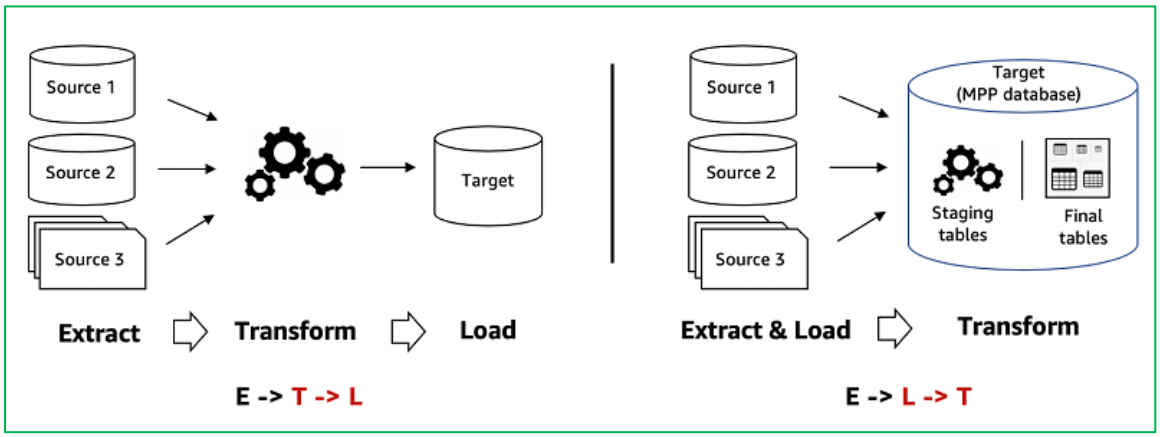
Tableau récapitulatif des types de données :

Type	Structure	Stockage typique	Usage IA principal	Outils Python
Structuré	Table (lignes/colonnes)	PostgreSQL, Snowflake, CSV	Features ML, métadonnées	Pandas, SQLAlchemy
Semi-structuré	Hiérarchique (clé-valeur)	MongoDB, S3 (JSON), Elasticsearch	Contexte RAG, logs	JSON, PyMongo
Non-structuré	Brut (binaire/texte libre)	S3, Data Lake (PDF, images)	Corpus entraînement, retrieval	Pillow, PyPDF, Unstructured
Vectoriel	Vecteur mathématique (N-dim)	Qdrant, Weaviate, pgvector	Recherche sémantique	NumPy, sentence-transformers

