

Atos

Data Pipeline IA

Documentation

DIEZ PECOSTE Raphael / PAPINI Julien

01/08/2025

Table des matières

TABLE DES MATIERES	2
PRESENTATION DU PROJET	3
I. INSTALLATION ET PREPARATION DE L'ENVIRONNEMENT	4
I.A PREREQUIS	4
I.B INSTALLATION DES DEPENDANCES	4
I.C CONFIGURATION DE L'ENVIRONNEMENT	5
I.D PARAMETRES NECESSAIRES POUR SNOWFLAKE.....	5
I.E UTILISATION DE GROQ	6
II. STRUCTURE DU REPERTOIRE.....	7
II.A RACINE DU PROJET	7
II.B CŒUR DU PIPELINE – SRC /	9
II.C CONFIGURATION – CONFIG /	13
II.D DOCUMENTATION – DOCS /	14
II.E SPECIFICATIONS ET PROMPTS – FORMATS /	14
II.F ENTREES UTILISATEUR – INPUTS /	15
II.G RESULTATS GENERES – OUTPUTS/.....	16
III. FONCTIONNEMENT DU PIPELINE	19
III.A SAISIE DES ENTREES DE L'UTILISATEUR.....	19
III.B INITIALISATION DU PROJET – NODE_INIT_PROJECT.....	23
III.C INGESTION DES METADONNEES – NODE_INGESTION	24
III.D GENERATION DU SCHEMA SOURCES – NODE_GENERATE_SOURCE_YAML	33
III.E PREPARATION DES SPECIFICATIONS DE FICHIERS - NODE_PREPARE_FILE_SPECS	37
III.F GENERATION DES FICHIERS – NODE_GENERATE_FILES.....	40
III.G CREATION DES FICHIERS – NODE_CREATE_FILE	43
III.H SAUVEGARDE DES GENERATIONS – NODE_SAVE_OUTPUTS.....	47
III.I FIN DE PROCESSUS – NODES_END_PROCESS	48

Présentation du Projet

Ce module permet de générer automatiquement un pipeline de transformation de données DBT à partir d'un fichier de spécifications métier (CSV ou Excel), en s'appuyant sur un modèle de langage (LLM) hébergé via l'API de Groq. Il vise à **réduire les efforts manuels** liés à l'écriture de code SQL et à la structuration de projets DBT.

Le pipeline produit automatiquement :

- 1) La **structure logique du projet DBT** (modèles staging, intermediate, marts),
- 2) Le **code SQL associé**, incluant les sources, tests, documentation,
- 3) Une **arborescence complète**, prête à l'emploi.

L'ensemble est orchestré à l'aide de **LangGraph**, un framework de graphes d'exécution, permettant un contrôle précis des étapes du pipeline, avec la possibilité d'itérations, de validation intermédiaire, ou de reprises sur erreur.

Terme	Définition
LLM	Large Language Model, utilisé pour générer automatiquement du code.
DBT	Data Build Tool, framework de transformation de données analytique.
Mart	Table finale exposée à l'utilisateur métier.
Specs	Spécifications décrivant les colonnes et les règles métier.
LangGraph	Framework d'orchestration par graphe pour chaîner les étapes du pipeline.

Terminologie utile afin de lire cette documentation

I. Installation et préparation de l'environnement

Ce chapitre guide pas à pas l'installation, la configuration et la collecte des informations nécessaires pour exécuter le générateur de projet DBT.

Il est conçu pour un utilisateur novice qui découvre l'outil.

I.A Prérequis

Avant de commencer, assurez-vous d'avoir :

- **Python 3.12+** installé
Vérifiez avec : `python --version`
- **pip** (installé avec Python).
- Une clé API **Groq** (voir I.E).
- **Git** (optionnel si vous clonez le projet depuis un dépôt).
- **Accès internet** (pour installer les dépendances et accéder à l'API LLM).
- **Compte Snowflake** (si vous utilisez le générateur avec une base Snowflake).

I.B Installation des dépendances

Toutes les dépendances sont listées dans `requirements.txt`.

Pour les installer, deux options :

I.B.1 Manuel (via terminal)

```
# Créer un environnement virtuel
python -m venv venv
source venv/bin/activate    # Linux / macOS
venv\Scripts\activate       # Windows

# Installer les dépendances
pip install -r requirements.txt
```

I.B.2 Automatique (via la CLI)

```
# Tout est automatisé avec une commande
python main.py install
```

Cette commande crée un environnement virtuel et installe automatiquement les dépendances.

💡 *Image à insérer : capture du terminal montrant l'exécution de `python main.py install`.*

I.C Configuration de l'environnement

Une fois les dépendances installées, il faut préparer deux fichiers essentiels :

1. **.env** (dans le dossier racine du projet)
→ Contient les clés API et identifiants Snowflake.
2. **profiles.yml** (dans ~/ .dbt /)
→ Fichier de configuration utilisé par DBT pour se connecter à Snowflake.

Vous pouvez les remplir **manuellement** ou laisser la commande :

```
# Tout est automatisé avec une commande  
python main.py init
```

Cette commande ouvre des **fenêtres interactives** (pop-ups) qui vous guideront pas à pas. Chaque champ est demandé, avec gestion des mots de passe masqués.

💡 *Image à insérer : pop-up Tkinter demandant SNOWFLAKE_USER ou SNOWFLAKE_PASSWORD.*

I.D Paramètres nécessaires pour Snowflake

Pour que DBT et le générateur se connectent à votre instance Snowflake, vous devez récupérer certains paramètres. La plupart peuvent se trouver dans « Account Details », voir dans l'[ANNEXE X](#).

Voici où les trouver :

Paramètre	Description	Où le trouver
SNOWFLAKE_ACCOUNT	Identifiant de votre compte Snowflake. Souvent au format xyz12345.eu-west-1 ou organisation.region.cloud	Dans l'URL Snowflake (ex. https://xyz12345.eu-west-1.snowflakecomputing.com).
SNOWFLAKE_USER	Nom d'utilisateur pour se connecter.	Fourni lors de la création du compte Snowflake.
SNOWFLAKE_PASSWORD	Mot de passe lié à l'utilisateur.	Déterminé à la création du compte. Peut être réinitialisé par l'admin.

SNOWFLAKE_ROLE	Rôle utilisé par DBT (ex. ACCOUNTADMIN, SYSADMIN, ANALYST). Attention de bien utiliser un role associé à la database.	Visible dans l'interface Snowflake, dépend de vos droits.
SNOWFLAKE_WAREHOUSE	Entrepôt virtuel à utiliser (compute).	Dans Snowflake → onglet "Warehouses".
SNOWFLAKE_DATABASE	Base de données cible (ex. HIMALAYAN_EXPEDITIONS).	Dans Snowflake → onglet "Databases".
SNOWFLAKE_SCHEMA	Schéma à utiliser dans la base.	Dans Snowflake → onglet "Databases" → choisir un schéma.

I.E Utilisation de Groq

Le projet utilise **Groq** pour la génération automatique de plans et de modèles DBT à partir de descriptions en langage naturel.

- Vous devez disposer d'une clé API Groq, disponible sur : <https://console.groq.com/>
- La clé est à renseigner dans votre fichier .env:

```
GROQ_API_KEY=sk-xxxxxx
```

- Le projet s'appuie sur le modèle **llama-3.3-70b-versatile**, qui offre un excellent compromis entre rapidité et qualité des générations. Le modèle est modifiable dans .env

Attention à la **Rate limits** : selon votre abonnement Groq, vous disposez d'un quota maximum de requêtes par minute. J'ai donc dû changer des modèles à chaque fois que mes prompts augmentaient en taille. Pour plus de détail sur les rate limits, voir **ANNEXE X**.

II. Structure du Répertoire

Le projet est organisé de manière modulaire autour de trois axes :

- **Le cœur applicatif** (src/) contenant la logique du graphe et les appels LLM,
- **Les fichiers de configuration, d'entrée et de sortie,**
- **La documentation** et les fichiers d'orchestration externes.

Voici l'arborescence complète :

```
DataPipeline_IA/
├── config/          # Contient les variables globales du projet
├── docs/            # Documentation : schémas, visuels, fichiers explicatifs
├── formats/         # Templates et prompts
├── inputs/          # Fichiers de spécifications fournis par l'utilisateur
└── outputs/         # Résultats générés automatiquement

├── src/             # Cœur du projet
│   ├── core/          # Modules de génération
│   ├── graph/         # Définition du graphe LangGraph
│   ├── test/           # Scripts et notebooks de test
│   └── utils/          # Fonctions utilitaires partagées

├── .env              # Fichier de configuration des variables sensibles
├── app.py            # Application Streamlit
├── main.py           # Point d'entrée CLI pour exécuter une task
├── README.md         # Page d'accueil du projet
└── backups/          # Sauvegarde d'un projet intéressant
└── requirements.txt  # Dépendances Python
```

Architecture globale du projet avec annotation

Pour aller plus en détail dans l'architecture du projet, nous allons parcourir dans un premier temps les fichiers à la racine du projet, puis dans un second temps les dossiers colorés qui représentent chacun une partie importante dans l'ensemble du code.

II.A Racine du projet

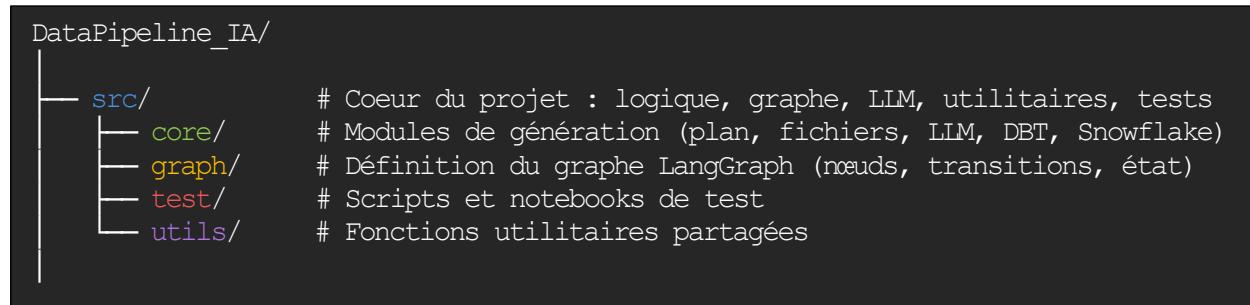
À la racine, on retrouve plusieurs fichiers essentiels, et un dossier backup dont nous aurons un aperçu détaillé du contenu dans la partie [Résultat générés – outputs/](#). Voici un tableau qui résume le rôle de chaque fichier de la racine :

