Agent conversationnel pour l'interrogation de la base de données Open Food Facts

Rapport final

Travail présenté à Luc Lamontagne

IFT-6005 - Projet intégrateur

réalisé par

Alain Boisvert

994 029 313

9 avril 2025

Table des matières

1 Introduction

L'accès aux informations nutritionnelles reste souvent limité par des interfaces techniques nécessitant des compétences en langages de requête comme SQL. Cette barrière empêche de nombreux utilisateurs d'exploiter pleinement des bases de données comme Open Food Facts [1], qui contient des informations détaillées sur des milliers de produits alimentaires.

Ce projet vise à développer un agent conversationnel utilisant des grands modèles de langage (LLM) pour permettre aux utilisateurs de poser des questions en langage naturel comme (*« Quelles collations sans allergènes ont un Nutri-score A ? »*). Cette approche démocratise l'accès aux données nutritionnelles tout en améliorant la qualité des réponses grâce à l'exploitation directe de sources structurées.

Ce rapport de mi-session présente l'état d'avancement du projet à la mi-session, les défis rencontrés et les solutions implémentées.

Première partie

Interrogation de la base avec DuckDB

Cette première partie du rapport, réalisée lors de la première moitié du semestre, présente l'approche d'interrogation de la base de données Open Food Facts à l'aide de DuckDB et d'un agent conversationnel basé sur un LLM.

2 Rappel de l'objectif du projet

L'objectif de ce projet est de développer un agent conversationnel permettant aux utilisateurs d'interroger la base de données Open Food Facts en langage naturel. Le système doit comprendre les questions des utilisateurs sur les produits alimentaires, les convertir en requêtes SQL, et fournir des réponses claires et précises. Les données manquantes ou incomplètes doivent être compensées par une recherche alternative dans le Guide alimentaire canadien [2].

3 Approche proposée

3.1 Architecture du système

L'architecture modulaire du système comprend les composants suivants :

- **Module de dialogue :** Gère les conversations avec l'utilisateur en utilisant un LLM pré-entraîné.
- **Convertisseur texte-SQL :** Transforme les questions en requêtes SQL adaptées à Open Food Facts. Il utilise d'abord une recherche sémantique dans un dictionnaire de données pour trouver les colonnes pertinentes.
- **Connecteur de base de données :** Communique avec DuckDB pour exécuter les requêtes. Les requêtes SQL sont vérifiées avant exécution pour garantir leur sécurité.
- **Recherche sur le Web :** Consulte le Guide alimentaire canadien quand les informations manquent dans la base de données.
- **Générateur de réponses :** Transforme les résultats bruts en réponses naturelles, en précisant les sources.

1. https://world.openfoodfacts.org/
2. https://guide-alimentaire.canada.ca/fr/

La figure 1 présente l'architecture modulaire du système, illustrant comment les questions des utilisateurs sont traitées par le module de dialogue LLM, enrichies par la recherche sémantique des colonnes pertinentes, puis converties en requêtes SQL pour interroger Open Food Facts, avec un repli vers le Guide alimentaire canadien lorsque nécessaire.

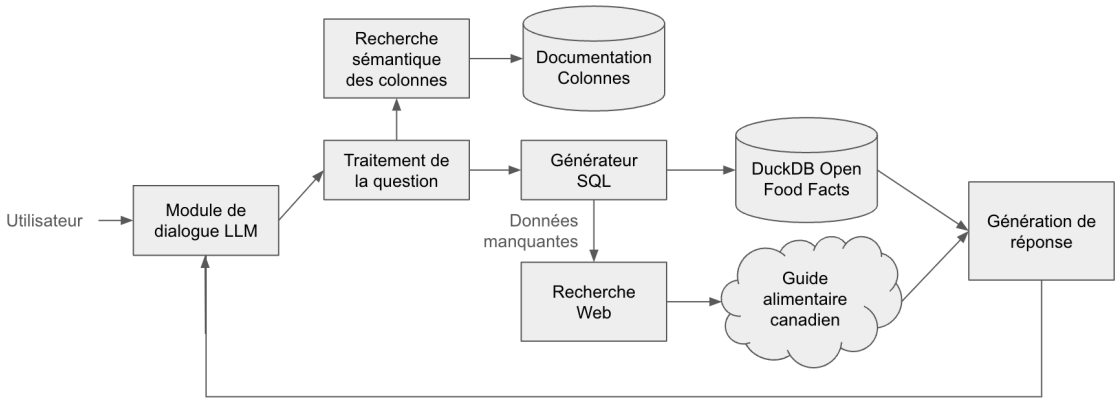

FIGURE 1 – Architecture du système d'agent conversationnel pour l'interrogation d'Open Food Facts

3.2 Sélection des technologies

J'ai choisi ces technologies pour mon projet :

- **DuckDB** comme base de données, car :
  - Elle est rapide pour les requêtes analytiques
  - Elle gère bien la mémoire
  - Elle supporte les fichiers Parquet et les requêtes SQL complexes
- **Hugging Face Smolagents** comme framework d'agent, car :
  - Il est simple à utiliser
  - Il permet d'intégrer différents LLMs et outils
  - Il est bien documenté et supporté
- **Modèles de langage** :
  - Pour le développement : Llama3.1:8B-Instruct via Ollama (en local)
  - Pour les tests : Llama3.1:8B-Instruct et Qwen2.5:7B via Ollama (en local), et Claude 3.5 Sonnet (modèle commercial via API)
  - Cette approche limite les coûts d'API
- **FAISS** pour la recherche sémantique :
  - Il identifie rapidement les colonnes pertinentes pour chaque question
  - Il transforme la documentation des colonnes en vecteurs faciles à comparer

Notons qu'au début du projet, j'envisageais d'utiliser CrewAI [3] pour l'orchestration des agents, mais j'ai finalement opté pour SmolAgents pour sa simplicité et sa flexibilité. Contrairement à CrewAI, qui impose une structure rigide avec des rôles prédéfinis, SmolAgents permet une approche plus modulaire où les agents interagissent librement via des outils définis par l'utilisateur. De plus, il s'intègre facilement aux outils déjà utilisés dans le projet et évite la complexité d'une organisation centralisée, ce qui rend le développement plus rapide et plus facile.

Aucune alternative à FAISS n'a été envisagée, car cette bibliothèque répond parfaitement aux besoins du projet. Elle est open-source, largement reconnue, soutenue par Facebook et simple à utiliser, tout en offrant d'excellentes performances pour la recherche sémantique.

3. https://www.crewai.com/open-source

4 État d'avancement et tâches réalisées

Le développement de l'agent conversationnel a progressé significativement durant cette première phase du projet. Cette section détaille les différentes tâches accomplies et leur état d'avancement, en suivant le plan initial.