Partie 2 : ETL vs ELT — Architecture des Pipelines

2.1 Pourquoi deux approches ?

Le choix ETL vs ELT détermine la vitesse, le coût et la flexibilité de vos pipelines IA.




```
# ============================================================
# SCÉNARIO : Préparation de données clients pour un modèle IA
# Source : Fichiers CSV bruts (simulation SaaS, ERP, Web)
# ============================================================

import pandas as pd
import hashlib  # Pour anonymisation (RGPD)
from datetime import datetime

# Données brutes simulées (3 sources différentes)
# Problèmes typiques : doublons, formats incohérents, données sensibles
data_source_crm = [
    {"id": 1, "email": "alice@email.com", "revenu": "45000€",
"date_inscription": "2023-01-15"},
    {"id": 2, "email": "bob@email.com", "revenu": "32000€",
"date_inscription": "2023-03-22"},
    {"id": 3, "email": "ALICE@EMAIL.COM", "revenu": "45000€",
"date_inscription": "2023-01-15"},  # Doublon
]

data_source_web = [
    {"user_id": 1, "pages_vues": 45, "temps_session_sec": 1200,
"conversion": "oui"},
    {"user_id": 2, "pages_vues": 12, "temps_session_sec": 180, "conversion":
"non"},
    {"user_id": 99, "pages_vues": 999, "temps_session_sec": 10,
"conversion": "oui"},  # ID inexistant
]

# ============================================================
# APPROCHE 1 : ETL (Extract-Transform-Load)
# ============================================================
# Logique : Transformer AVANT de stocker
# Quand l'utiliser : Données sensibles, qualité critique, schéma fixe

print("=" * 70)
print("APPROCHE ETL : Transformation avant stockage")
print("=" * 70)

def pipeline_etl(crm_data, web_data):
```

```
    """
    Pipeline ETL complet avec transformation en Python
    """

    # EXTRACT - Chargement des sources
    df_crm = pd.DataFrame(crm_data)
    df_web = pd.DataFrame(web_data)
    print(f"1. EXTRACT : {len(df_crm)} lignes CRM, {len(df_web)} lignes
Web")

    # TRANSFORM - Nettoyage et enrichissement (tout en Python)

    # Normalisation email (minuscules pour dédoublonnage)
    df_crm['email_clean'] = df_crm['email'].str.lower().str.strip()
    # str.lower() : minuscules | str.strip() : enlève espaces

    # Anonymisation RGPD (hash irréversible)
    def hash_email(email):
        """Transforme l'email en identifiant anonyme"""
        return hashlib.md5(email.encode()).hexdigest()[:8]
        # md5 = algorithme de hash | hexdigest = sortie texte
        # [:8] = garde seulement 8 caractères (suffisant pour l'exemple)

    df_crm['id_anonyme'] = df_crm['email_clean'].apply(hash_email)
    # apply() : applique la fonction à chaque ligne

    # Suppression doublons (même email, même revenu)
    df_crm_clean = df_crm.drop_duplicates(subset=['email_clean', 'revenu'])
    # subset : colonnes à considérer pour l'unicité

    # Nettoyage revenu (texte → nombre)
    df_crm_clean['revenu_num'] = df_crm_clean['revenu'].str.replace('€',
'').astype(float)
    # str.replace() : enlève le symbole | astype(float) : convertit en nombre décimal

    # Standardisation dates
    df_crm_clean['date_iso'] =
pd.to_datetime(df_crm_clean['date_inscription']).dt.strftime('%Y-%m-%d')
    # to_datetime : parse les dates | strftime : format ISO standard

    # Jointure avec données web (merge)
    df_web.rename(columns={'user_id': 'id'}, inplace=True)  # Alignement clés
    df_final = df_crm_clean.merge(df_web, on='id', how='left')
    # merge() = jointure SQL | how='left' = garde tous les CRM même sans web

    # Filtrage qualité (sessions trop courtes = bots ?)
    df_final = df_final[df_final['temps_session_sec'] > 30]

    # Sélection colonnes finales (schéma strict)
    df_result = df_final[[
        'id_anonyme', 'revenu_num', 'date_iso',
        'pages_vues', 'temps_session_sec', 'conversion'
    ]]

    print(f"2. TRANSFORM : {len(df_result)} lignes après nettoyage")

    # LOAD - Stockage dans Data Warehouse structuré
    # (Simulation : dans la vraie vie → PostgreSQL, Snowflake, BigQuery)
    df_result.to_parquet("etl_output.parquet", index=False)
    print(f"3. LOAD : Export Parquet ({len(df_result)} lignes)")
```



```
    return df_result

# Exécution ETL
result_etl = pipeline_etl(data_source_crm, data_source_web)
print("\nRésultat ETL :")
print(result_etl)

# ============================================================
# APPROCHE 2 : ELT (Extract-Load-Transform)
# ============================================================
# Logique : Charger TOUT brut, puis transformer en SQL
# Quand l'utiliser : Big Data, exploration rapide, schéma évolutif

print("\n" + "=" * 70)
print("APPROCHE ELT : Chargement brut puis transformation SQL")
print("=" * 70)

def pipeline_elt(crm_data, web_data):
    """
    Pipeline ELT : Minimal processing, then SQL transformation
    """

    # EXTRACT - Identique
    df_crm = pd.DataFrame(crm_data)
    df_web = pd.DataFrame(web_data)

    # LOAD IMMÉDIAT - On stocke TOUT, même les anomalies
    # Simulation Data Lake / Warehouse (S3, BigQuery, Snowflake)
    df_crm.to_parquet("raw_crm.parquet", index=False)
    df_web.to_parquet("raw_web.parquet", index=False)
    print(f"1. LOAD BRUT : {len(df_crm)} + {len(df_web)} lignes stockées
telles quelles")

    # TRANSFORM - En SQL (simulation avec Pandas SQL-like)
    # Dans la vraie vie : dbt, BigQuery SQL, Spark SQL

    # Rechargement pour transformation
    raw_crm = pd.read_parquet("raw_crm.parquet")
    raw_web = pd.read_parquet("raw_web.parquet")

    # Transformation SQL-like avec Pandas
    # Équivalent SQL :
    # SELECT
    #   MD5(LOWER(email)) as id_anonyme,
    #   CAST(REPLACE(revenu, '€', '') AS FLOAT) as revenu_num,
    #   ...
    # FROM raw_crm
    # LEFT JOIN raw_web ON id = user_id
    # WHERE temps_session_sec > 30

    # Transformation en chaîne (fluent API)
    df_transformed = (
        raw_crm
        .assign(
            email_clean=lambda x: x['email'].str.lower(),
            id_anonyme=lambda x: x['email_clean'].apply(
                lambda e: hashlib.md5(e.encode()).hexdigest()[:8]
            ),
```