Fichier	Rôle
main.py	<p>Point d'entrée principal du programme en mode CLI. Il orchestre les différentes tâches et sert d'interface directe en ligne de commande pour les utilisateurs avancés. Tâches disponibles :</p> <ul style="list-style-type: none"> - Install : permet l'installation des dépendances - Run : permet l'exécution de la pipeline sans interface GUI - Backup : permet de sauvegarder un projet particulier - Help : permet l'affichage d'une aide si besoin - Clean : permet le nettoyage des fichiers caches et des outputs qui ne sont plus nécessaires (attention à faire un backup avant !)
app.py	<p>Interface principale basée sur Streamlit. Elle offre une expérience interactive avec en plus l'affichage des logs et d'un graphe représentant l'avancée de la génération en temps réel. La restitution finale du projet DBT se fait aux endroits sélectionnés par l'utilisateur. Il peut également télécharger les données de la génération (logs, state, plan et contenu des fichiers générés)</p>
.env	<p>Fichier de configuration des variables d'environnement sensibles (connexion Snowflake, clés API LLM, etc.) mais également les informations rentrées par l'utilisateur avec la tâche <code>python main.py init</code> (base de données choisie, profil DBT, etc.).</p>
.pylintrc	<p>Configuration de l'outil de linting pour assurer une homogénéité du code.</p>
requirements.txt	<p>Liste des dépendances Python nécessaires au projet. A installer avec <code>python main.py install</code> ou <code>pip install -r requirements.txt</code></p>
backups/	<p>Stocke les états complets d'exécutions passées, permettant de revenir sur des résultats antérieurs. Se crée via la commande : <code>python main.py backup</code> Chaque sous-dossier porte un nom basé sur le projet (exemple <code>projet_test</code>) et la date/heure, ce qui donne par exemple : <code>projet_test_22_20250819_152600/</code> Ce mécanisme de sauvegarde complète celui des outputs/ et garantit une traçabilité totale, même après un nettoyage (<code>python main.py clean</code>).</p>

Tableau récapitulatif des fichiers présents dans la racine (DataPipeline_IA)

II.B Cœur du pipeline – `src/`

Le dossier `src/` constitue le cœur du projet. Il regroupe l'ensemble des composants permettant d'orchestrer la génération d'un projet DBT à partir d'une description utilisateur, en s'appuyant sur un LLM et un graphe d'exécution construit avec **LangGraph**.



Architecture de la partie source du projet avec annotation

II.B.1 Modules de génération - `src/core/`

Ce dossier centralise toutes les **fonctions métier du pipeline** : appel au LLM, construction du plan, écriture de fichiers, interactions avec Snowflake.

Fichier	Rôle
<code>generate_dbt_project.py</code>	Lance la création complète d'un projet DBT. Il initialise l'environnement, charge les formats attendus, construit le graphe LangGraph, et exécute le pipeline. C'est lui qui relie toutes les briques (planification, génération des fichiers, sauvegarde des outputs).
<code>llm.py</code>	Encapsule les appels au LLM (ici via l'API Groq, avec le modèle llama-3.1-8b-instant).
<code>plan_fct.py</code>	Gère l'extraction de plans depuis la première génération du LLM (passerelle entre les sorties LLM et les représentations JSON internes).
<code>write_files.py</code>	Écrit physiquement les fichiers générés par le pipeline (modèles, sources, tests dbt). Il reçoit en entrée les spécifications et les sauvegarde dans la structure de projet DBT (sous format .sql, .yaml, etc.).
<code>snowflake_fct.py</code>	Permet d'ouvrir une connexion Snowflake et d'ingérer les métadata provenant de la base de données hébergée dessus.

Tableau récapitulatif des fichiers présents dans `src/core/`

Il contient également toute la logique des tâches, utilisable dans la console, qui permet de gérer le projet plus ergonomicement. Toutes ces tâches sont regroupées dans `src/core/tasks/` :

Fichier	Rôle
<code>task_backup.py</code>	Gère la sauvegarde d'un projet DBT généré. Crée une copie horodatée du projet sous <code>backups/<project_name>_<YYYYMMDD_HHMMSS>/</code> . Utilisé via <code>python main.py backup <project_name></code> .
<code>task_clean.py</code>	Assure le nettoyage de l'environnement : suppression des dossiers <code>__pycache__</code> et du répertoire <code>outputs/</code> , puis recréation d'un <code>outputs/</code> vide. Utilisé via <code>python main.py clean</code> .
<code>task_run.py</code>	Lance la pipeline de génération DBT en mode console. Demande les paramètres d'entrée (chemin de sortie, nom de projet, profil DBT, fichier de spécification) puis exécute LangGraph en tâche de fond avec logs en streaming. Utilisé via <code>python main.py run</code> .
<code>task_init_env.py</code>	Initialise l' environnement de travail en deux étapes : <ol style="list-style-type: none"> 1. Installation des dépendances depuis <code>requirements.txt</code>. 2. Configuration des fichiers <code>.env</code> (clés API, credentials Snowflake) et <code>profiles.yml</code> (profil DBT). Les entrées sensibles sont masquées et confirmées avant écrasement. Utilisé via <code>python main.py init</code> .

Tableau récapitulatif des fichiers présents dans `src/core/tasks`

II.B.2 Orchestration LangGraph - `src/graph/`

Le dossier `graph/` contient la définition du **pipeline LangGraph**, utilisé pour orchestrer chaque étape du projet, tout en gardant un état partagé entre les nœuds. Il contient les sous dossiers `edges/` et `nodes/` qui comme leur nom l'indique contiennent respectivement les arêtes et les noeux du graphe.

a) *Fichiers principaux*

- `build_graph.py` : définit la structure du graphe (les nœuds et les transitions conditionnelles). Il organise l'ordre des étapes (ingestion, génération de plan, préparation des fichiers, génération, écriture, sauvegarde). Le graphe est disponible en [ANNEXE](#)
- `state.py` : définit la structure du **state partagé entre les nœuds** (via un TypedDict). Ce dictionnaire contient notamment la description utilisateur, le plan généré, les fichiers attendus, l'état de la génération, et les données ingérées. L'exemple du contenu du `state` en fin de processus est disponible en [ANNEXE](#)

b) *Transitions conditionnelles – graph/edges/*

Ces fichiers contiennent des fonctions qui orientent dynamiquement l'exécution du graphe selon l'état ou la configuration (`load_existing_plan`, `should_generate_object`, etc.).

Fichier	Rôle
<code>edge_select_plan.py</code>	Choix entre génération ou chargement d'un plan DBT sauvegardé d'un projet précédent.
<code>edge_select_files.py</code>	Choix entre génération ou chargement du contenu de fichiers sauvegardé d'un projet précédent.
<code>edge_verify_plan.py</code>	Vérification de validité du plan généré par LLM, dans le cas d'échec on regénère selon le nombre d'échec autorisé (3 par défaut)

Tableau récapitulatif des fichiers présents dans `src/graph/edges`

c) *Étapes du pipeline – graph/nodes/*

Chaque fichier ici représente un **nœud du graphe**, chargé d'une étape précise dans la génération automatique du pipeline DBT. Certains nœuds sont appelés une seule fois (par ex : `node_ingestion`), d'autres plusieurs fois avec des paramètres différents (ex : `node_generate_file`, pour chaque type de fichier) , et d'autres peuvent n'être jamais appelés (ex : `node_save_outputs` si l'option n'est pas sélectionnée, ou si le plan est bien généré `node_end_process_bad_plan_generation` n'intervient pas).

Fichier	Rôle
---------	------

node_init_project.py	Initialise la structure d'un projet DBT en créant les répertoires de base et en supprimant le projet exemple fourni par défaut par dbt.
node_ingestion.py	Lit et interprète les fichiers d'entrée (CSV, Excel, TXT, etc.) et les transforme en dictionnaire structuré utilisable par le graphe. Ingère également les métadonnées depuis Snowflake.
node_generate_plan.py	Appel au LLM pour produire un plan de modèles DBT. Il prend en entrée les prompts et les données ingérées à l'étape précédente.
node_prepare_file_specs.py	Prépare les spécifications détaillées des fichiers à créer : il découpe le plan en spécifications fichier par fichier.
node_generate_file.py	Appel au LLM pour générer le contenu d'un fichier (SQL, test, documentation) en fonction de sa spécification.
node_create_file.py	Écriture physique des fichiers .sql ou .yml dans l'arborescence DBT, au bon endroit.
node_generate_source_yaml.py	construit le fichier schema.yml à partir du plan et des sources, sans recours au LLM (remplaçant l'ancien mécanisme basé sur prompts).
node_save_outputs.py	Sauvegarde finale du projet DBT dans outputs/
Node_end_process.py	Clôture le pipeline en cas de succès. (Et répare la variable state qui est corrompu dans le Streamlit à cause des fonctions upload)
node_end_process_bad_plan_generation.py	Arrêt contrôlé si le plan généré est vide ou invalide (sans balise `json`, json corrompu ...) au milieu)

Tableau récapitulatif des fichiers présents dans src/graph/nodes

II.B.3 Tests et prototypage - `src/test/`

Ce dossier contient principalement les fichiers et fonctions utilisés avant d'implémenter l'interface avec Streamlit. Nous faisions alors nos tests sur des notebooks pour ne lancer que les cellules utiles au débogage. Le dossier contient donc :

- **test_chef_orchestre2.ipynb** : notebook pour tester le graphe LangGraph et visualiser son comportement étape par étape. Utilisé au tout début du projet.
- **test_générer_exemple.py** : script simplifié permettant de générer un exemple de spécification de projet DBT à partir d'un fichier source. Contenu détaillé en [I.F Entrées utilisateur – inputs/](#)

II.B.4 Outils partagés - `src/utils/`

Le dossier rassemble un ensemble d'outils transversaux utilisés par plusieurs composants du pipeline. Ces modules n'appartiennent pas directement à la logique de génération ni à l'orchestration LangGraph, mais ils fournissent des **services de support essentiels** : suivi des graphes, gestion des logs en mémoire, et fonctions génériques de manipulation de fichiers et modèles.

Fichier	Rôle
<code>utils.py</code>	Centralise des fonctions utilitaires génériques. Ces fonctions sont indépendantes de la logique métier mais facilitent la sérialisation , la gestion de fichiers , l' intégration avec Streamlit et l' usage des templates. C'est une boîte à outils réutilisable par les noeuds du graphe et les tâches du pipeline.
<code>graph_tracker.py</code>	Introduit une fonctionnalité de tracabilité des graphes LangGraph . Ce module enregistre les transitions entre noeuds lors de l'exécution du pipeline et permet de générer automatiquement une représentation visuelle sur le Streamlit.
<code>logging_buffer.py</code>	Implémente un système de journalisation en mémoire utilisé par l'application Streamlit. En plus d'écrire immédiatement les logs dans plusieurs fichiers dédiés, ceux-ci sont filtrés et stockés dans un tampon consultable en temps réel , ce qui permet de les afficher en direct dans l'interface (avec défilement). Ce mécanisme rend le suivi d'exécution plus fluide et interactif.