4.1 Mise en place de l'environnement de développement

J'ai d'abord créé un environnement de développement complet :

- **Infrastructure et outils**
  - Un dépôt GitHub [4] pour le code et la documentation
  - Un environnement virtuel Python avec les bonnes dépendances
  - Des outils comme Black pour le formatage et Pylint pour l'analyse de code
- **Dépendances principales**
  - DuckDB pour la base de données
  - Smolagents [5] pour le framework d'agent
  - FAISS [6] pour la recherche sémantique
  - Les bibliothèques habituelles pour l'analyse de données et le traitement du langage
- **Configuration des API**
  - Des variables d'environnement pour les clés API dans un fichier `.env`
  - L'accès à l'API d'Anthropic pour Claude 3.5 Sonnet
  - L'interface avec Ollama pour les tests locaux

4.2 Préparation de la base de données

J'ai téléchargé le fichier Parquet d'Open Food Facts (3,6 millions de produits) :

```
wget -P data/ https://huggingface.co/datasets/openfoodfacts/product-database/
    resolve/main/food.parquet
```

Listing 1 – Téléchargement du fichier Parquet

Ensuite, j'ai converti le fichier Parquet en base DuckDB pour faciliter les requêtes SQL :

```
DATA_DIR = Path("../data")
PARQUET_PATH = DATA_DIR / "food.parquet"
FULL_DB_PATH = DATA_DIR / "food_full.duckdb"

con = duckdb.connect(str(FULL_DB_PATH), config={'memory_limit': '8GB'})
con.execute(f"CREATE TABLE products AS SELECT * FROM '{PARQUET_PATH}'")
con.close()
```

Listing 2 – Conversion en base DuckDB

Pour accélérer les requêtes, j'ai créé une version avec seulement les 94 802 produits canadiens :

```
FILTERED_DB_PATH = DATA_DIR / "food_canada.duckdb"

con = duckdb.connect(str(FILTERED_DB_PATH))
con.execute(f"ATTACH DATABASE '{FULL_DB_PATH}' AS full_db")
con.execute(f"""
    CREATE TABLE products AS
    SELECT * FROM full_db.products
    WHERE array_contains(countries_tags, 'en:canada')
""")
```

4. https://github.com/boisalai/ift-6005
5. https://github.com/huggingface/smolagents
6. https://github.com/facebookresearch/faiss

La création des bases DuckDB s'effectue via le script `data.py`.

J'ai aussi analysé les données et découvert que certaines colonnes sont très complètes ($> 95\%$) alors que d'autres sont peu renseignées ($< 30\%$).

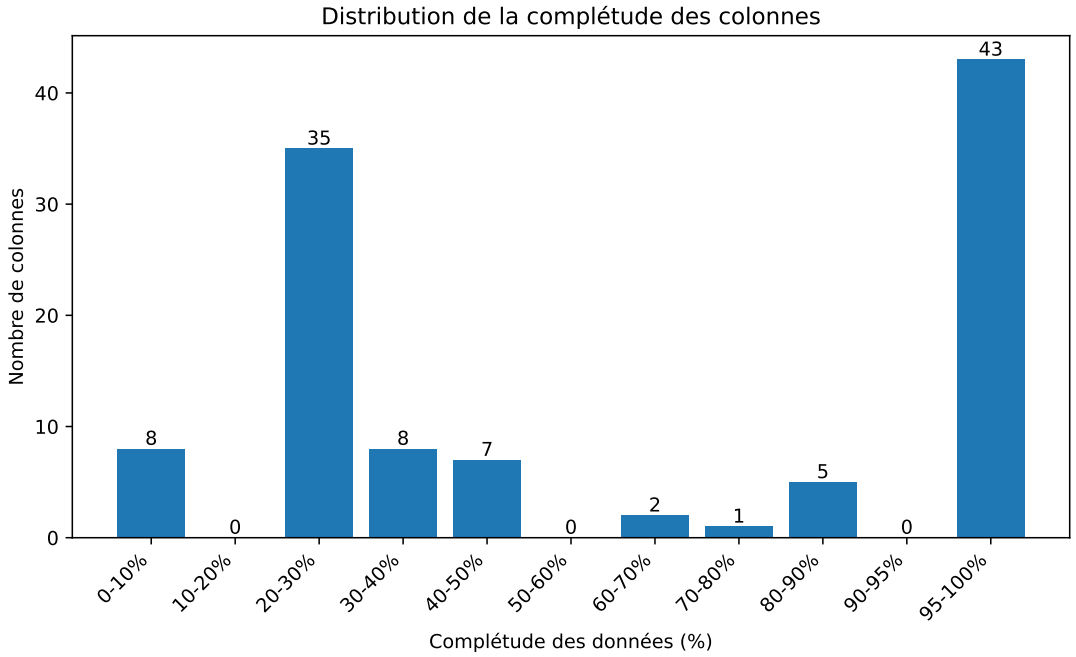

FIGURE 2 – Distribution de la complétude des colonnes de la base de données

J'ai également remarqué que certaines colonnes ont des structures complexes. Par exemple, la colonne `categories` contient du texte libre séparé par des virgules, alors que `categories_tags` contient des identifiants standardisés incluant des préfixes de langue.

Exemple de données pour la colonne `categories` :

```
"Sweeteners,Syrups,Simple syrups,Agave syrups"
"Plant-based beverages,Fruit-based,Juices and nectars,Fruit juices,Lemon juice"
"Snacks,Sweet snacks,Biscuits and cakes,Cakes"
```

Exemple de données pour la colonne `categories_tags` :

```
["en:plant-based-foods-and-beverages", "en:beverages",
    "en:plant-based-beverages", "en:coconut-milks"]
["en:snacks", "en:sweet-snacks", "en:cocoa-and-chocolate-products",
    "en:chocolates"]
["en:dairies", "en:milk", "en:whole-milk"]
```

4.3 Documentation des données

J'ai créé un dictionnaire détaillé (`columns_documentation.json`) pour aider l'agent à générer des requêtes SQL. Il contient des informations sur chacune des 109 colonnes. La documentation pour chaque colonne ressemble à ceci :

```
"nova_groups_tags": {
  "type": "VARCHAR[]",
  "description": "Array containing NOVA food classification group tags. NOVA
      ...",
  "examples": [
    "['en:1-unprocessed-or-minimally-processed-foods']",
    "['en:4-ultra-processed-food-and-drink-products']",
    "['en:3-processed-foods']"
  ],
  "is_nullable": true,
  "common_queries": [
    {
      "description": "Distribution of products across NOVA groups",
      "sql": "SELECT nova_groups_tags, COUNT(*) as product_count FROM products
          ..."
    },
    /* Autres... */
  ]
},
```

Cette documentation est générée par un agent qui :

- Interroge la colonne dans la base de données
- Recherche des informations sur le site Open Food Facts
- Propose une description de la colonne
- Génère des requêtes SQL typiques
- Ajoute la documentation au fichier JSON

Ces tâches s'effectuent via le script `docoff.py`.