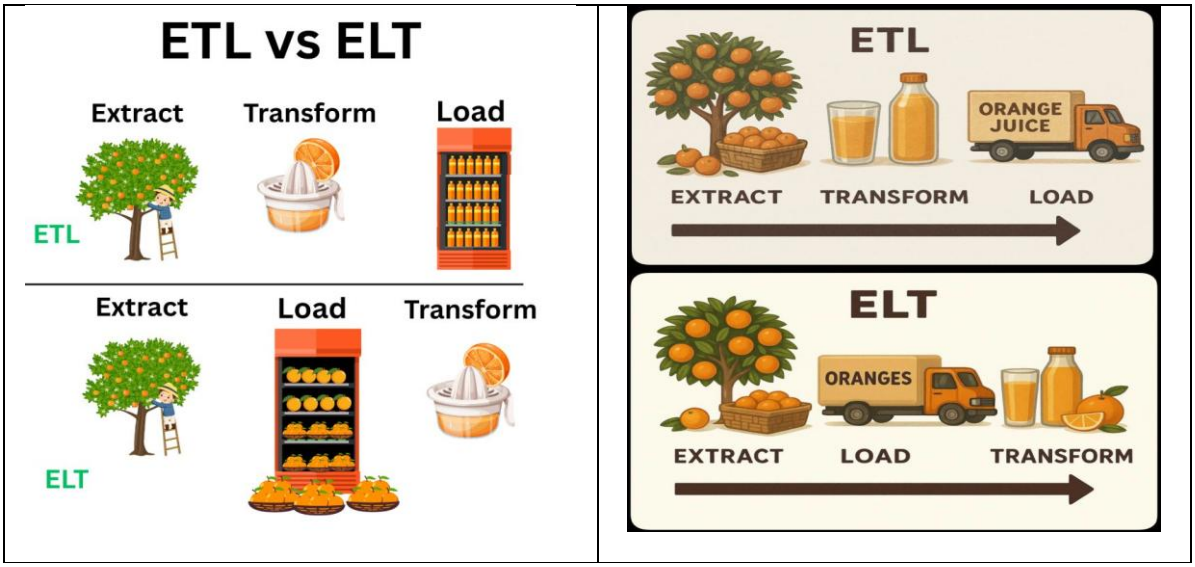


```
            revenu_num=lambda x: x['revenu'].str.replace('€',
'').astype(float),
            date_iso=lambda x: pd.to_datetime(x['date_inscription'])
        )
        .merge(raw_web.rename(columns={'user_id': 'id'}), on='id',
how='left')
        .query('temps_session_sec > 30')  # SQL-like filtering
        .drop_duplicates(subset=['email_clean', 'revenu'])
        [['id_anonyme', 'revenu_num', 'date_iso', 'pages_vues',
'temps_session_sec', 'conversion']]
    )
    # assign() = crée nouvelles colonnes | lambda x : fonction anonyme rapide
    # query() = syntaxe SQL dans Pandas | plus lisible que df[df['col'] > 0]

    print(f"2. TRANSFORM SQL : {len(df_transformed)} lignes finales")

    return df_transformed

# Exécution ELT
result_elt = pipeline_elt(data_source_crm, data_source_web)
print("\nRésultat ELT (identique mais process différent) :")
print(result_elt)
```

```
============================================================
```