Tableau récapitulatif des fichiers présents dans `src/utils/`

II.C Configuration – `config/`

Ce dossier contient les **paramètres de configuration globaux** du projet, notamment les chemins, fichiers d'entrée/sortie et variables d'environnement utilisées dans le pipeline. Il est composé de deux fichiers :

- `variables.py` centralise :
 - les **chemins** ((inputs, outputs, formats, etc.)

- les **constantes** (valeurs par défaut, templates, dictionnaires associatifs, LLM choisi, etc.)
- les **clés d'accès** via le .env (accès Snowflake, Grok pour appels aux LLM, etc.).
- css_variables.py contient le **CSS pour l'interface** Streamlit afin de gérer la disposition de la page, les polices d'écriture, couleurs et tailles des éléments, etc. Ce mécanisme permet de garantir un bon rendu entre les différentes sections de l'interface (formulaire de saisie, affichage des logs, graphe en temps réel entre autres).

II.D Documentation – [docs/](#)

Ce dossier contient les **documents de référence et les illustrations** utilisées pour comprendre ou expliquer le fonctionnement du projet.

Fichier	Rôle
Documentation.docx	Ce présent fichier pour la documentation sous format Word
README.md	Fichier Markdown principal de la documentation
exemple_specs.png	Exemple annoté d'un fichier de spécifications utilisateur
graphViz.drawio.png	Schéma du graphe LangGraph généré (structure logique du pipeline)
Orchestrenom.drawio.png	Autre représentation visuelle de l'orchestration globale
test.drawio.png	Schéma de test / validation utilisé lors du développement

Tableau récapitulatif des fichiers présents dans docs/

II.E Spécifications et Prompts – [formats/](#)

Ce dossier contient un template du **plan attendu** par le système et les **prompts LLM** utilisés pour la génération des fichiers DBT. Chaque prompt correspond à un type de fichier DBT que l'on veut générer.

II.E.1 Template principal

Ce template (nommé expected_format.json) sous format JSON **contient la façon dont on veut que le LLM génère le plan** qui permettra ensuite de générer les fichiers un à un. Il comporte les différents types de fichiers retrouvés dans une structure DBT. Il est chargé au début de la génération, modifié pendant la phase d'ingestion (pour y rentrer les

informations des sources stockées sur Snowflake, c'est-à-dire le nom des tables, des colonnes, leur typage, etc.). Enfin il est utilisé dans la génération du plan, en l'injectant dans prompt_plan.txt (voir suite).

II.E.2 Prompts LLM

Ces prompts sont contenus dans le sous-dossier `prompts/` de `formats/`. Ils sont ensuite chargés en tant qu'objet *Template* du module *jinja2*. Ainsi on peut les compléter facilement avec des variables / données chargée au lancement du programme, comme les métadonnées de Snowflake ou le fichier de spécifications ingérés à chaque lancement, ou les chemins vers les dossiers choisis par l'utilisateur par exemple.

Voici les différents prompts :

Fichier	Rôle
<code>prompt_file_model.txt</code>	Prompt pour générer un fichier .sql de modèle
<code>prompt_file_test.txt</code>	Prompt pour générer les tests associés (.yml)
<code>prompt_plan.txt</code>	Prompt pour produire le plan global des modèles à partir de la spec

Tableau récapitulatif des fichiers présents dans `formats/prompts/`

II.F Entrées utilisateur – `inputs/`

Ce dossier contient les **fichiers de spécifications fournis par l'utilisateur**. Ils peuvent être en différents formats : .csv, .tsv, .txt, .xlsx. Ils ont été générés pour la plupart avec le notebook `test_générer_exemple.py` ([voir partie I.B.3](#)). Ces fichiers contiennent la même chose : la table finale (mart) attendue à la fin du projet DBT sous la forme :

Column name	Description	SQL
-------------	-------------	-----

Voir en [ANNEXE](#) un exemple de fichier de spécification.

Fichier	Rôle
<code>a_peaks.tsv</code>	Exemple de fichier tabulé (\t) de spécifications
<code>routes.csv</code>	Exemple au format CSV classique
<code>spec_members.txt</code>	Fichier texte brut de spécifications

Tableau récapitulatif des fichiers présents dans inputs/

Ces fichiers sont lus par le nœud `node_ingestion.py`, transformés selon le type vers une structure Python uniforme (dictionnaire), puis utilisés dans les étapes suivantes, où l'on requiert d'avoir une vue sur l'objectif final du projet.

II.G Résultats générés – outputs/

Ce dossier contient les **projets DBT générés** automatiquement.

Exemple de structure de l'output

Chaque sous-dossier correspond à une exécution complète du pipeline. Chaque exécution est identifiée par un **nom de projet**, potentiellement suffixé (_1, _2, etc.) pour éviter les collisions si un projet du même nom a déjà été généré. Ces sous-dossiers contiennent systématiquement trois éléments :

1. Le projet DBT généré
2. Les **logs d'exécution** dans logs/
3. Les **artefacts d'orchestration** dans saved/

```
outputs/
  AprilExpedition/
    AprilExpedition/
      logs/                                # Projet DBT généré
      saved/                               # Journaux d'exécution
      saved/                               # Artefacts de l'orchestration
  AprilExpedition_1/
    AprilExpedition/
      logs/
      saved/
  test/
    logs/
    saved/
    test/
  test_log_direct_2/                      # Ici test_log_direct et test_log_direct_1
    logs/                                  # ont vraisemblablement été supprimés
    saved/
    test_log_direct/
  vendredi_3/                             # Pareil qu'au-dessus
    logs/
    saved/
    vendredi/
```

II.G.1 Journaux d'exécution - logs /

Ce dossier contient tous les **logs générés pendant le processus**, accessibles pour vérification ou débogage. Les fichiers de log sont **séparés par type**, facilitant le débogage:

Fichier	Description
dbt.log	Log généré automatiquement par la commande dbt init
debug.log	Messages d'information généraux sur le déroulement du pipeline
errors.log	Trace des erreurs levées (analyse, LLM, écriture fichiers...)
warnings.log	Avertissements non bloquants émis pendant l'exécution

Tableau récapitulatif des fichiers présents dans logs/ d'un projet

Les logs sont redirigés en temps réel dans l'interface Streamlit et enregistrés ici pour archivage.

II.G.2 Sauvegardes du pipeline – saved/

Le dossier saved/ contient deux fichiers structurés en JSON qui permettent de **reconstituer l'état du raisonnement du LLM** sans devoir relancer une génération complète. Cela permet notamment de faire le **débogage sans dépenser de tokens** avec des appels au LLM, mais également de **réutiliser un plan intéressant** mais dont la génération du contenu des fichiers peut être améliorée. Ces fichiers jouent donc un rôle fondamental dans la logique de persistance du graphe LangGraph :

Fichier	Description
saved_plan.json	Contient le plan généré par le LLM . Ce plan décrit la structure logique du pipeline à générer : noms de modèles, dépendances entre tables, logique de transformation, etc. Il s'agit d'une version épurée et structurée du retour du LLM, utilisée pour l'exécution ou la relecture.
saved_files.json	Contient la liste des fichiers à générer , toujours extraite du retour du LLM mais centrée sur l'organisation des fichiers DBT : leur nom, leur type (model, test, doc, etc.), leur destination et leur contenu. Ce fichier est utilisé pour régénérer les fichiers sur disque si besoin sans nouvelle requête IA.

Tableau récapitulatif des fichiers présents dans saved/ d'un projet

Ces fichiers ne contiennent **pas les réponses complètes du LLM**, uniquement les éléments structurants utilisés pour la génération (on se moque des phrases introductives, qu'il a fallu filtrer avec des expressions régulières).

II.G.3 Projet DBT généré - <nom_projet>/

Ce dossier contient le **projet DBT complet** produit à partir des spécifications. Il inclut :

- Les modèles SQL (.sql)

- Le schéma YAML (.yaml)
- Les tests des modèles (.sql)
- La structure models/, sources/...

#a détailler dans le rapport (Ce contenu est détaillé en **Chapitre IV – Structure du projet DBT générée.**)

III.Fonctionnement du pipeline

Ce chapitre présente l'enchaînement des opérations réalisées pour générer automatiquement un projet DBT structuré à partir de spécifications fonctionnelles ou de descriptions en langage naturel.

Le pipeline est **piloté par un graphe LangGraph**, dont les nœuds exécutent des tâches bien définies sur un état partagé. Chaque nœud correspond à une étape distincte du processus. Les informations circulent entre les nœuds via un objet state, mis à jour dynamiquement. Cette section décrit en détail le rôle de chaque nœud, leurs entrées, traitements, et sorties.

Le pipeline intelligent repose sur :

- **Un LLM** (modèle de langage) pour interpréter la demande de l'utilisateur, générer un plan, puis produire les fichiers DBT.
- **Un graphe LangGraph**, où chaque nœud correspond à une action logique (ingestion, planification, génération de fichiers, etc.).
- **Un état partagé (state)**, qui conserve toutes les informations utiles et qui circule dans le graphe, permettant à chaque nœud d'accéder aux résultats intermédiaires.
- **Un orchestrateur** (`generate_dbt_project.py`) qui pilote l'ensemble du processus et encapsule la logique de lancement.
- **Une interface Streamlit** (`app.py`), qui sert de point d'entrée pour l'utilisateur et fournit un retour visuel et interactif sur l'avancement.

Ce chapitre détaille le fonctionnement du pipeline en explorant les différents nœuds qui le composent, leurs entrées et sorties, ainsi que la manière dont ils interagissent avec l'état partagé et les fichiers générés.

III.A Saisie des entrées de l'utilisateur

La première étape du pipeline consiste à recueillir les informations de départ fournies par l'utilisateur. Ces entrées peuvent provenir de deux canaux : l'interface graphique développée avec **Streamlit** ou la ligne de commande via le fichier `main.py` :

- via **l'interface graphique** Streamlit, pensée pour un usage simple et accessible,
- via la **ligne de commande** (`main.py`), destinée aux utilisateurs avancés ou aux intégrations automatisées. Il permet via des tâches de faciliter la gestion d'un projet.

III.A.1 Interface graphique – Streamlit (app.py)

L’interface Streamlit constitue le **point d’accès principal** du projet. Elle permet à l’utilisateur d’entrer ses inputs avec des champs de saisies, des sélecteurs de fichiers / dossiers et des options à cocher ou décocher.

a) *Présentation générale*

Au lancement via la commande « `streamlit run app.py` », l’utilisateur accède à une interface web qui centralise la saisie des informations nécessaires :

- fichier de spécifications (Excel, TSV ou CSV),
- des informations sur le projet (nom souhaité, endroits où le créer, profil DBT)
- options avancées (réutilisation d’un plan ou de fichiers précédents, sauvegarde des résultats).