4.4 Création du jeu de test

J'ai créé un jeu de 100 questions-réponses en anglais et en français en utilisant les requêtes SQL documentées à l'étape précédente. Pour chaque requête SQL, un agent :

- Évalue si la requête répond à une question pertinente pour un consommateur
- Génère des questions en français et en anglais
- Crée des réponses claires correspondant à la requête SQL
- Vérifie que les résultats de la requête permettent de répondre à la question

Les paires sont stockées dans `qa_pairs.json` avec cette structure :

```
{
  "column": "nutriscore_score",
  "sql": "SELECT code, product_name, nutriscore_score FROM products WHERE
      nutriscore_score IS NOT NULL ORDER BY nutriscore_score ASC LIMIT 50",
  "questions": {
    "fr": "Quels sont les produits alimentaires les plus sains selon le Nutri-
        Score ?",
    "en": "What are the healthiest food products according to the Nutri-Score?"
  },
  "answers": {
    "fr": "D'après les résultats, les produits les plus sains sont
        principalement des légumineuses comme les lentilles rouges et les pois
        jaunes cassés, avec un Nutri-Score de -17, ce qui est excellent.",
    "en": "According to the results, the healthiest products are mainly legumes
        such as red split lentils and yellow split peas, with a Nutri-Score of
        -17, which is excellent."
  }
},
/* Autres... */
```

Le jeu de 100 questions-réponses est créé avec le script `question_answer.py`.

4.5 Développement de la recherche sémantique

J'ai implémenté une recherche sémantique avec FAISS et le modèle `all-MiniLM-L6-v2` pour identifier les colonnes pertinentes dans la base de données :

- **Préparation des embeddings**
  - La documentation des 109 colonnes est convertie en vecteurs
  - Le modèle MiniLM génère des embeddings de dimension 384
  - Ce modèle multilingue fonctionne en français et en anglais
- **Indexation FAISS**
  - Un index est créé pour les recherches rapides par similarité
  - Les embeddings sont sauvegardés dans des fichiers pour éviter de recalculer l'index
- **Processus de recherche**
  - Pour chaque question, l'embedding de la question est comparé à ceux des colonnes
  - Les 5 colonnes les plus similaires sont retournées avec un score
  - Seules les colonnes avec un score > 0.5 sont utilisées

Les résultats de cette recherche sont ajoutés au prompt de l'agent pour l'aider à générer de meilleures requêtes SQL.

Cette approche s'inspire des travaux de Gao et al. [1] sur l'apprentissage contextuel (*in-context learning*) pour Text-to-SQL, en adoptant une méthode similaire où l'agent identifie d'abord les colonnes sémantiquement pertinentes pour la question posée, puis utilise ces informations avec des exemples de requêtes SQL dans le prompt afin d'améliorer la qualité de la conversion du langage naturel vers SQL.

4.6 Création des outils d'agent

Un agent intelligent est un logiciel qui peut interagir avec son environnement, collecter des données et les utiliser pour effectuer des tâches autodéterminées afin d'atteindre des objectifs prédéterminés.

J'ai choisi Smolagents pour son approche simple et flexible. Cet outil est idéal pour expérimenter rapidement avec la logique d'agent, particulièrement lorsque l'application est relativement simple.

J'ai développé deux outils pour l'agent :

— **Exécution SQL sécurisée** : Permet à l'agent de générer et d'exécuter des requêtes SQL sur DuckDB avec des vérifications de sécurité
— **Recherche sur le Web** : Permet à l'agent de consulter le Guide alimentaire canadien quand les informations ne sont pas dans Open Food Facts

4.7 Stratégie d'évaluation

J'évalue l'agent selon quatre métriques principales :

— **Précision d'exécution (EX)** : Mesure si l'agent génère des requêtes SQL correctes avec la pondération suivante :
  — 20% pour avoir une requête
  — 30% pour l'exécution sans erreur
  — 50% pour des résultats corrects
— **Précision sémantique (PS)** : Compare la similarité entre les réponses de l'agent et les réponses attendues
— **Respect de séquence (RS)** : Vérifie si l'agent suit une stratégie de recherche cohérente (d'abord la base de données, puis le Guide alimentaire canadien)
— **Temps de réponse moyen (TRM)** : Mesure la rapidité du traitement

4.8 Avancement par rapport à la planification initiale

Voici un tableau comparatif entre la planification initiale et l'état actuel du projet :

— **Mise en place de l'environnement**
  — **Planification initiale** : 5h – Environnement Python, GitHub [7]
  — **État actuel** : Environnement configuré, GitHub prêt, Black & Pylint utilisés
  — **Statut** : Terminé ✓
— **Préparation de la base de données**
  — **Planification initiale** : 5h – Création d'une base DuckDB
  — **État actuel** : Base DuckDB créée avec filtrage des produits canadiens. Voir le script `data.py`
  — **Statut** : Terminé ✓
— **Création du jeu de test (100 questions)**
  — **Planification initiale** : 10h – Générer des questions et requêtes SQL
  — **État actuel** : 100 questions-réponses créées en français et anglais. Voir le script `question_answer.json` et le fichier `qa_pairs.json`
  — **Statut** : Terminé ✓
— **Documentation des données**
  — **Planification initiale** : Prévoir un dictionnaire des colonnes
  — **État actuel** : Documentation détaillée (109 colonnes, exemples, SQL). Voir le script `docoff.py` et le fichier `columns_documentation.json`
  — **Statut** : Terminé ✓
— **Développement de la conversion texte-SQL**
  — **Planification initiale** : 25h – Approche texte-SQL avec LLM
  — **État actuel** : Implémentation avec FAISS pour recherche sémantique. Voir le script `evaluation_04.py`
  — **Statut** : Terminé ✓
— **Développement du module de dialogue**
  — **Planification initiale** : 20h – Utilisation de Qwen2-7B-Instruct
  — **État actuel** : Utilisation de Llama3.1:8B, Qwen2.5:7B, Claude 3.5. Voir le script `evaluation_04.py`

7. https://github.com/boisalai/ift-6005

  - **Statut** : Terminé ✓
- **Implémentation du générateur de réponses**
  - **Planification initiale** : 25h – Transformer résultats SQL en texte
  - **État actuel** : Fonctionnel avec SmolAgents et intégration Guide alimentaire canadien. Voir le script `evaluation_04.py`
  - **Statut** : Terminé ✓
- **Stratégie d'évaluation (métriques EX, PS, RS, TRM)**
  - **Planification initiale** : 10h – Définir et tester les métriques
  - **État actuel** : Métriques définies et tests réalisés sur plusieurs modèles. Voir le script `evaluation_04.py`
  - **Statut** : Terminé ✓
- **Optimisation des requêtes SQL et prompts**
  - **Planification initiale** : 15h – Améliorer la génération SQL
  - **État actuel** : Ajustements des prompts et meilleure gestion des colonnes SQL. Voir le script `evaluation_04.py`
  - **Statut** : Terminé ✓
- **Gestion des données manquantes**
  - **Planification initiale** : 15h – Trouver des stratégies alternatives
  - **État actuel** : Approche hybride avec Guide alimentaire canadien intégrée. Voir le script `evaluation_04.py`
  - **Statut** : Terminé ✓
- **Sélection dynamique de LLM**
  - **Planification initiale** : 10h – Tester plusieurs modèles pour équilibre coût/performance
  - **État actuel** : Non réalisé.
  - **Statut** : Non réalisé.
- **Exploration de nouvelles méthodes (ex. RAG, Neo4j)**
  - **Planification initiale** : Optionnelle en phase avancée
  - **État actuel** : Mentionnée comme piste future
  - **Statut** : À explorer

En résumé, les tâches principales sont terminées selon la planification. Les optimisations avancées et l'exploration de nouvelles approches (ex. RAG) sont encore à explorer.