COMPARAISON ET QUAND CHOISIR QUOI → COMPARAISON ETL vs ELT

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



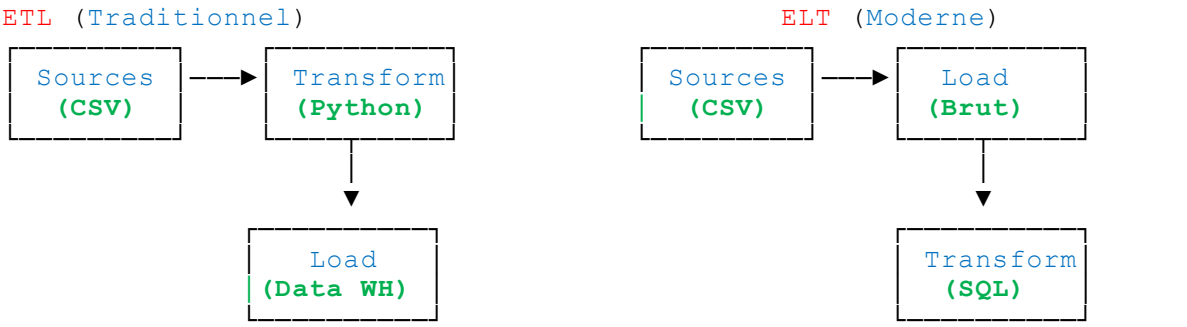
Critère	ETL (Transform d'abord)	ELT (Load d'abord)
Vitesse initiale	Lente (processing avant stockage)	Rapide (chargement brut)
Flexibilité	Faible (schéma figé)	Haute (re-transformable)
Coût stockage	Faible (données nettoyées)	Élevé (tout stocké)
Coût compute	Élevé (serveurs dédiés)	Faible (SQL cloud scalable)
Qualité données	Garantie en amont	Contrôlable a posteriori

Critère	ETL (Transform d'abord)	ELT (Load d'abord)
Cas d'usage IA	Données sensibles, production	Exploration, Big Data, Data Lake
Outils typiques	Python, Spark, Airflow	dbt, BigQuery, Snowflake, Fivetran

```
# Recommandation pour ce module
```

- `ETL pour : Données clients sensibles (RGPD), pipelines de production`
- `ELT pour : Exploration de corpus texte (RAG), logs volumineux`
- `Hybride : ETL pour l'anonymisation, ELT pour l'analyse`

Visualisation du flux ETL vs ELT :



Avantage : Qualité garantie Data
Inconvénient : Rigidité coûteux

Avantage : Flexibilité, Big

Inconvénient : Stockage

Partie 3 : Stockage Moderne — Du Data Lake au Lakehouse

3.1 Pourquoi le Lakehouse change tout pour l'IA ?

Le **Lakehouse** combine le stockage bon marché du Data Lake avec la fiabilité des Data Warehouses. ***Essentiel pour les datasets d'entraînement massifs***.

```
# ============================================================
# SIMULATION LAKEHOUSE AVEC DELTA LAKE (PYSPARK)
# ============================================================
# Installation requise : pip install pyspark delta-spark

from pyspark.sql import SparkSession
from pyspark.sql.functions import col, lit, current_timestamp
from delta import configure_spark_with_delta_pip

# Configuration Spark avec support Delta Lake
builder = SparkSession.builder \
    .appName("LakehouseIA") \
      .config("spark.sql.extensions",
      "io.delta.sql.DeltaSparkSessionExtension") \
      .config("spark.sql.catalog.spark_catalog",
      "org.apache.spark.sql.delta.catalog.DeltaCatalog") \
      .config("spark.driver.memory", "2g")   # Limite mémoire pour démo
```