Un bouton principal déclenche ensuite la génération du projet DBT. L’avancement et les logs sont affichés en temps réel dans une fenêtre modale interactive. Juste à côté se trouve un graphe récapitulant les nœuds et arêtes que le programme va parcourir, en indiquant par un code couleur le chemin suivi, l’étape actuelle et les nœuds provoquant une erreur.

b) *Organisation des fonctionnalités*

L’application se décompose en plusieurs étapes clés :

1. Configuration de la page

La page Streamlit est configurée avec un titre, un style CSS personnalisé, et une disposition large pour maximiser l’espace.

L’interface affiche d’emblée une présentation succincte du rôle de l’outil.

2. Chargement des fichiers d’entrée

L’utilisateur doit importer un fichier de spécifications (`.xlsx .tsv` ou `.csv`).

Un message d’erreur s’affiche immédiatement si ce fichier est manquant au moment de la génération, ou si l’extension n’est pas bonne.

3. Paramétrage du projet DBT

Plusieurs champs permettent de définir :

- le nom du projet DBT,
- le nom du profil DBT associé,
- le schéma cible (pas encore utile),
- le répertoire de sortie (avec possibilité de sélection via une boîte de dialogue système).

En cas de conflit (dossier déjà existant), un suffixe numérique est automatiquement ajouté au nom du projet afin d'éviter l'écrasement.

4. Options avancées

- L'utilisateur peut choisir de sauvegarder les résultats (plan et/ou contenu des fichiers pour une génération ultérieure). Valeur True par défaut.
- Il peut également charger un plan déjà existant (au format JSON).
- Pour les besoins de débogage, il est aussi possible de réutiliser directement des fichiers DBT générés auparavant : cela évite le coup en temps et énergie d'un appel du LLM.

5. Gestion de l'exécution

Le bouton “**Generate Project**” déclenche l’ensemble du processus dans un thread séparé, garantissant la fluidité de l’interface.

Une fenêtre modale affiche alors :

- un graphe de suivi en direct des nœuds LangGraph,
- les logs en direct,
- une animation d’attente (spinner).

À la fin de la génération, l’utilisateur peut télécharger :

- l’état interne (state) complet du pipeline (dans le fichier result.json),
- les logs détaillés (dans le fichier generation.log, s’il ne veut pas aller les chercher directement dans les logs de son projet fraîchement créé).

6. Robustesse et gestion des erreurs

Différents types d’erreurs sont interceptés et affichés à l’utilisateur :

- fichier introuvable,
- erreur JSON,
- clé manquante dans l’état interne,
- problème de permissions,
- erreurs de typage ou exceptions inattendues.

Cette granularité permet de diagnostiquer précisément l’origine d’un problème.

Vous trouverez en **ANNEXE** l’aperçu de la page Streamlit accessible via `app.py`.

III.A.2 Ligne de console – main.py

En parallèle, une interface en ligne de commande est disponible via le script `main.py`. Celui-ci reprend la **logique des tâches** (initialisation, clean, backup, run, etc.) et permet de piloter le pipeline sans interface graphique. Le fichier `main.py` constitue le **point d'entrée** principal du programme en mode **CLI** (Command Line Interface). Il ne contient pas directement la logique métier de chaque commande, mais sert désormais uniquement à **router** les commandes vers les bonnes tâches définies dans `src/core/tasks/`.

a) *Commandes disponibles*

Commande CLI	Module associé (dans <code>src/core/tasks/</code>)	Description
<code>python main.py install</code>	<code>task_install.py</code>	Crée l'environnement virtuel et installe les dépendances nécessaires au projet.
<code>python main.py run</code>	<code>task_run.py</code>	Lance le pipeline de génération DBT en mode console, avec logs en direct.
<code>python main.py backup <project_name></code>	<code>task_backup.py</code>	Crée un backup horodaté du projet DBT sous <code>backups/</code> .
<code>python main.py clean</code>	<code>task_clean.py</code>	Supprime les caches (<code>__pycache__</code>) et le dossier <code>outputs/</code> , puis recrée un environnement propre.
<code>python main.py init</code>	<code>task_init_env.py</code>	Initialise l'environnement de génération : configuration interactive (<code>.env</code> , <code>profiles.yml</code>) de tous les paramètres utiles pour créer un projet automatiquement.
<code>python main.py help</code>	Intégré au <code>main.py</code>	Affiche la liste des commandes disponibles et leur usage.

Tableau récapitulatif des commandes disponibles

b) *Fonctionnement général*

- `main.py` analyse les arguments de la ligne de commande (`sys.argv`).

- En fonction de la commande demandée (`install`, `run`, `backup`, `clean`, `help`, `init`), il **importe dynamiquement** et appelle la fonction correspondante depuis `src/core/tasks/`. L'importation dynamique est nécessaire avant l'installation des dépendances (avec `python main.py install`) sinon cela provoque des erreurs d'import de module(s) inexistant(s) (car pas encore installés)
- Chaque fichier de `tasks/` contient une implémentation isolée et documentée de la commande.

III.B Initialisation du projet – `node_init_project`

La première étape du graphe LangGraph consiste à initialiser l'environnement de travail. Le nœud `node_init_project` se charge de préparer le terrain avant l'ingestion des données.

Son rôle principal est de :

- Créer une nouvelle arborescence DBT via la commande :

```
dbt init {project_name} --profile {profile_name}
```
- Supprimer les exemples par défauts créés automatiquement (car inutiles)

En sortie, l'état contient donc un environnement DBT propre et prêt à accueillir les fichiers générés. Il faut cependant gérer le profile dbt utilisé dans :

`C:\Users\{nom_utilisateur}\.dbt\profiles.yml`

Pour remplir ce fichier il faut se munir de ses propres informations Snowflake. Le nom du profile (dans l'exemple `mon_profile`) est le même que `profile_name` renseigné dans `app.py` ou `main.py`. Attention à correctement le renseigner, sinon une erreur à l'étape est levée « `[ERROR] - app : X Unexpected error in worker thread: Error while creating the dbt project: Command '['dbt', 'init', 'mon_projet', '--profile', 'mon_profil']' returned non-zero exit status 2.` ».

Il faut donc pour s'assurer d'avoir bien créer le profile que l'on veut utiliser, facilement faisable avec `python main.py init`, pour créer un nouveau profile, ou simplement vérifier l'existant du profile.

Exemple de configuration :

```
mon_profile:  
  outputs:  
    dev:  
      account: EXEMPLE_COMPTE_SNOWFLAKE  
      database: EXEMPLE_BDD_SNOWFLAKE  
      password: EXEMPLE_MOT_DE_PASSE_SNOWFLAKE  
      role: EXEMPLE_ROLE  
      schema: EXEMPLE_SCHEMA  
      threads: 10  
      type: snowflake  
      user: EXEMPLE_UTILISATEUR_SNOWFLAKE  
      warehouse: EXEMPLE_WAREHOUSE_SNOWFLAKE  
    target: dev
```

III.C Ingestion des métadonnées – *node_ingestion*

Le nœud `node_ingestion` constitue la première étape clé du pipeline de génération de projet dbt.

Son rôle est de **centraliser toutes les métadonnées utiles** (issues de fichiers fournis par l'utilisateur et de la base de données Snowflake) et de les stocker dans l'état partagé du graphe. En combinant ces deux sources, le LLM dispose d'une vue **complète et contextualisée** du projet :

- besoins métiers (colonnes attendues),
- réalité technique (tables et colonnes existantes).

Cela garantit une génération de projet dbt plus robuste et adaptée au contexte réel de l'utilisateur.

Ce nœud assure deux fonctions complémentaires :

1. **Ingestion des spécifications de marts** à partir d'un fichier (CSV, TSV, Excel).
2. **Ingestion des métadonnées Snowflake** (tables, colonnes, types, descriptions, ...) directement depuis le schéma de la base.

III.C.1 Ingestion des spécifications (marts data)

L'utilisateur peut fournir un fichier de spécification décrivant les colonnes attendues dans ses marts (ou modèles cibles).

Le fichier doit obligatoirement contenir les colonnes suivantes :

- **Column name** : nom de la colonne,
- **Description** : description métier de la colonne,
- **SQL** : expression SQL permettant de calculer la colonne (peut être le nom de la donnée directement, sous format : `table.nom`).

<i>Column name</i>	<i>Description</i>	<i>SQL</i>
<i>expid</i>	Expedition internal ID	<code>exped.expid</code>
<i>summit</i>	Summit name	<code>peaks.pkname ', ' peaks.location</code>
<i>year</i>	Year of the expedition	<code>exped.year</code>

Exemple de fichier en entrée (CSV/Excel)

Après ingestion et normalisation des noms de clés, le fichier est converti en dictionnaire python comme ci-après :

```
state["ingested_marts_data"] = [
    {
        "column_name": "EXPID",
        "description": "Expedition internal ID",
        "sql": "EXPED.EXPID"
    },
    {
        "column_name": "SUMMIT",
        "description": "Summit name",
        "sql": "PEAKS.PKNAME || ', ' || PEAKS.LOCATION"
    },
    {
        "column_name": "YEAR",
        "description": "Year of the expedition",
        "sql": "EXPED.YEAR"
    },
    ...
]
```

Exemple du contenu de la variable state (clé: ingested_marts_data contenant les données du fichier de spécification)

III.C.2 Ingestion des métadonnées Snowflake

En complément, le nœud se connecte à **Snowflake** pour récupérer automatiquement les métadonnées des tables existantes. Cela permet au LLM de générer un projet dbt cohérent avec la réalité technique de la base. L'ingestion des métadonnées techniques repose sur les fonctions centralisées dans le module `src/core/snowflake_fct.py`.

Ces fonctions assurent la connexion sécurisée à Snowflake, l'extraction des informations sur les tables et colonnes d'un schéma, puis leur mise en forme pour être compatibles avec la structure attendue par **dbt** dans les fichiers `schema.yml`.

a) Connexion à Snowflake

La connexion est gérée via le context manager `get_snowflake_connection(logger)`, qui ouvre une session Snowflake à partir des paramètres définis dans le fichier caché `.env` (utilisateur, mot de passe, compte, rôle) et le fichier `config/variables.py` (warehouse, database, schéma).

Cette fonction est ensuite appelée avec un pattern qui garantit :

- une connexion unique et sécurisée par bloc `with`,
- la persistance de la connexion et sa fermeture automatique,
- la remontée contrôlée des erreurs dans les logs.

b) Extraction des métadonnées

La fonction principale pour l'extraction des données de Snowflake, `ingest_metadata_from_snowflake_to_sources_format(logger)`, orchestre le processus comme suit :

1. Récupération des tables :

Une requête `SHOW TABLES IN <DATABASE>.SCHEMA` est exécutée pour obtenir la liste des tables disponibles.