5 Résultats et discussion

Le tableau suivant résume les résultats des tests effectués sur l'agent conversationnel en utilisant différents modèles de langage et en évaluant les métriques EX, PS, RS et TRM sur 20 questions pour chaque modèle, en français et en anglais.

J'ai limité les tests à 20 questions par modèle afin de réduire les coûts d'API associés aumodèle commercial Claude 3.5 Sonnet. Tous les tests ont été réalisés sur un MacBook Pro M1 équipé de 16 Go de RAM.

	Llama3.1:8B		Qwen2.5:7B		Claude 3.5 Sonnet	
	en	fr	en	fr	en	fr
Nombre de questions	20	20	20	20	20	20
Précision d'exécution (EX) (%)	31.1	25.0	28.6	40.0	42.5	47.5
Exécution SQL sans échec (%)	40.0	35.0	55.0	60.0	90.0	85.0
Précision sémantique (PS) (%)	16.7	20.0	66.5	49.5	62.3	54.7
Respect de séquence (RS) (%)	100.0	100.0	100.0	100.0	100.0	100.0
Temps de réponse moyen (TRM)	432 s	273 s	239 s	201 s	37 s	41 s

TABLE 1 – Résultats des tests sur différents modèles de langage

Ici Llama3.1:8B réfère au modèle `ollama/llama3.1:8B-instruct-q8_0`, Qwen2.5:7B au modèle `ollama/qwen2.5:7B-instruct` et Claude 3.5 Sonnet au modèle `anthropic/claude-3-5-sonnet-20241022`.

Les principales observations tirées de cette évaluation se dégagent clairement :

- **Performance des modèles** : Claude 3.5 Sonnet démontre une supériorité marquée par rapport aux modèles Llama3.1:8B et Qwen2.5:7B, notamment en termes de précision d'exécution (EX) et de rapidité de traitement (TRM).
- **Différences linguistiques** : La performance varie selon la langue traitée. En français, Claude 3.5 Sonnet et Qwen2.5:7B atteignent une meilleure précision d'exécution qu'en anglais, ce qui témoigne de leur robustesse multilingue.
- **Respect de séquence** : L'ensemble des modèles obtient un score parfait (100 %) pour le respect de séquence (RS), confirmant que tous suivent rigoureusement la stratégie de recherche définie, quel que soit leur niveau de performance global.
- **Précision sémantique** : Qwen2.5:7B surpasse nettement Llama3.1:8B dans la précision sémantique, particulièrement en anglais (66,5 % contre 16,7 %), ce qui pourrait indiquer une meilleure compréhension contextuelle du domaine alimentaire.

Un aspect particulièrement révélateur émerge de l'analyse détaillée des résultats : l'écart significatif entre la génération de requêtes SQL syntaxiquement correctes et la production de résultats pertinents. Claude 3.5 Sonnet parvient à générer des requêtes exécutables dans 85-90% des cas, mais seulement 42-47% d'entre elles produisent des résultats véritablement utiles. Cette disparité suggère que la difficulté principale ne réside pas dans la maîtrise de la syntaxe SQL, mais plutôt dans la compréhension fine de la structure de données spécifique à Open Food Facts et dans la capacité à formuler des requêtes adaptées à ses particularités.

6 Problèmes rencontrés et solutions

6.1 Complexité structurelle des données

Un défi majeur a été de comprendre les structures complexes d'Open Food Facts. Sans documentation officielle complète, j'ai dû explorer la base en détail pour décoder des colonnes et comprendre les relations entre les colonnes. La base contient des structures hétérogènes, comme les colonnes `categories` et `categories_tags`. Cette complexité rend difficile la génération de bonnes requêtes SQL.

Pour résoudre ce problème, j'ai créé une documentation détaillée de la base de données avec pour chaque colonne :

- Le type de données et sa description
- Des exemples de valeurs
- Des modèles de requêtes SQL adaptées

Comme mentionné précédemment, les colonnes pertinentes pour une question de l'utilisateur sont identifiées par une recherche sémantique, puis utilisées pour enrichir le prompt de l'agent.

Je pense qu'on peut encore améliorer cette documentation, peut-être avec plus d'exemples annotés.

6.2 Limitations des modèles de langage légers

L'utilisation d'un modèle léger et gratuit (Llama3.1:8B) pendant le développement a permis d'avancer sans coûts d'API, mais a montré des limites :

- Difficulté à comprendre des instructions complexes
- Problèmes pour maintenir un format de réponse cohérent
- Génération SQL parfois incorrecte
- Difficulté à suivre des séquences d'actions

Pour résoudre ces problèmes, j'ai :

- Amélioré les prompts : Instructions plus simples et directes, avec plus d'exemples
- Modifié l'architecture : Utilisé les mécanismes de MultiStepAgent de Smolagents pour mieux suivre la progression

Des tests avec Claude 3.5 Sonnet montrent des résultats bien meilleurs, confirmant l'intérêt d'une approche hybride : développement avec des modèles légers, puis déploiement avec des modèles plus puissants.

7 Prochaines étapes

Si je continue ce projet sur cette voie, je peux :

- **Mieux documenter la structure de la base de données** : Créer une meilleure carte des relations entre colonnes, ajouter des exemples pour les structures difficiles, et enrichir la documentation avec des informations sur la distribution des données.
- **Optimiser la recherche sémantique** : Ajouter une recherche vectorielle sur les fiches produits en plus de l'approche SQL. Cela aiderait à trouver des produits similaires quand les requêtes SQL sont limitées.

Je pourrais aussi explorer une approche différente avec **RAG sur un graphe de connaissances** en utilisant Neo4j [8] et Langchain [9]. Cette méthode pourrait mieux représenter les relations complexes entre produits et informations nutritionnelles.

Deuxième partie

Graphe de produits alimentaires avec recherche sémantique

Pour la seconde partie du projet, j'ai exploré l'utilisation d'un graphe avec recherche sémantique comme alternative à l'approche DuckDB présentée précédemment. Cette section décrit la mise en œuvre et l'évaluation de cette approche.