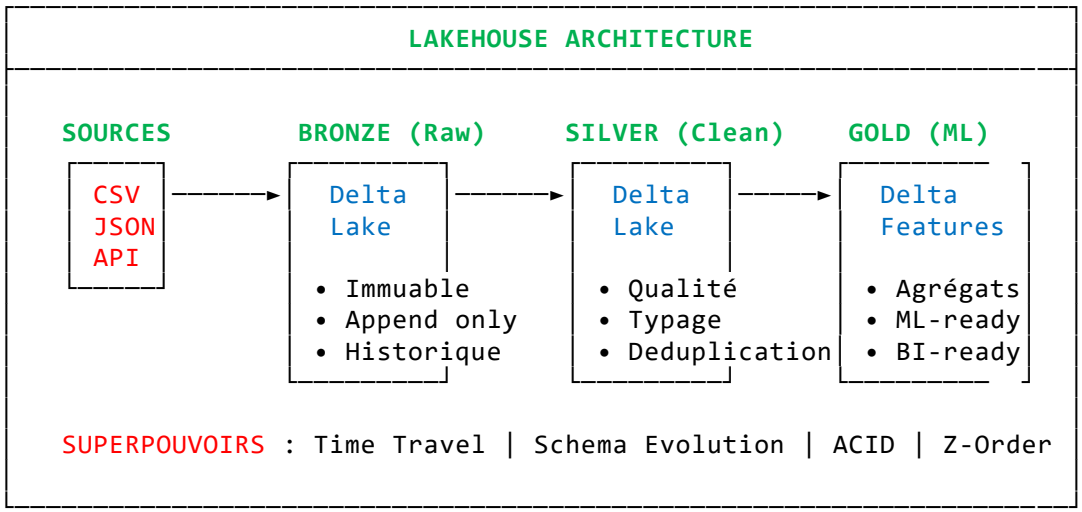
```
# Création session (point d'entrée Spark)
spark = configure_spark_with_delta_pip(builder).getOrCreate()
# SparkSession = contexte d'exécution distribuée
# configure_spark_with_delta_pip = active l'extension Delta

print("=" * 70)
print("ARCHITECTURE LAKEHOUSE : Bronze → Silver → Gold")
print("=" * 70)

# ============================================================
# COUCHE BRONZE (Données brutes, immutables, historisées)
# ============================================================
# Analogie : Archives nationales - on ne modifie jamais l'original

print("\n COUCHE BRONZE : Ingestion brute")

# Données simulées (logs d'application)
data_bronze = [
    ("2024-01-15", "user_001", "click", "{\"page\": \"/produit\"}",
"mobile"),
    ("2024-01-15", "user_002", "achat", "{\"montant\": 150.0}", "desktop"),
    ("2024-01-15", "user_001", "click", "{\"page\": \"/panier\"}",
"mobile"),
    ("2024-01-16", None, "error", "{\"code\": 500}", "bot"),  # Donnée
corrompue
]

# Création DataFrame Spark (distribué)
columns = ["date", "user_id", "event_type", "payload", "device"]
df_bronze = spark.createDataFrame(data_bronze, columns)

# Écriture Delta Lake (format Lakehouse)
df_bronze.write \
    .format("delta") \
    .mode("overwrite") \
    .save("lakehouse/bronze/events")
# format("delta") = active les fonctionnalités ACID
# mode("overwrite") = remplace si existe (premier chargement)

print(f"  → {df_bronze.count()} événements stockés (format Delta)")
print(f"  → Localisation : lakehouse/bronze/events")

# ============================================================
# COUCHE SILVER (Nettoyage, déduplication, typage)
# ============================================================
# Analogie : Bibliothèque - organisé, catalogué, accessible

print("\n COUCHE SILVER : Nettoyage et structuration")

# Lecture Bronze (time travel possible ici)
df_silver = spark.read.format("delta").load("lakehouse/bronze/events")

# Transformation qualité (Spark SQL)
df_silver_clean = df_silver \
    .filter(col("user_id").isNotNull()) \  # Supprime lignes sans user
    .withColumn("date_parsed", col("date").cast("date")) \  # Typage fort
    .withColumn("processed_at", current_timestamp()) \  # Audit
    .withColumn("source", lit("web_app"))  # Traçabilité

# MERGE (Upsert) : mise à jour incrémentale intelligente
```



```
# Si user_id+date existe → met à jour, sinon → insère
# (Simulation simplifiée ici par overwrite)

df_silver_clean.write \
    .format("delta") \
    .mode("overwrite") \
    .save("lakehouse/silver/events")

print(f"  → {df_silver_clean.count()} événements après nettoyage")
print(f"  → Colonnes enrichies : date_parsed, processed_at, source")

# ============================================================
# COUCHE GOLD (Agrégation, features pour ML)
# ============================================================
# Analogie : Encyclopédie - synthèse, prêt à l'emploi

print("\n COUCHE GOLD : Features pour modèles IA")

# Lecture Silver
df_gold = spark.read.format("delta").load("lakehouse/silver/events")

# LIGNE 93 : Agrégation (features comportementales)
from pyspark.sql.functions import count, sum as spark_sum, avg

df_features = df_gold.groupBy("user_id").agg(
    count("*").alias("nb_events"),            # Feature 1 : activité
    countDistinct("event_type").alias("nb_types_differents"),   # Feature 2 :
diversité
    spark_sum(
        col("payload.montant").cast("float")   # Extraction JSON imbriqué
    ).alias("total_depenses")                  # Feature 3 : valeur
).fillna(0)   # Remplace NULL par 0 pour ML

# Écriture optimisée pour entraînement ML
df_features.write \
    .format("delta") \
    .mode("overwrite") \
    .save("lakehouse/gold/user_features")

print(f"  → {df_features.count()} profils utilisateurs créés")
print(f"  → Features : nb_events, nb_types_differents, total_depenses")

# ============================================================
# FONCTIONNALITÉS AVANCÉES DU LAKEHOUSE
# ============================================================

print("\n" + "=" * 70)
print("SUPERPOUVOIRS DU LAKEHOUSE")
print("=" * 70)

# 1. TIME TRAVEL (Voyage dans le temps)
print("\n 1. TIME TRAVEL : Revenir à une version antérieure")
# Dans la vraie vie : SELECT * FROM table VERSION AS OF 5
print("   → 'Oups, j'ai supprimé des données hier...'")
print("   → Solution : spark.read.format('delta').option('versionAsOf',
5).load(path)")
print("   → Utile pour : Reproductibilité des expériences ML")

# 2. SCHEMA EVOLUTION (Évolution sans cassure)
```



```
print("\n 2. SCHEMA EVOLUTION : Ajouter des colonnes sans casser
l'historique")
print("   → Ancien code lit 5 colonnes, nouveau code lit 7 colonnes")
print("   → Les deux fonctionnent en parallèle !")

# 3. Z-ORDER (Optimisation requêtes)
print("\n  3. Z-ORDER : Organiser les données pour accès rapide")
print("   → OPTIMIZE lakehouse/gold/user_features ZORDER BY (user_id)")
print("   → Recherche par user_id : 10x plus rapide")

# 4. VACUUM (Gestion coûts)
print("\n 4. VACUUM : Nettoyer les vieilles versions")
print("   → VACUUM RETAIN 168 HOURS (garde 7 jours d'historique)")
print("   → Réduit les coûts de stockage cloud")

# Arrêt propre Spark
spark.stop()
print("\n Démonstration Lakehouse terminée")
```




```
# Comparaison avec ancienne génération
```