2. Récupération des colonnes :

Pour chaque table identifiée, une seconde requête `SHOW COLUMNS IN TABLE <DATABASE>.SCHEMA.<TABLE>` extrait les colonnes associées, avec leur type de données.

- Si le type est encodé en JSON (cas fréquent), il est décodé pour extraire la valeur du champ type.
- En cas d'échec du parsing, une alerte est loggée et la valeur brute est utilisée.

3. Mise en forme des colonnes :

Chaque colonne est représentée par un dictionnaire minimal :

```
{  
    "name": "<column_name>",  
    "description": "<description>",  
    "data_type": "<data_type>",  
}
```

Pour récupérer les descriptions de colonnes depuis Snowflake, on teste si elles sont vides ou non. Il faut en effet les récupérer pour les injecter dans le source.yml mais pas dans le prompt du LLM car cela augmente considérablement la taille et donc les hallucinations. Ainsi, le champ "description" n'apparaît que si pertinent.

4. Construction de la source dans le State:

L'ensemble des tables et de leurs colonnes est regroupé dans un dictionnaire qui sera stocké dans la partie ["sources"] de la variable état state pour se greffer directement au futur prompt (voir [II.C.3.b](#)). Elles seront également utiles pour construire la structure YAML source .yml qui décrit le projet dbt.

```
state["sources"] = {  
    "name": "<database>_<schema>",  
    "description": "Extracted from Snowflake schema <DATABASE>.<SCHEMA>",  
    "path": "<DATABASE>.<SCHEMA>",  
    "tables": [  
        {  
            "name": "<table_name>",  
            "columns": [  
                {  
                    "name": "<col1>",  
                    "data_type": "<type1>",  
                    "description": "Member ID.", # apparaît si présent  
                },  
                {  
                    "name": "<col2>",  
                    "data_type": "<type2>"  
                },  
                ...  
            ]  
        },  
        ...  
    ]  
}
```

Exemple du contenu de la variable state (clé : "sources" contenant les données du fichier source.yml)

5. Retour de la fonction :

La fonction renvoie une **liste contenant un unique objet source** (structure extensible si, à terme, plusieurs schémas devaient être ingérés).

c) *Intégration dans le pipeline*

La fonction `make_ingestion(log)` fabrique la fonction exécutée par le graphe LangGraph. Lors de son exécution, elle :

1. Lance l'ingestion des fichiers utilisateurs (marts).
2. Lance l'ingestion des métadonnées Snowflake.
3. Met à jour le `state` avec :
 - o `state["ingested_marts_data"]`
 - o `state["sources"]`
 - o `state["generation_status"]`

En cas d'erreur (fichier manquant, Snowflake inaccessible...), la fonction logge l'incident et met à jour le `generation_status` pour permettre une gestion d'erreur dans le graphe.

Une bonne ingestion se termine par les logs :

```
[11:52:21] [DEBUG] 📁 Reading file: Tableau_Spec.xlsx
[11:52:21] [INFO] ✅ Mart data ingested (10 rows)
[11:52:21] [DEBUG] 💡 Starting metadata ingestion from Snowflake.
[11:52:21] [DEBUG] Connecting to Snowflake...
[11:52:22] [DEBUG] Snowflake connection established.
[...]
[11:52:24] [DEBUG] Snowflake connection closed.
[11:52:24] [INFO] ✅ Snowflake metadata ingested (4 tables)
[11:52:24] [DEBUG] ⚡ Snowflake Ingestion details: 4 tables, 161 columns
(1x65 + 1x61 + 1x23 + 1x12)
```

III.C.3 Planification du projet

La phase de planification constitue le **pivot du pipeline** : elle traduit les inputs bruts (description fonctionnelle, spécifications, métadonnées Snowflake) en un **plan structuré et exploitable**. Elle intervient immédiatement après l'ingestion des métadonnées et avant

la génération des fichiers DBT.

Ce plan est un dictionnaire JSON, enregistré dans `state["plan"]`, qui décrit :

- Les **sources** (les tables brutes issues de Snowflake),
- Les **modèles de DBT** (staging / intermediates / marts),
- Les **relations et dépendances** entre objets (DAG),
- Les **tests** de qualité des données à appliquer,
- Les **descriptions métiers** destinées à enrichir la documentation dbt.

Ce plan est le **document directeur** du projet DBT : il détermine quels modèles seront créés, à partir de quelles sources, comment les transformations seront organisées, comment s'organise l'arborescence et quelles dépendances logiques doivent être respectées.

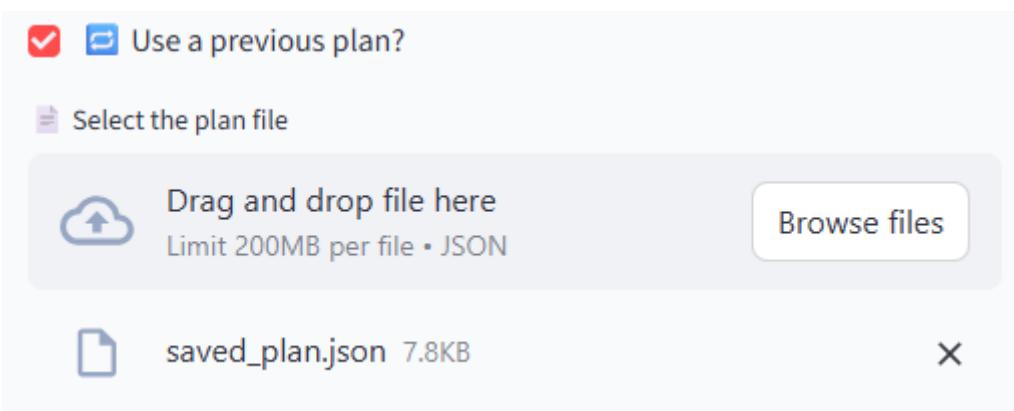
D'un point de vue technique, cette étape s'appuie sur deux mécanismes :

1. **Un module de sélection (edge_select_plan)** : il décide si le pipeline doit charger un plan existant (fichier sauvegardé dans `saved/`) ou en générer un nouveau à l'aide du LLM.
2. **Un générateur de plan (node_generate_plan)** : si aucune version précédente n'est fournie, il construit automatiquement un plan en interrogeant le LLM, sur la base d'un prompt adapté ([ANNEXE](#)), injectant les formats attendus et les métadonnées disponibles.

a) *Chargement d'un plan existant – load_plan*

Cette première possibilité correspond à une logique de réutilisation.

Si l'utilisateur a déjà effectué une exécution du pipeline et qu'un plan a été sauvegardé dans le dossier `saved/` de ce premier projet, celui-ci peut être rechargé au lieu d'être régénéré.



Aperçu de l'utilisation d'un plan précédent

- **Entrées :**
 - L'état (state) contient une clé path_to_saved_plan renseignée manuellement par l'utilisateur ou automatiquement par une exécution précédente.
- **Traitement :**
 - Le pipeline lit le fichier JSON du plan sauvegardé.
 - Le contenu est validé puis injecté dans la clé state["plan"] .
- **Sorties :**
 - Un plan immédiatement disponible, sans appel au LLM.

Cette approche présente deux avantages majeurs :

1. Elle permet de **gagner du temps** en évitant une nouvelle génération pour le débogage.
2. Elle garantit une **reproductibilité**, puisque le plan est figé et correspond exactement à une version précédemment validée. Ainsi si le plan plaît à l'utilisateur mais pas le contenu des fichiers, il peut repartir de cette étape.

b) Génération d'un nouveau plan – node_generate_plan

Si aucun plan n'est disponible ou si l'utilisateur souhaite repartir d'une nouvelle description, c'est le **LLM** qui est sollicité pour élaborer un plan.

Ce mécanisme est plus complexe car il doit à la fois :

- intégrer les **contraintes du format attendu**,
- exploiter les **métadonnées disponibles** (tables, colonnes, descriptions des marts),
- respecter des règles de cohérence propres à DBT.

Étapes de la génération :

1. Construction du prompt

Le module commence par assembler un prompt textuel, à partir d'un template stocké dans formats/prompts/plan_prompt.txt.

Ce template est complété:

- le expected_format.json (description structurée du type de plan attendu, par exemple on veut les clés modèles, sources, tests, etc.),

- les métadonnées ingérées depuis Snowflake, ajoutée au sources du point précédent à l'étape d'ingestion (voir [II.C.2.b](#) point 4)
- les métadonnées `ingested_marts_data` (correspondant au fichier d'entrée des spécifications) qui décrivent les marts que nous voulons générés *in fine*.

Ainsi, le LLM reçoit une consigne claire, normalisée et contextualisée (qui peut cela dit être améliorée avec du prompt engineering).

[insérer exemple : extrait du prompt généré, avec placeholders remplacés par du JSON de sources et colonnes]

2. Appel au LLM

Le LLM (via Groq) est invoqué avec ce prompt.

Le retour attendu est un **JSON** contenant un plan structuré, avec des clés telles que :

- `models` : description des modèles à créer (nom, SQL de transformation, dépendances),
- `tests` : quelques validations à appliquer (not null, unique, etc.).

[insérer exemple : retour brut du LLM avant parsing]

3. Extraction et validation du plan

Le texte renvoyé par le LLM est traité par la fonction `extract_plan`, qui isole la portion JSON valide et la convertit en dictionnaire Python.

Des vérifications sont effectuées pour s'assurer que le contenu est exploitable (par exemple : le JSON est bien formé, les clés attendues sont présentes).

Si le format (balise `json`) ne correspond pas, que des caractères spéciaux (ex : « ... ») corrompent le dictionnaire, que le temps de génération est trop long, ou toute autres erreurs résultant de la génération adviennent, nous bouclons (selon le nombre d'essais `MAX_GENERATION_ATTEMPTS` prédéfini dans `variables.py`) sur l'appel du nœud. Ce système de **retry** permet de **ne pas interrompre le processus** si le LLM hallucine ou sort de sa tâche.

4. Stockage dans l'état partagé

Une fois validé, le plan est enregistré dans `state["plan"]`.

L'état conserve également une clé `generation_status`, qui trace la réussite ou l'échec du processus.