8 Avantages potentiels d'un graphe de connaissances

Un graphe de connaissances présenterait potentiellement plusieurs avantages par rapport à une base de données relationnelle ou DuckDB pour notre cas d'usage :

- **Représentation des relations** : Les données alimentaires étant intrinsèquement relationnelles, un graphe pourrait modéliser plus naturellement les liens entre produits, ingrédients et catégories.
- **Requêtes complexes** : Les requêtes à plusieurs niveaux de relations seraient potentiellement plus intuitives à exprimer comme parcours de graphe qu'avec des jointures SQL.
- **Flexibilité et évolutivité** : L'ajout de nouvelles relations ou propriétés serait simplifié, sans nécessiter de restructuration complète du schéma.

8. https://neo4j.com/
9. https://www.langchain.com/

9 Lien avec les recherches récentes

L'article Mohammadjafari et al. [2] sur l'utilisation des LLMs pour la génération de SQL confirme l'intérêt des graphes de connaissances. Dans la section IV intitulée « LEVERAGING KNOWLEDGE GRAPHS IN LLM-BASED TEXT-TO-SQL SYSTEM », les auteurs décrivent plusieurs avantages :

- **Représentation structurée des entités et relations** : « Knowledge Graphs are structured representations of real-world entities and relationships between them, serving as a rich repository that connects and organizes semantic concepts. They provide, in a database context, a way of navigating through complex relationships and hierarchical structures that can represent large volumes of structured data effectively » [2].
- **Liaison de schéma améliorée** : « Knowledge graphs allow large language models to link natural language queries with the respective tables and columns of the database schema through explicit semantic relations from diverse entities » [2].
- **Enrichissement contextuel** : « Knowledge graphs provide that additional layer of context necessary for an LLM to disambiguate terms in NL queries. This context permits the model to resolve references to entities or relationships that may not be immediately obvious from the query itself and hence improve the accuracy of the generated SQL » [2].

Ces avantages sont pertinents pour notre projet, où la compréhension du contexte des produits alimentaires (relations entre produits, ingrédients, catégories, etc.) est essentielle pour répondre correctement aux questions des utilisateurs.

10 Choix de la technologie de graphe

J'ai d'abord envisagé Neo4j [10], base de données orientée graphe populaire, pour créer un graphe de produits alimentaires. Cependant, après plusieurs essais, j'ai rencontré des limitations avec la version gratuite de Neo4j qui restreint le nombre de relations pouvant être créées. Ces contraintes m'obligeaient à réduire significativement le nombre de produits canadiens, rendant impossible une comparaison équitable avec l'approche DuckDB.

Malgré une tentative avec une version payante de Neo4j, le temps de création du graphe s'est avéré excessivement long, probablement en raison de la localisation du serveur distant.

Pour contourner ces obstacles, j'ai opté pour NetworkX [11], une bibliothèque Python open source pour la manipulation de graphes. Cette solution présente plusieurs avantages et inconvénients par rapport à Neo4j :

NetworkX	**Neo4j**
Bibliothèque Python en mémoire	Base de données persistante dédiée
Facile à intégrer dans l'écosystème Python (NumPy, Pandas, scikit-learn)	Nécessite une connexion client-serveur
Idéal pour prototypage et graphes de taille moyenne	Optimisé pour très grands graphes en production
API Python native, courbe d'apprentissage faible	Requiert l'apprentissage du langage Cypher
Limité par la mémoire disponible	Stockage sur disque, moins limité en taille
Pas de visualisation intégrée	Visualisation puissante intégrée
Très utilisé en recherche académique	Utilisé dans les applications commerciales

TABLE 2 – Comparaison entre NetworkX et Neo4j

10. https://neo4j.com/
11. https://networkx.org/

Ce choix m'a permis de créer et manipuler un graphe de produits alimentaires entièrement en mémoire avec Python, tout en utilisant la même source de données que pour DuckDB, garantissant ainsi des analyses comparatives équitables.

11 Création du graphe

11.1 Création du graphe de produits alimentaires

J'ai développé un script `create_nx_graph.py` pour générer un graphe NetworkX à partir du même fichier parquet utilisé pour la base DuckDB. Le fonctionnement de ce script se décompose en plusieurs étapes clés :

1. Chargement du modèle SentenceTransformer pour générer des embeddings vectoriels des descriptions de produits
2. Lecture et traitement du fichier parquet source par fragments pour optimiser l'utilisation de la mémoire
3. Création de nœuds typés pour chaque entité (Produit, Marque, Catégorie, Ingrédient, etc.)
4. Établissement des relations entre les nœuds selon différents types (HAS_BRAND, CONTAINS, CONTAINS_ADDITIF, etc.)
5. Génération d'embeddings pour les produits afin de permettre la recherche sémantique
6. Sauvegarde du graphe finalisé au format pickle pour une utilisation ultérieure

Le graphe résultant contient 94 802 produits canadiens et présente une structure riche en relations. Les tableaux 3 et 4 détaillent sa composition par type de nœuds et de relations.

Type de nœud	**Nombre**
Product	94 802
Brand	10 478
Category	7 801
Ingredient	44 617
Label	1 827
Additif	347
Allergen	120
Country	130
Nutriment	14
Total	**160 136**

TABLE 3 – Distribution des nœuds par type

Type de relation	**Nombre**
HAS_NUTRIMENT	697 936
CONTAINS	426 820
HAS_CATEGORY	104 451
SOLD_IN	101 491
HAS_LABEL	65 159
HAS_BRAND	43 149
CONTAINS_ADDITIF	41 146
CONTAINS_ALLERGEN	19 323
Total	**1 499 475**

TABLE 4 – Distribution des relations par type

Ces tableaux représentent l'intégralité des types de nœuds et de relations implémentés dans le graphe.

Cette structure relationnelle permet de modéliser efficacement les connexions entre produits, ingrédients, additifs et autres entités pertinentes pour répondre aux questions des utilisateurs.

11.2 Génération des embeddings

Pour permettre la recherche sémantique dans le graphe, chaque produit est doté d'un embedding vectoriel. Ces vecteurs sont générés par la fonction `get_product_embedding_text()` qui combine plusieurs attributs du produit en un texte descriptif structuré :

```
text_parts = []
if product_name:
    text_parts.append(f"Nom: {product_name}")
if pd.notna(product.get('brands')):
    text_parts.append(f"Marque: {product.get('brands')}")
if pd.notna(product.get('categories')):
    text_parts.append(f"Catégories: {product.get('categories')}")
# [...autres attributs selon disponibilité...]
text = " ".join(text_parts)
```

Listing 3 – Génération du texte pour l'embedding

Ce texte est ensuite transmis au modèle SentenceTransformer (`all-MiniLM-L6-v2`) qui génère un vecteur d'embedding de dimension 384. Cette représentation vectorielle est stockée comme propriété du nœud produit dans le graphe et permet d'effectuer des recherches par similarité sémantique.

11.3 Interface avec NetworkX

L'agent NetworkX est un agent conversationnel implémenté avec SmolAgents, configuré spécifiquement pour interroger le graphe de connaissances construit avec NetworkX.