LAKEHOUSE vs ANCIENNES ARCHITECTURES

Caractéristique	Data Warehouse (90s)	Data Lake (2010)	Lakehouse (2020)
Stockage	Cher (SAN/NAS)	Bon marché (S3)	Bon marché (S3)
Qualité des données	Excellente	Faible (data swamp)	Excellente
Types de données	Structuré seulement	Tout (brut)	Tout (structuré + brut)
Performance	Rapide	Lente	Rapide (indexation)
ML / IA	Difficile	Possible	Natif
Coût total	$$$$$	$$	$$$

Partie 4 : Préparation Pratique — Du CSV au Dataset ML

4.1 Pipeline complet : Données brutes → Features ML

Ce code intègre tout ce que nous avons vu : types de données, ETL/ELT, Lakehouse.

```
# ============================================================
# PIPELINE COMPLET : Fichier brut → Dataset prêt pour Fine-tuning LLM
# ============================================================

import pandas as pd
import json
import hashlib
from datetime import datetime
import numpy as np

print("=" * 70)
print("PIPELINE IA : Préparation dataset pour Fine-tuning")
print("=" * 70)

# ÉTAPE 1 - INGESTION (Type 1: Structuré + Type 3: Texte)
print("\n ÉTAPE 1 : Ingestion multi-sources")

# Source A : Données structurées (CRM)
clients = pd.DataFrame([
    {"client_id": 1, "nom": "Alice Martin", "segment": "premium",
"anciennete_mois": 24},
    {"client_id": 2, "nom": "Bob Dupont", "segment": "standard",
"anciennete_mois": 6},
    {"client_id": 3, "nom": "Charlie Bernard", "segment": "premium",
"anciennete_mois": 12},
])

# Source B : Données semi-structurées (Logs JSON)
with open("interactions.json", "w") as f:
    json.dump([
        {"client_id": 1, "date": "2024-01-15", "canal": "email",
"satisfaction": 4, "commentaire": "Service rapide, merci !"},
        {"client_id": 1, "date": "2024-01-20", "canal": "chat",
"satisfaction": 5, "commentaire": "Excellent conseiller, très patient."},
        {"client_id": 2, "date": "2024-01-10", "canal": "tel",
"satisfaction": 2, "commentaire": "Attente trop longue, déçu."},
        {"client_id": 3, "date": "2024-01-25", "canal": "email",
"satisfaction": 5, "commentaire": "Résolution immédiate, parfait."},
    ], f, indent=2)

# Lecture JSON (Type semi-structuré → structuré)
interactions = pd.read_json("interactions.json")
print(f"  → Clients : {len(clients)} lignes (structuré)")
print(f"  → Interactions : {len(interactions)} lignes (JSON)")

# ÉTAPE 2 - TRANSFORMATION (Approche hybride ETL/ELT)
print("\n ÉTAPE 2 : Transformation pour IA")

# Anonymisation (RGPD) - ETL strict
def anonymiser_nom(nom):
    """Garde initiale + hash, supprime identité"""
    initiale = nom[0]  # Première lettre
    hash_nom = hashlib.sha256(nom.encode()).hexdigest()[:6]
```



```
    # sha256 = plus sécurisé que md5 pour données sensibles
    return f"{initiale}***_{hash_nom}"

clients['nom_anonyme'] = clients['nom'].apply(anonymiser_nom)

# Feature Engineering (création variables ML)
# Règle métier : Score client = ancienneté * coefficient segment
coefficient_segment = {"premium": 2.0, "standard": 1.0, "basic": 0.5}
clients['score_fidelite'] = clients['anciennete_mois'] *
clients['segment'].map(coefficient_segment)
# .map() = applique le dictionnaire de correspondance

# Agrégation interactions (ELT-like)
stats_interactions = interactions.groupby('client_id').agg({
    'satisfaction': 'mean',            # Satisfaction moyenne
    'commentaire': ' '.join,           # Concaténation textes (Type 3)
    'date': 'count'                    # Nombre d'interactions
}).rename(columns={'date': 'nb_contacts', 'satisfaction':