- **Exemple de plan généré**

Un plan typique peut ressembler à ceci, chaque partie étant raccourci afin de rentrer dans la documentation :

```
"models": [
  {
    "name": "exped",
    "description": "Staging model for expeditions data",
    "model_type": "stage",
    "domain": "himalayan_expeditions_seeds",
    "path": "models/staging/exped.sql",

    "columns": [
      {
        "name": "expid",
        "data_type": "TEXT"
      }, ...
    ],
    "tests": [
      {
        "name": "unique_expid",
        "description": "Expedition ID should be unique",
        "path": "tests/staging/exped/unique_expid_test.sql"
      }, ...
    ]
  },
]
```

Bloc `models` qui décrit les modèles SQL à créer

```
"tests": [
  {
    "name": "exped_expid_unique",
    "description": "Expedition ID should be unique",
    "path": "tests/",
    "model": "exped",
    "dependencies": [
      "exped"
    ]
  },
  ...
]
```

Bloc `tests` qui décrit les tests SQL à créer

```

"sources": [
  {
    "name": "himalayan_expeditions_seeds",
    "description": "Raw data from Himalayan Expeditions Seeds",
    "path": "HIMALAYAN_EXPEDITIONS.SEEDS",
    "tables": [
      {
        "name": "EXPED",
        "columns": [
          {
            "name": "EXPID",
            "data_type": "TEXT"
          },
          ...
        ]
      },
      {
        "name": "MEMBERS",
        "columns": [ ... ]
      }
    ]
  ],
]

```

Bloc `sources` qui réfère les métadonnées des sources disponibles sur Snowflake pour la création du contenu SQL dans les fichiers.

Intérêt de cette étape

La génération de plan joue un rôle pivot dans le pipeline :

- Elle **formalise la demande utilisateur** en une représentation exploitable par DBT.
- Elle **normalise les dépendances**, évitant que le LLM génère directement des fichiers incohérents.
- Elle **alimente les étapes suivantes**, puisque ce plan sert de base à la génération automatique des fichiers DBT (SQL, schema.yml).

III.D Génération du schéma sources – `node_generate_source_yaml`

Une fois le plan généré ou chargé, le pipeline doit produire les fichiers nécessaires à DBT. La première étape de cette génération concerne le **fichier schema.yml**, qui est **conforme aux standards dbt**, intégrant à la fois les **sources** (issues du métamodèle attendu et de Snowflake) et les **models** (issus du plan généré à l'étape précédente).

Le nœud `node_generate_source_yaml` se charge de cette tâche. Contrairement à la génération de plan, il ne fait **pas appel à l'IA** : il construit directement le YAML en exploitant les informations disponibles dans l'état partagé (`state`) extraites de Snowflake.

Ce fichier est central dans dbt, puisqu'il :

- Déclare les **sources de données** (databases, schemas, tables, colonnes).
- Décrit les **modèles dbt** créés par le pipeline.
- Définit des **tests automatiques** (ex. `not_null` sur les colonnes `id`).

III.D.1 Rôle du nœud

1. Assembler les métadonnées des sources :

À partir de `expected_format["sources"]`, le nœud récupère les informations nécessaires :

- Nom de la source (`name`)
- Description
- Base de données et schéma (extrait de `path`)
- Liste des tables et leurs colonnes

Chaque table est enrichie avec des colonnes et leurs descriptions, ainsi que des règles simples (comme l'ajout automatique d'un test `not_null` sur les colonnes terminant par `_id`).

2. Intégrer les modèles planifiés :

En parallèle, le nœud lit `plan["models"]` afin de documenter les modèles générés :

- Nom et description du modèle
- Liste des colonnes avec leurs métadonnées
- Application de règles de qualité simples (comme `not_null` sur les identifiants).

3. Produire un fichier dbt conforme :

Ces deux blocs (sources + models) sont rassemblés dans une structure dict Python qui est **sérialisée en YAML**.

Le résultat est enregistré dans le chemin :

```
<base_path>/<project_name>/models/staging/schema.yml
```

4. Mettre à jour l'état partagé :

Le YAML est conservé dans `state["generated_files"]["sources"]` afin de permettre une réutilisation ou un affichage dans l'interface utilisateur.

III.D.2 Fonctionnement détaillé

Le nœud fonctionne en plusieurs étapes :

1. Lecture des entrées depuis l'état :

- `state["sources"]` → métadonnées des tables de Snowflake.
- `plan["models"]` → plan de transformation produit par le LLM.
- `project_name` et `base_path` → pour savoir où écrire le fichier.

2. Construction des blocs YAML :

- `sources_yaml` : bloc listant toutes les sources.
- `models_yaml` : bloc listant tous les modèles planifiés.

3. Application de règles de validation simples :

- Si une colonne se termine par `id` → ajout du test `not_null`.
- Les noms sont systématiquement convertis en minuscules pour uniformiser.

4. Écriture et mise en cache :

- Le YAML final est sauvegardé sur disque.
- Une copie est stockée dans `state["generated_files"]["sources"]`.

Exemple :

```
version: 2
sources:
- name: himalayan_expeditions_seeds
  description: Extracted from Snowflake schema HIMALAYAN_EXPEDITIONS.SEEDS
  database: HIMALAYAN_EXPEDITIONS
  schema: SEEDS
  tables:
    - name: exped
      columns:
        - name: expid
          description: Column EXPID
          tests:
            - is_unique
        - name: peakid
          description: Column PEAKID
          tests:
            - not_null
        ...
    - name: members
      ...
models:
- name: exped
  description: Staging model for expeditions data
  columns:
    - name: expid
      description: Column expid
      tests:
        - not_null
    ...
    - name: summit
      description: Column summit
    ...
- name: int_expedition
  description: Intermediate model combining expedition data
  columns:
    - name: expid
      description: Column expid
      tests:
        - not_null
    - name: summit_name
      description: Column summit_name
    - name: start_date
      description: Column start_date
    - name: end_date
      description: Column end_date
```

Exemple de contenu du fichier source.yml :

La partie **sources** qui sert directement à DBT et créer automatiquement (sans LLM). Elle représente **les données sur Snowflake** (les descriptions sont génériques mais peuvent être récupérées facilement, à condition d'être exclue des prompts)

Exemple de contenu du fichier source.yml :

La partie **models** correspond aux modèles DBT générés par le LLM.

Cet exemple illustre bien le rôle du nœud : **fournir à dbt une vision unifiée** des sources brutes et des modèles créés, tout en garantissant **des tests automatiques de qualité**.

III.D.3 Intérêt de ce nœud

- **Automatisation complète** : l'utilisateur n'a pas à rédiger le schema.yml manuellement, ce qui réduit les erreurs humaines. De même pour le LLM, on lui enlève une tâche (qu'il n'excellait pas soit dit en passant)
- **Conformité DBT garantie** : le fichier suit la structure imposée par DBT (version: 2, sections obligatoires).
- **Cohérence avec le plan** : les modèles et les colonnes décrits correspondent exactement à ceux définis lors de la génération du plan.
- **Indépendance vis-à-vis du LLM** : contrairement à d'autres nœuds, node_generate_source_yaml est purement déterministe et assure une reproductibilité totale.

III.E Préparation des spécifications de fichiers - *node_prepare_file_specs*

Le nœud **prepare_file_specs** joue un rôle central : il traduit le **plan de projet** (`state["plan"]`) en un format détaillé et actionnable pour la génération physique des fichiers.

Son objectif est de **normaliser et structurer** les informations afin que chaque fichier (qu'il s'agisse d'un modèle, d'un test) dispose de **métadonnées complètes** : chemin, nom, dépendances, colonnes, description, etc.

Une fois ce nœud exécuté, l'état partagé contient une clé `state["file_specs"]`, qui organise les fichiers en trois grandes catégories :

- `models` → fichiers SQL dbt des modèles (staging, intermediate, marts)
- `tests` → fichiers SQL dbt pour valider les données et la cohérence du projet

Chaque entrée est identifiée par une clé construite à partir de son chemin et de son nom (path\name), et contient toutes les métadonnées nécessaires.

III.E.1 Fonctionnement commun

Pour chaque élément du plan :

1. Le nœud construit un **dictionnaire normalisé** avec :
 - name → nom logique du fichier (ex : stg_expeditions)
 - description → résumé fonctionnel ou métier (ex : "*Expeditions table from HIMALAYAN_EXPEDITIONS.SEEDS schema*")
 - path → chemin de sortie dans le projet dbt (ex : models/staging/himalayan_expeditions/stg_expeditions.sql)
 - dependencies → modèles ou tables dont dépend le fichier (ex : stg_expeditions, stg_peaks)
 - filetype → type de fichier (models, tests)
2. La spécification est **rangée par catégorie** (models, tests).
3. Chaque catégorie applique ensuite ses propres enrichissements spécifiques (colonnes pour les modèles, cibles de test pour les tests).

Ainsi, file_specs devient la **véritable “checklist” d’implémentation**, que le nœud suivant (node_generate_files) utilisera pour générer les fichiers physiques.

III.E.2 Modèles (filetype: models)

Pour les modèles, le nœud enrichit les spécifications avec :

- **Le type de modèle** (stage, intermediate, marts)
- **Le domaine fonctionnel** (ex : HIMALAYAN_EXPEDITIONS)
- La liste des **colonnes** avec leur nom et type SQL (ex : expid: TEXT, bcddate: DATE)

Exemple concret :

```
"models/marts/himalayan_expeditions/marts_expeditions.sql\\marts_expeditions": {
    "name": "marts_expeditions",
    "description": "Final model for expeditions",
    "model_type": "marts",
    "domain": "HIMALAYAN_EXPEDITIONS",
    "path": "models/marts/himalayan_expeditions/marts_expeditions.sql",
    "columns": [
        {
            "name": "expid",
            "data_type": "TEXT"
        },
        {
            "name": "is_success",
            "data_type": "BOOLEAN"
        },
        {
            "name": "start_date",
            "data_type": "DATE"
        },
    ],
    "dependencies": "int_expeditions_with_peaks",
    "filetype": "models"
}
```

Cet exemple illustre un **modèle de marts** : il renvoie les informations souhaitées par l'utilisateur après le traitement et les opérations effectuées dans le intermediate

III.E.3 Tests (filetype: tests)

Pour les tests, la spécification inclut :

- **Le modèle ciblé** (ex : stg_expeditions)
- **La nature du test** (not null, unique, relationships, etc.)
- **Les dépendances implicites** (le modèle testé et ses sources associées)

Exemples concrets :

```
"tests/staging/himalayan_expeditions/test_stg_expeditions_not_null.sql\\test_stg_expeditions_not_null": {
    "name": "test_stg_expeditions_not_null",
    "description": "Test if expid is not null in stg_expeditions",
    "path": "tests/staging/himalayan_expeditions/test_stg_expeditions_not_null.sql",
    "model": "stg_expeditions",
    "dependencies": "stg_expeditions",
    "filetype": "tests"
}
```

Le test devra garantir la **qualité des données** en s'assurant que la clé expid n'est jamais nulle.

```
"tests/intermediate/himalayan_expeditions/test_int_expeditions_with_peaks_relationship.sql\\test_int_expeditions_with_peaks Relationship": {
    "name": "test_int_expeditions_with_peaks Relationship",
    "description": "Test relationship between int_expeditions_with_peaks and stg_expeditions",
    "path": "tests/intermediate/himalayan_expeditions/test_int_expeditions_with_peaks_relationship.sql",
    "model": "int_expeditions_with_peaks",
    "dependencies": "int_expeditions_with_peaks, stg_expeditions, stg_peaks",
    "filetype": "tests"
}
```

Ce test devra s'assurer **l'intégrité référentielle** entre plusieurs modèles : sont-ils bien reliés ?