Contrairement à DuckDB qui utilise SQL ou Neo4j qui utilise Cypher, NetworkX est une bibliothèque Python pure qui n'a pas de langage de requête dédié. Pour faciliter l'interaction entre l'agent et NetworkX, j'ai développé une couche d'abstraction (NetworkXQueryTool) qui permet à l'agent NetworkX de soumettre des requêtes standardisées au format JSON, comme dans cet exemple :

```
{
    "operation": "search_by_relation",
    "relation_type": "CONTAINS_ADDITIF",
    "relation_target": "E330",
    "limit": 5
}
```

Listing 4 – Exemple de requête JSON pour NetworkX

Ces requêtes JSON sont ensuite interprétées et traduites en appels à l'API Python native de NetworkX. Cette approche, bien que simplifiant l'interaction avec le graphe, s'est révélée être une limitation importante pour l'agent, qui ne pouvait pas exploiter pleinement la flexibilité qu'offre le parcours de graphe.

12 Évaluation comparative entre DuckDB et NetworkX

Pour évaluer objectivement les performances des approches relationnelle et graphe, j'ai développé un script d'évaluation `evaluate_nx.py` qui compare les deux agents sur le même ensemble de questions-réponses (fichier `qa_pairs.json`), en français et en anglais.

Le processus d'évaluation suit ces étapes :

1. Configuration de deux agents distincts : un agent DuckDB et un agent NetworkX
2. Test systématique des deux agents sur chaque question du jeu de test
3. Évaluation hybride des réponses combinant :
   — Métriques automatiques (BLEU, ROUGE) pour la similarité textuelle

— Évaluation par LLM (Claude 3.5 Haiku) pour la pertinence sémantique

4. Mesure du temps de réponse et du taux de réussite/échec
5. Génération d'un rapport comparatif détaillé

Les métriques d'évaluation utilisées sont :

— **Taux de réussite (%)** : Pourcentage de questions pour lesquelles l'agent a fourni une réponse jugée sémantiquement correcte selon l'évaluation du LLM et des métriques automatiques. Une réponse est considérée comme correcte lorsqu'elle atteint un score combiné minimum de pertinence.
— **Taux d'échec technique (%)** : Pourcentage de questions pour lesquelles l'agent a rencontré une erreur technique empêchant la génération d'une réponse satisfaisante, ou pour lesquelles l'agent a explicitement indiqué ne pas pouvoir répondre (messages contenant « Désolé » ou « Sorry »).
— **Temps moyen/médian (s)** : Durée moyenne et médiane pour traiter une question, de la soumission à la génération de la réponse finale.
— **Score combiné** : Pondération de scores LLM (70%), BLEU-2 (15%) et ROUGE-L (15%) évaluant la qualité globale de la réponse.
— **Métriques détaillées** : Scores individuels LLM, BLEU-2 et ROUGE-L permettant d'analyser plus finement la performance des agents.

Il est important de noter que le taux de réussite et le taux d'échec ne sont pas nécessairement complémentaires (leur somme n'est pas toujours égale à 100%). Une question peut ne pas être classée comme un succès sans pour autant constituer un échec technique, notamment lorsque l'agent fournit une réponse syntaxiquement valide mais dont le contenu est partiellement ou totalement incorrect.

13 Résultats de l'évaluation

J'ai conduit une évaluation complète portant sur 100 questions, à la fois en français et en anglais. Les résultats sont présentés dans le tableau 5.

Métrique	**Français**		**Anglais**		**Différence**	
	DuckDB	**NetworkX**	**DuckDB**	**NetworkX**	**FR**	**EN**
Taux de réussite (%)	43,00	17,00	50,00	26,00	26,00	24,00
Taux d'échec (%)	21,00	48,00	17,00	49,00	27,00	32,00
Temps moyen (s)	26,37	26,69	26,58	29,94	0,32	3,36
Temps médian (s)	26,21	27,65	25,03	28,60	1,44	3,57
Score combiné moyen	0,34	0,19	0,37	0,25	0,15	0,13
Score LLM moyen	0,42	0,23	0,46	0,30	0,19	0,16
BLEU-2 moyen	0,0815	0,0398	0,0426	0,0331	0,0417	0,0095
ROUGE-L moyen	0,1767	0,1144	0,1077	0,1023	0,0624	0,0054

TABLE 5 – Résultats de l'évaluation sur 100 questions en français et en anglais

14 Interprétation des résultats

Le constat le plus frappant est la performance globalement décevante des deux agents. L'agent DuckDB, malgré ses meilleurs résultats, ne parvient à répondre correctement qu'à la moitié des questions au mieux (43% en français, 50% en anglais). L'agent NetworkX présente des résultats encore plus préoccupants, avec un taux de réussite de seulement 17% en français et 26% en anglais, ainsi qu'un taux d'échec avoisinant les 50%. Ces performances sont nettement insuffisantes pour une application pratique.

Plusieurs tendances se dégagent des résultats :

- **Écart significatif entre approches** : L'agent DuckDB surpasse systématiquement l'agent NetworkX sur toutes les métriques, avec un écart d'environ 24-26 points de pourcentage en taux de réussite.
- **Différences linguistiques** : Les deux agents obtiennent de meilleurs résultats en anglais qu'en français, avec une différence plus marquée pour NetworkX (26% contre 17%) que pour DuckDB (50% contre 43%).
- **Temps de réponse** : Les temps de réponse sont comparables en français, mais l'agent NetworkX se montre sensiblement plus lent en anglais (+3,36 secondes en moyenne).
- **Qualité des réponses** : Les scores combinés et individuels (LLM, BLEU, ROUGE) confirment la supériorité de l'agent DuckDB en termes de pertinence et précision.

L'analyse des journaux d'exécution révèle plusieurs problèmes récurrents : échecs sur des questions apparemment simples, réponses sous forme de code plutôt que de langage naturel, et réponses pertinentes mais structurellement différentes des références (pénalisées par les métriques d'évaluation automatiques).

14.1 Analyse des facteurs limitant les performances de NetworkX

La différence marquée entre les deux agents s'explique principalement par leur implémentation :

- L'agent DuckDB génère directement des requêtes SQL, un format bien maîtrisé par les LLMs actuels et adapté à leur paradigme de génération de texte.
- En revanche, l'agent NetworkX souffre de trois limitations principales :
  - **Architecture contraignante** : La couche d'abstraction JSON créée limite considérablement la flexibilité de l'agent, le restreignant à un ensemble prédéfini d'opérations qui réduit sa capacité à traiter des questions complexes.
  - **Familiarité des LLM** : Les modèles de langage sont généralement bien entraînés sur SQL et Cypher (Neo4j), mais beaucoup moins sur l'API Python de NetworkX, compliquant la génération de requêtes pertinentes.
  - **Méthode d'interaction** : L'interface abstraite imposée empêche l'agent d'exploiter pleinement les capacités natives de parcours de graphe, contrairement à une génération directe de code Python NetworkX.

Lors d'expérimentations préliminaires avec Neo4j, l'agent semblait mieux à même de générer des requêtes Cypher, suggérant qu'une implémentation permettant à l'agent de formuler librement des requêtes de parcours de graphe aurait probablement donné de meilleurs résultats. Malheureusement, des limitations techniques m'ont contraint à migrer vers NetworkX, avec les conséquences observées sur les performances.