'satisfaction_moyenne'})

# Jointure (merge)
dataset_ml = clients.merge(stats_interactions, on='client_id', how='left')

# ÉTAPE 3 - CRÉATION FORMAT FINE-TUNING (Type 4: Vectoriel implicite)
print("\n ÉTAPE 3 : Génération dataset Fine-tuning")

def creer_prompt_finetuning(row):
    """
    Crée un exemple d'entraînement pour un LLM
    Format : Instruction → Input → Output
    """
    instruction = "Générer un résumé client pour le service après-vente."

    input_data = f"""
Client : {row['nom_anonyme']}
Segment : {row['segment']}
Ancienneté : {row['anciennete_mois']} mois
Score fidélité : {row['score_fidelite']:.1f}
Satisfaction moyenne : {row['satisfaction_moyenne']}/5
Nombre de contacts : {int(row['nb_contacts'])}
Historique : {row['commentaire'][:100]}...
"""

    # Output = ce qu'on veut que le LLM apprenne à générer
    if row['satisfaction_moyenne'] >= 4.5:
        output = f"Client VIP très satisfait. Priorité haute. Score:
{row['score_fidelite']:.0f}"
    elif row['satisfaction_moyenne'] >= 3:
        output = f"Client fidèle à conserver. Score:
{row['score_fidelite']:.0f}"
    else:
        output = f"Client insatisfait nécessitant suivi. Score:
{row['score_fidelite']:.0f}"

    return {
        "instruction": instruction,
        "input": input_data.strip(),
        "output": output,
        "metadata": {
```



```
            "client_id_hash":
hashlib.md5(str(row['client_id']).encode()).hexdigest()[:8],
            "date_generation": datetime.now().isoformat(),
            "version_dataset": "1.0"
        }
    }

# Application à chaque ligne
dataset_finetuning = dataset_ml.apply(creer_prompt_finetuning,
axis=1).tolist()
# axis=1 = applique fonction sur chaque ligne (axis=0 = colonnes)
# .tolist() = convertit Series Pandas en liste Python standard

# Export formats standards
print(f"  → {len(dataset_finetuning)} exemples générés")

# Format JSONL (JSON Lines) - standard pour LLMs
with open("dataset_finetuning.jsonl", "w", encoding="utf-8") as f:
    for exemple in dataset_finetuning:
        f.write(json.dumps(exemple, ensure_ascii=False) + "\n")
        # json.dumps = convertit dict en string JSON
        # ensure_ascii=False = conserve accents
        # \n = saut de ligne (format JSONL = 1 JSON par ligne)

print(f"  → Export : dataset_finetuning.jsonl")

# Format Parquet (pour features structurées)
dataset_ml[['client_id', 'score_fidelite', 'satisfaction_moyenne',
'nb_contacts']].to_parquet(
    "features_ml.parquet", index=False
)
print(f"  → Export : features_ml.parquet (features numériques)")

# ÉTAPE 4 - VALIDATION QUALITÉ
print("\n ÉTAPE 4 : Validation qualité dataset")

# Vérifications automatiques
assert len(dataset_finetuning) > 0, "Dataset vide !"
assert all('instruction' in ex for ex in dataset_finetuning), "Champ
instruction manquant"
assert all('output' in ex for ex in dataset_finetuning), "Champ output
manquant"

# Statistiques de qualité
longueurs_input = [len(ex['input']) for ex in dataset_finetuning]
print(f"  → Longueur moyenne input : {np.mean(longueurs_input):.0f}
caractères")
print(f"  → Longueur max input : {max(longueurs_input)} caractères")
# Attention : LLMs ont une limite de contexte (4096, 8192, 128k tokens...)

# Exemple final
print("\n" + "=" * 70)
print("EXEMPLE D'ENTRÉE GÉNÉRÉE (pour vérification)")
print("=" * 70)
print(json.dumps(dataset_finetuning[0], indent=2, ensure_ascii=False))

# Récapitulatif architecture données
```