III.F Génération des fichiers – *node_generate_files*

III.F.1 Fonctionnement général

Le rôle du nœud **node_generate_files** est de **produire le contenu des fichiers dbt** à partir des spécifications préparées dans l'étape précédente (*file_specs*).

L'idée est la suivante :

- Chaque type de fichier (modèle, test) dispose d'un **template de prompt** spécifique.
- Les métadonnées extraites de *file_specs* (nom du modèle, colonnes, description, SQL attendu, contraintes, etc.) sont injectées dans ce template. Ces métadonnées sont propres au type de fichier concerné.
- Le prompt ainsi rendu est envoyé au **LLM** (`get_llm()`), qui génère le contenu du fichier.

- Le résultat est sauvegardé dans `state["generated_files"]`, classé par type de fichier.

III.F.2 Détails de l'implémentation

La fonction est créée via une **factory function** `make_generate_file(filetype, template_path, template_fields, log)`.

Elle retourne une fonction `generate_file(state)` qui :

1. **Récupère les spécifications** du type de fichier concerné dans `state["file_specs"]["filetype"]`.
2. **Construit un contexte** (à injecter dans le template à l'étape suivante) en sérialisant proprement les listes/dictionnaires en JSON.
 - Exemple : une liste de colonnes est transformée en JSON formaté.
3. **Charge le template** (`load_prompt_template`) et remplace les placeholders par les valeurs du contexte. Les différentes valeurs sont :
 - *name* : nom du fichier (.sql),
 - *description* : qui explique le but du fichier,
 - *dependencies* : nom des fichiers dont dépend l'actuel
 - *models_type* : le nom du modèle concerné,
 - *columns (models)* : liste des colonnes (sous format json) concernées,
4. **Envoie le prompt au LLM** via `llm.invoke(prompt)` et récupère la réponse.
5. **Sauvegarde le contenu généré** dans `state["generated_files"]["filetype"]`.
6. Met à jour `state["generation_status"]`.

III.F.3 Particularités selon le type de fichier

Même si la logique est identique, les **templates et prompts diffèrent** selon le type de fichier :

1. **Modèles (models/)**
 - a. Génèrent du code SQL (fichiers .sql) définissant des **sélections et transformations de données**.
 - b. Le prompt insiste sur les relations entre colonnes, les contraintes et la lisibilité.

- c. Il y a trois différents types de modèle (stage, intermediate et marts). Le LLM ici doit comprendre lequel est concerné pour adapter totalement la génération
- d. Exemple attendu : models/stg_expeditions

```
```sql
{{ config(
 materialized = 'table'
) }}

SELECT
 expid::TEXT AS expid,
 year::FIXED AS year,
 season::TEXT AS season,
 nation::TEXT AS nation,
 leaders::TEXT AS leaders,
 sponsor::TEXT AS sponsor,
 success1::BOOLEAN AS success1,
 success2::BOOLEAN AS success2,
 success3::BOOLEAN AS success3,
 success4::BOOLEAN AS success4,
 bcdate::DATE AS bcdate,
 termdate::DATE AS termdate,
 peakid::TEXT AS peakid
FROM
 HIMALAYAN_EXPEDITIONS.SEEDS.EXPDITIIONS
```

```

Ce fichier est un **exemple** d'un « bon » **contenu généré** par le **LLM**. En effet, il a bien sélectionné les colonnes intéressantes pour la tâche. Cependant, il y a une **coquille** : dans le « FROM HIMALAYAN_EXPEDITIONS.SEEDS.EXPDITIIONS », la table comporte une faute (il manque un « E »). HIMALAYAN_EXPEDITIONS.SEEDS.EXPDITIIONS Devrait être HIMALAYAN_EXPEDITIONS.SEEDS.EXPEDITIONS Ce genre d'erreur est imprévisible, et rend la génération obsolète est **dépendante** d'une **correction** (humaine ou IA avec un futur nœud par exemple)

2. Tests (tests/)

- a. Génèrent des fichiers SQL décrivant les **tests dbt** (unicité, non-null, références, contraintes et autres tests que le LLM peut juger utile). Certains sont forcés (voir lien)
- b. Exemple attendu :

```
{{ config(
    tags = ['unit_tests']
) }}

WITH test_data AS (
    SELECT expid
    FROM {{ ref('stg_expeditions') }}
    GROUP BY expid
    HAVING COUNT(*) > 1
)
SELECT *
FROM test_data

```

Ce fichier est un **exemple** d'un bon **contenu généré** par le **LLM**. En effet, c'est un **test d'unicité**, qui renvoie les expid qui ont plusieurs occurrences. Ainsi si le test est bon, il ne renvoie rien : ce qu'on attend d'un test DBT.

3. Robustesse et gestion des erreurs

- Si le LLM échoue ou renvoie une erreur (RuntimeError), un **contenu de fallback** est enregistré (message d'erreur au lieu du fichier).
- Chaque génération est **loggée** en détail pour suivi et debug (prompt envoyé, réponse reçue, fichier généré).
- Le système reste **idempotent** : une relance régénère les fichiers manquants ou défectueux sans casser le reste.

III.G Création des fichiers – *node_create_file*

III.G.1 Rôle du nœud

Le nœud **node_create_file** incarne la phase finale de matérialisation des fichiers générés. Jusqu'à présent, les contenus SQL (modèles, tests, etc.) étaient stockés uniquement dans l'état interne du pipeline (state["generated_files"]) sous forme de chaînes de texte produites par le LLM.

Ce nœud est responsable de transformer ces contenus virtuels en **fichiers réels** dans l'arborescence du projet dbt.

Il assure :

- la correspondance entre les spécifications (state["file_specs"]) et les contenus générés (state["generated_files"]>,
- la création des répertoires si nécessaire,
- l'écriture des fichiers sur disque avec l'extension adéquate (.sql, .yaml, etc.),
- la mise à jour de l'état du pipeline pour confirmer la réussite de l'opération.

En d'autres termes, c'est ce nœud qui rend le projet **exploitable par dbt**.

III.G.2 Structure et fonctionnement

Le nœud est construit à partir d'une **fonction fabrique** make_create_file(filetype, file_extension, log) qui retourne une fonction spécialisée pour chaque type de fichier à créer.

Étapes principales :

1. Filtrage par type

- On extrait de file_specs et generated_files uniquement les fichiers correspondant au type demandé (par ex. "models", "tests").

- Exemple : pour filetype="models", seuls les fichiers définis comme modèles seront pris en compte.

2. Vérification des contenus générés

- Si un chemin défini dans file_specs n'a pas de contenu correspondant dans generated_files, un avertissement est loggé et le fichier est ignoré.
- Cela permet d'éviter la création de fichiers vides ou incohérents.

3. Nettoyage du contenu

- Le contenu est passé à extract_code_block() pour retirer d'éventuelles balises Markdown (comme sql ...), souvent présentes dans les sorties LLM.

4. Construction du chemin complet

- À partir des informations dans le state et les variables par défaut (DEFAULT_DBT_PROJECT_NAME, DEFAULT_PROJECT_FOLDER), le chemin final est déterminé.
- Exemple :

<root>/<dbt_project_name>/<path_dbt_architecture>/<filename>.sql

5. Écriture du fichier

- Grâce à write_code_to_file(content, full_path), le fichier est créé ou écrasé.
- Les logs détaillent le succès ou l'échec de chaque écriture.

6. Mise à jour de l'état

- Après traitement de tous les fichiers du type, state["generation_status"] est mis à jour pour indiquer le succès global.

III.G.3 Exemple d'exécution

Lorsqu'il est exécuté, le noeud node_create_file parcourt la structure contenue dans state["file_specs"] et génère les fichiers correspondants sur le système.

Par exemple, supposons que state["file_specs"] contienne les spécifications suivantes (simplifiées) :

```

"models": {
    "models/staging/stg_expeditions": {
        "name": "stg_expeditions",
        "description": "A staging model for the EXPED table ...",
        "model_type": "stage",
        "path": "models/staging/",
        "columns": [
            {
                "name": "expid",
                "data_type": "TEXT",
                "description": "Expedition internal ID"
            },
            {
                "name": "year",
                "data_type": "FIXED",
                "description": "Year of the expedition"
            },
            ...
        ],
        "dependencies": "himalayan_expeditions_seeds.EXPED",
        "filetype": "models"
    },
    ...
    "models/marts/mart_expeditions": {
        "name": "mart_expeditions",
        "description": "A final marts model that contains the required columns ...",
        "model_type": "marts",
        "path": "models/marts/",
        "columns": [
            {
                "name": "expid",
                "data_type": "TEXT",
                "description": "Expedition internal ID"
            },
            {
                "name": "is_success",
                "data_type": "BOOLEAN",
                "description": "Boolean, Success or not of the expedition"
            },
            ...
        ],
        "dependencies": "int_expeditions_with_peaks",
        "filetype": "models"
    }
}

```

Partie « models » du state["file_specs"] (raccourcie)

```

"tests": {
    "tests/test_stg_peaks_unique.sql\\test_stg_peaks_unique": {
        "name": "test_stg_peaks_unique",
        "description": "Test that peakid is unique in stg_peaks model",
        "path": "tests/test_stg_peaks_unique.sql",
        "model": "stg_peaks",
        "dependencies": "stg_peaks",
        "filetype": "tests"
    },
    "tests/test_int_expeditions_with_peaks_relationship.sql\\test_int_expeditions_with_peaks Relationship": {
        "name": "test_int_expeditions_with_peaks Relationship",
        "description": "Test the relationship between int_expeditions_with_peaks and stg_expeditions models",
        "path": "tests/test_int_expeditions_with_peaks_relationship.sql",
        "model": "int_expeditions_with_peaks",
        "dependencies": "int_expeditions_with_peaks, stg_expeditions",
        "filetype": "tests"
    },
}

```

Partie « tests » du state["file_specs"] (raccourcie)

À partir de cette configuration, le nœud crée automatiquement :

- un fichier SQL `models/staging/stg_expeditions.sql` contenant la requête définissant le modèle de staging,
- un fichier SQL `models/marts/mart_expeditions.sql` pour le modèle de marts final,
- ainsi que deux fichiers de test dans les répertoires `tests/{nom_du_test.sql}/` (cela arrive assez souvent mais ce n'est pas gênant). Ils vérifient respectivement l'unicité des valeurs de la colonne correspondant à un ID (peakid) et la relation entre le fichier SQL intermediate et un des stages ("int_expeditions_with_peaks, stg_expeditions").