15 Limites et perspectives d'amélioration

Plusieurs pistes pourraient améliorer significativement les performances des deux agents, particulièrement l'agent NetworkX :

- **Modèles plus performants** : L'utilisation de Claude 3.7 ou GPT-4 pourrait améliorer la compréhension des questions et la génération de requêtes.
- **Augmentation du paramètre `max_steps`** : Limité à 5 pour réduire les coûts d'utilisation de l'API Claude Haiku, ce paramètre définit le nombre maximal d'étapes que l'agent peut effectuer pour résoudre une tâche. Cette limite restreint la capacité de l'agent à explorer et raffiner ses approches face à des requêtes complexes, surtout lorsque plusieurs étapes intermédiaires sont nécessaires.
- **Liberté de génération pour NetworkX** : Permettre à l'agent de générer directement des requêtes de parcours de graphe plutôt que de le restreindre à des méthodes prédéfinies.

— **Exemples de réponses** : Fournir des exemples clairs de format de réponse attendu pourrait améliorer la qualité et la cohérence des sorties.
— **Mise en œuvre avec Neo4j** : Malgré les défis techniques rencontrés, l'utilisation d'une base de données graphe dédiée comme Neo4j, avec son langage de requête Cypher, pourrait offrir de meilleures performances que l'approche NetworkX en mémoire.
— **Évaluation manuelle** : Une évaluation humaine d'un sous-ensemble de réponses permettrait d'identifier plus précisément les forces et faiblesses de chaque approche.

16 Conclusion de la partie 2

Cette exploration d'une approche basée sur les graphes pour l'interrogation de la base Open Food Facts a mis en lumière à la fois le potentiel et les défis de cette méthode. Si les résultats empiriques favorisent clairement l'approche relationnelle avec DuckDB, il serait prématuré de conclure à la supériorité intrinsèque de cette dernière.

Les performances décevantes de l'agent NetworkX semblent davantage liées à des choix d'implémentation qu'à des limitations fondamentales de l'approche par graphe. La recherche récente continue de souligner les avantages potentiels des graphes de connaissances pour enrichir le contexte sémantique des requêtes en langage naturel.

Cette expérience m'a permis de tirer plusieurs enseignements importants :
— **Impact du choix du modèle** : L'utilisation d'un LLM performant comme Claude 3.5 Sonnet améliore considérablement la qualité des requêtes générées et des réponses fournies.
— **Flexibilité des requêtes** : Les agents LLM sont plus performants lorsqu'ils peuvent générer librement leurs requêtes dans un langage qu'ils maîtrisent déjà (SQL, Cypher) plutôt que d'être contraints par une interface abstraite.
— **Importance de la documentation** : Une documentation détaillée de la structure des données et des exemples de requêtes est essentielle pour guider efficacement les agents.
— **Structure des données** : Il est préférable de reconstruire directement dans la base de données les relations entre les champs selon leur taxonomie, plutôt que de s'appuyer uniquement sur une documentation externe.

L'ensemble du code source de ce projet, incluant les scripts de création du graphe et d'évaluation, est disponible publiquement dans le dépôt GitHub https://github.com/boisalai/ift-6005.

Références

[1] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. Text-to-SQL Empowered by Large Language Models : A Benchmark Evaluation. *arXiv preprint arXiv :2308.15363*, 2023. URL https://arxiv.org/abs/2308.15363.

[2] Ali Mohammadjafari, Anthony S Maida, and Raju Gottumukkala. From Natural Language to SQL : Review of LLM-based Text-to-SQL Systems. *arXiv preprint arXiv :2410.01066*, 2024. URL https://arxiv.org/abs/2410.01066.

Annexe A : Exemple d'évaluation d'une requête utilisateur

Cette annexe illustre le processus d'évaluation d'une requête posée à l'agent conversationnel. Elle détaille les différentes étapes suivies, depuis l'analyse sémantique des colonnes pertinentes jusqu'à la génération et l'évaluation de la réponse, en mettant en évidence la méthodologie employée pour mesurer la performance du système.

Question utilisateur : « What food products without additives are available in the database ? »

Étape 1 : Définition de la question et référence dans le jeu de test

Cette question correspond à la première entrée dans le fichier `qa_pairs.json`, structurée comme suit :

```
{
  "column": "additives_n",
  "sql": "SELECT code, product_name FROM products WHERE additives_n = 0",
  "questions": {
    "fr": "Quels sont les produits alimentaires sans additifs disponibles dans
        la base de données?",
    "en": "What food products without additives are available in the database?"
  },
  "answers": {
    "fr": "La base de données contient 5843 produits sans additifs, incluant des
         produits comme le sirop d'érable biologique du Vermont, le lait faible
        en gras, l'agave bleu biologique, et le lait de coco.",
    "en": "The database contains 5843 products without additives, including
        items such as organic Vermont maple syrup, low-fat milk, organic blue
        agave, and coconut milk."
  }
},
...
```

Étape 2 : Analyse sémantique des colonnes pertinentes

Le système utilise FAISS pour comparer l'embedding de la question avec ceux des descriptions de colonnes :

```
relevant_columns = self._search_relevant_columns(question)
```

Listing 5 – Recherche sémantique des colonnes pertinentes

Cette recherche identifie 5 colonnes potentiellement pertinentes avec leurs scores de similarité :

- `unknown_ingredients_n` (score : 0,656)
- `additives_tags` (score : 0,638)
- `ingredients_original_tags` (score : 0,622)
- `ingredients_without_ciqual_codes` (score : 0,612)
- `data_quality_info_tags` (score : 0,563)

Étape 3 : Préparation du prompt pour l'agent

Les informations sur ces colonnes sont préparées et stockées dans `columns_info` :

```
columns_info = []
for col in relevant_columns:
    if col['similarity'] > 0.5:
        column_section = [
            f"Column: {col['name']}\n"
            f"Type: {col['type']}\n"
            f"Description: {col['description']}\n"
            f"Examples of values: {', '.join(map(str, col['examples'][:3]))}"
        ]

        if 'common_queries' in col and col['common_queries']:
            column_section.append(f"Query examples:")
            for query in col['common_queries']:
                column_section.append(
                    f"# {query.get('description', '')}:\n"
                    f"{query.get('sql', '')}"
                )

        columns_info.append("\n".join(column_section))
```