ARCHITECTURE DE DONNÉES UTILISÉE

Types de données utilisés :

→ Structuré (Type 1) : Tables clients, métriques agrégées
→ Semi-structuré (Type 2) : Logs JSON, métadonnées dataset
→ Non-structuré (Type 3) : Textes commentaires (traités comme strings)
→ Vectoriel (Type 4) : Implicite (embeddings générés plus tard par LLM)

Pipeline :

Sources (CSV/JSON) → **ETL (Anonymisation)** → **ELT (Agrégation)**
→ **Features Engineering** → Format Fine-tuning (JSONL) + Features (Parquet)

Stockage :

- **JSONL :** Corpus pour entraînement LLM (S3, partagé avec équipe ML)
- **Parquet :** Features numériques pour modèles classiques ou hybridation

Récapitulatif pédagogique du Module 2

Concept	Définition simple	Pourquoi c'est crucial pour l'IA
4 Types de données	Structuré (table), Semi (JSON), Non (texte/image), Vectoriel (embedding)	Choisir le bon stockage et traitement selon le cas d'usage
ETL	Transformer avant stocker	Qualité garantie, RGPD, production
ELT	Stocker puis transformer SQL	Flexibilité, Big Data, exploration
Lakehouse	Data Lake + fiabilité Warehouse	Stockage cheap + qualité données ML + time travel
Delta Lake	Format de fichier avec ACID	Reproductibilité des expériences ML, gestion versions datasets
Fine-tuning dataset	Format instruction-input-output	Apprendre au LLM une tâche spécifique avec exemples