Listing 6 – Préparation des informations sur les colonnes pertinentes

Ces informations sur les colonnes pertinentes sont insérées des notes additionnelles pour l'agent comme ceci :

```
additional_notes = dedent(
    """\
    You are a helpful assistant that answers questions about food products
    using the Open Food Facts database.

    POTENTIALLY RELEVANT COLUMNS:
    The following columns have been identified through semantic search as
        potentially relevant,
    with their similarity scores (higher means more likely relevant):

    {columns_text}

    SEARCH SEQUENCE RULES:
    1. ALWAYS start with database queries using the most relevant columns
    2. If initial query fails, try alternative database queries with different
        columns or approaches
    3. Only if database queries are unsuccessful, search the Canada Food Guide
    4. Document EVERY attempt in the steps array, including failures
    5. Never skip straight to Food Guide without trying database first
    6. Always include the source of the information in the answer ("Open Food
        Facts" or "Canada Food Guide")
    7. Always respond in the same language as the question (French or English)

    RESPONSE FORMAT REQUIREMENTS:
    1. Provide ONLY the natural language answer to the user's question
    2. Maximum response length: 200 characters
    3. DO NOT include SQL queries, code snippets, or technical details
    4. DO NOT explain your reasoning or methodology
    5. Respond in the same language as the question (French or English)
    6. DO mention the source of information ("Open Food Facts" or "Canada Food
        Guide")

    Please follow these rules to ensure a consistent and effective search
        strategy.
    """
).format(columns_text=columns_text)
```

Listing 7 – Préparation des notes additionnelles pour l'agent

Étape 4 : Création et exécution de l'agent

L'agent est créé de cette façon, où `query_db` et `search_web_agent` sont des outils d'agent définis pour interroger la base de données et le site Web du Guide alimentaire canadien.

```
# Création du modèle LLM
model = LiteLLMModel(
    model_id="ollama/llama3.1:8b-instruct-q8_0",
    api_base="http://localhost:11434",
    num_ctx=8192
)

# Create agent
agent = CodeAgent(
    tools=[query_db],
    model=model,
    managed_agents=[search_web_agent],
    additional_authorized_imports=["json"],
    verbosity_level=LogLevel.INFO,
)
```

Listing 8 – Création de l'agent

L'agent est appelé avec le prompt enrichi :

```
start_time = time.time()
agent_response = agent.run(
    question,
    additional_args={"additional_notes": additional_notes},
)
response_time = time.time() - start_time
```

Listing 9 – Exécution de l'agent avec les notes additionnelles

Étape 5 : Génération de la réponse

L'agent analyse les informations et génère la réponse suivante : « According to Open Food Facts database, additive-free products include natural foods like blueberries and pistachios, basic staples like spaghetti and rice, and beverages like coconut water and coffee. »

Étape 6 : Traçage des étapes de l'agent

Le système enregistre toutes les étapes suivies par l'agent :

```
# Extrait les étapes depuis la mémoire de l'agent
if hasattr(agent, 'memory') and hasattr(agent.memory, 'steps'):
    steps_sequence = []

    for step in agent.memory.steps:
        # Analyse chaque étape et outil utilisé
        if not hasattr(step, 'tool_calls') or not step.tool_calls:
            continue

        tool_call = step.tool_calls[0]

        # Enregistre les requêtes SQL, recherches web et autres actions
        if tool_call.name == "python_interpreter" and isinstance(tool_call.
            arguments, str):
            code = tool_call.arguments
            step_data = {}

            # Détermine le type d'action (requête SQL, recherche web, etc.)
            if "query_db" in code:
                sql_query = self._extract_sql_from_code(code)
                # Enregistre l'information sur la requête SQL
            elif "search_food_guide" in code:
                # Enregistre l'information sur la recherche web
```

Listing 10 – Traçage des étapes de l'agent

Ces étapes permettent de suivre la séquence de recherche (d'abord la base de données, puis le guide alimentaire si nécessaire).

Étape 7 : Évaluation des métriques

Trois métriques principales sont calculées :

1. Précision d'exécution (EX) : Comparer la requête générée avec la référence

```
def _calculate_sql_accuracy(self, response_data: dict, qa_pair: Dict[str, Any])
    -> float:
    agent_sql = response_data.get('sql_query')

    # Exécuter les requêtes
    reference_results = self.execute_query(qa_pair['sql'])
    agent_results = self.execute_query(agent_sql)

    # Calculer les métriques individuelles
    query_present = 1.0 if agent_sql else 0.0
    execution_success = float(agent_results.success)
    results_match = 0.0

    # Comparer les résultats
    if reference_results.success and agent_results.success:
        # Calcul de la similarité entre les deux ensembles de résultats
```

Listing 11 – Calcul de la précision d'exécution

2. Précision sémantique (PS) : Comparer la réponse générée avec la référence

Je demande à un modèle LLM de comparer les deux réponses pour évaluer leur similarité sémantique.

Les deux réponses sont :

— **Réponse attendue** : "The database contains 5843 products without additives, including items such as organic Vermont maple syrup, low-fat milk, organic blue agave, and coconut milk."
— **Réponse de l'agent** : "According to Open Food Facts database, additive-free products include natural foods like blueberries and pistachios, basic staples like spaghetti and rice, and beverages like coconut water and coffee."

```
def _calculate_semantic_accuracy(self, response_data: dict, qa_pair: Dict, lang:
    str) -> float:
    # Utiliser un LLM pour évaluer la similarité sémantique
    prompt = dedent(f"""\
    Compare these two responses and rate their semantic similarity from 0 to 1:
    Response #1: {qa_pair['answers'][lang]}
    Response #2: {agent_response}
    """)

    # Le modèle LLM évalue et retourne un score de similarité
```

Listing 12 – Calcul de la précision sémantique

3. Respect de séquence (RS) : Vérifier que l'agent a suivi le bon ordre des sources

```
def _evaluate_search_sequence(self, response_data: dict) -> dict:
    steps = response_data.get('steps', [])

    # Initialiser les compteurs
    db_attempts = []
    web_attempt = None
    sequence_respected = True

    # Vérifier l'ordre des étapes
    for step in steps:
        if step['action'] in ['database_query', 'alternative_query']:
            db_attempts.append(step)
            # Vérifier si une recherche web a déjà eu lieu (violation)
            if web_attempt is not None:
                sequence_respected = False
                break
        elif step['action'] == 'food_guide_search':
            web_attempt = step
```

Listing 13 – Évaluation du respect de séquence

Étape 8 : Compilation des résultats

Les résultats de l'évaluation sont compilés :

```
# Créer un objet EvaluationResult contenant toutes les informations
return EvaluationResult(
    question_id=qa_pair.get('id', 0),
    language=lang,
    question=question,
    expected_answer=qa_pair['answers'][lang],
    agent_answer=agent_answer,
    metrics=metrics,
    expected_sql=qa_pair.get('sql', ''),
    agent_sql=agent_sql
)
```

Listing 14 – Compilation des résultats

Résultats finaux

L'évaluation finale montre :

- Précision d'exécution (EX) : 0,00% (l'agent n'a pas généré de requête SQL valide)
- Précision sémantique (PS) : 80,00% (forte similarité entre les réponses)
- Respect de séquence (RS) : 100,00% (protocole de recherche respecté)
- Temps de réponse moyen (TR) : 45,95 secondes

Cet exemple illustre le parcours complet, démontrant comment l'agent analyse la question, génère une réponse et comment le système évalue objectivement cette réponse selon plusieurs dimensions.