Ceci n'est qu'une sélection exhaustive parmi quatre modèles générés (2 stages, 1 intermediate et 1 marts) et huit tests générés (3 d'unicité, 3 de non-nullité, 2 de relation)

Le processus se déroule de manière générique : chaque entrée de `file_specs` est traduite en un fichier physique correspondant à son type (models, tests), avec le contenu conforme aux spécifications.

III.H Sauvegarde des générations – node_save_outputs

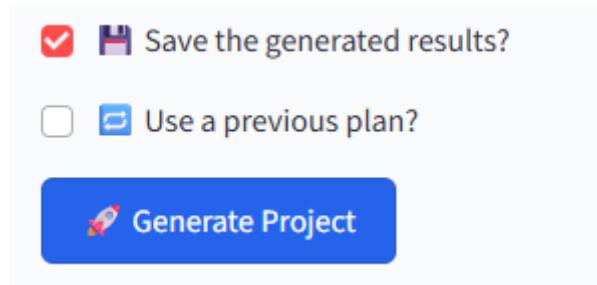
III.H.1 Rôle du nœud

Le nœud **node_save_outputs** a pour objectif de **sauvegarder sur disque** les artefacts générés lors de l'exécution du graphe LangGraph :

- le **plan de génération** (plan)
- les **fichiers dbt générés** (generated_files)

Cela permet de **réutiliser** ces sorties lors d'une exécution ultérieure, sans avoir à repasser par les étapes de génération IA (plan ou fichiers). On y gagne donc en :

- **Reproductibilité** : on peut recharger un plan/fichiers déjà générés.
- **Traçabilité** : chaque exécution conserve ses artefacts (pratique pour débogage).
- **Flexibilité** : l'utilisateur peut désactiver la sauvegarde via la case décochée.



Aperçu de l'option « Sauvegarde » sur l'application Streamlit

III.H.2 Structure du code

Le fichier `node_save_outputs.py` contient deux éléments principaux :

1. `make_save_outputs(log)`

- Fabrique une fonction `save_outputs(state)` compatible LangGraph.
- Sauvegarde les artefacts présents dans `state`.
- Utilise la fonction utilitaire `save_response` (définie dans `src/utils/utils.py`) pour écrire sur disque.

2. `decide_save(state)`

- Permet de déterminer si la sauvegarde doit être effectuée ou non, selon la valeur du booléen `state["save_project"]`.

- Retourne "save" ou "skip".

III.H.3 Logique d'exécution

a) Sauvegarde des résultats

Si `state["plan"]` existe (case cochée / vaut True), il est enregistré dans `{base_path}/saved/saved_plan.json`.

Si `state["generated_files"]` existe (case cochée / vaut True), ils sont enregistrés dans `{base_path}/saved/saved_files.json`.

On peut voir le détail dans la partie [I.G.2](#) où le dossier saved et son contenu sont plus explicitement détaillés.

b) Exemple de log

```
[11:54:21] [DEBUG] 📁 Saving plan and files ...
[11:54:21] [INFO] ✅ Plan saved to {base_path}/saved/saved_plan.json
[11:54:21] [INFO] ✅ Generated files saved to {base_path}/saved/saved_files.json
```

III.I Fin de processus – `nodes_end_process`

III.I.1 Rôle du nœud

Le nœud **end_process** est le dernier maillon du graphe LangGraph.

Son rôle est simple mais essentiel : **clôturer proprement l'exécution** du pipeline de génération DBT.

Il sert de **point terminal** pour :

- garantir que le graphe aboutit à une fin explicite,
- centraliser les informations utiles dans l'état (state),
- signaler aux logs que tout est terminé.

Sans ce nœud, le graphe pourrait s'arrêter implicitement après la sauvegarde, mais on perdrait en **lisibilité** et en **cohérence** d'exécution. De plus, on évoquera aussi le nœud **node_end_process_bad_plan_generation** qui est utilisé uniquement dans le cas particulier où la génération du plan a échoué après plusieurs tentatives, et qu'il est inutile de poursuivre.

III.I.2 Terminaison standard – node_end_process

a) Objectif

Le nœud `node_end_process` a pour mission de :

1. **Nettoyer l'état (state)** pour le rendre sérialisable (afin de pouvoir l'afficher, sauvegarder ou transmettre).
2. **Consolider les informations finales** : statut de génération, messages explicites pour l'utilisateur, éventuel résumé.
3. **Préparer un état sûr et traçable** à destination des systèmes externes (ex. : logs, Streamlit, sauvegarde).

C'est donc un nœud central pour garantir la robustesse et l'exploitabilité du pipeline.

b) Étapes de traitement

1. Journalisation de l'arrivée au nœud

- Log : "🏁 Termination node reached."
- Permet de savoir sans ambiguïté que le pipeline est arrivé à sa conclusion.

2. Nettoyage et sérialisation sécurisée

- Chaque valeur de `state` est passée par `safe_serialize` (fonction utilitaire qui gère les cas non sérialisables comme objets Python, instances personnalisées, etc.).
- L'état est ensuite vidé puis réinjecté avec ces valeurs sérialisées.
- But : éviter tout plantage lors du `json.dumps` final.

Exemple : une liste Python, un objet logger, ou un dictionnaire imbriqué sera converti en représentation string ou JSON-compatible. Cette étape fut nécessaire car l'utilisation de bouton de chargement de fichier sur l'application Streamlit crée des objets de type `UploadedFile` qui corrompent les dictionnaires (plus d'explication en [ANNEXE](#)).

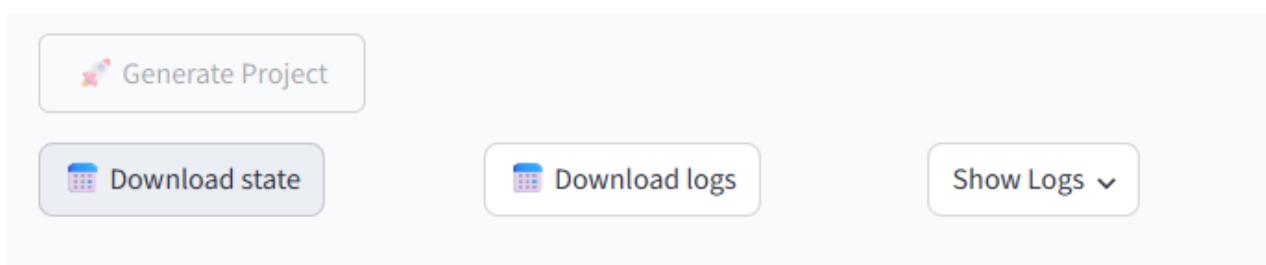
3. Détermination du statut de génération

- Lecture de state["generation_status"].
- Trois scénarios possibles :
 - "failed" → **Erreur générale**
 - Log : ✗ Generation process ended with errors.
 - Résumé : "⛔ The DBT project generation failed."
 - "html error" → **Erreur spécifique du LLM**
 - Log : ⚠ Generation ended with an HTML error from LLM.
 - Résumé : "⚠ LLM returned HTML error."
 - Autres cas → **Succès**
 - Log : ✓ Generation process completed successfully.
 - Résumé : "✓ The DBT project was generated successfully."

4. Production de l'état final

L'état (state) complet est de nouveau passé dans `safe_serialize`, cette fois avec `indent=2` et `ensure_ascii=False`, pour générer une version lisible et exploitable (UTF-8).

Le state est enfin affiché au complet dans les logs (en mode DEBUG), et disponible en téléchargement sur la page Streamlit (app.py).



c) Importance dans le pipeline

- **Robustesse** : le pipeline ne peut jamais se terminer avec un état (state) invalide ou non sérialisable.
- **Clarté** : chaque exécution se conclut par un message explicite (succès, erreur générique, erreur LLM). Le graphe a une étape claire de fin de processus.

- **Interopérabilité** : la sortie est au format JSON lisible, directement exploitable par l'UI ou des scripts d'automatisation.
- **Auditabilité** : l'état complet est loggé, facilitant le diagnostic en cas de problème.

III.I.3 Cas particulier – *node_end_process_bad_plan_generation*

a) *Objectif*

Il permet d'activer une terminaison spécifique lorsque la **génération du plan a échoué trop de fois**. Il empêche par conséquent le pipeline de continuer inutilement en produisant des fichiers à partir d'un plan inexistant. Il permet aussi au programme de ne pas tourner trop longtemps, car la **génération du plan** peut s'avérer **chronophage** et donc augmenter considérablement le temps global du processus.

b) *Fonctionnement*

- Récupère MAX_GENERATION_ATTEMPTS depuis l'état. Il ne peut être défini par l'utilisateur que dans les variables en dur, mais l'implémentation du choix dans le GUI est facile et ergonomique.
- Log l'erreur, en affichant le nombre d'essai (= MAX_GENERATION_ATTEMPTS arrivé dans ce nœud)

[11:55:21] [ERROR] ❌ Plan generation failed after 3 attempts. Ending process.

- Modifie le `state` pour y indiquer la fin de la génération.

ANNEXE

ANNEXE – DOCUMENTATION DETAILLEE DES FONCTIONS DE UTILS.PY

`save_response(content: str, output_path: str) -> None`

Cette fonction enregistre du contenu (généralement un dictionnaire ou un objet JSON) dans un fichier, en formatant le JSON de manière lisible (**indentation et UTF-8**).

- **Utilité dans le projet** : sauvegarder le plan généré ou d'autres artefacts intermédiaires du pipeline.
- **Exemple d'usage** : enregistrer state["plan"] dans outputs/plan.json pour le réutiliser ultérieurement.

`should_generate_object(state: State, object_type: str) -> bool`

Vérifie si un objet doit être régénéré ou si une version précédente doit être réutilisée, en fonction de l'état partagé (state). La logique repose sur des flags comme use_previous_plan ou use_previous_files.

- **Utilité dans le projet** : évite de régénérer inutilement des artefacts (plan, fichiers, etc.) si l'utilisateur a demandé leur réutilisation.
- **Exemple d'usage** : dans edge_select_plan.py, cette fonction détermine si le pipeline doit recalculer un plan ou charger un plan sauvégarde.

`get_time_with_precision(precision: int = 3) -> str`

Retourne l'heure courante au format HHMMSS.SSS, avec un nombre configurable de décimales pour les secondes.

- **Utilité dans le projet** : générer des **suffixes horodatés uniques** pour les projets, backups ou logs.
- **Exemple d'usage** : lors de la création d'un dossier de résultats, on peut ajouter l'horodatage 152045.123 pour éviter les collisions de noms.

`select_folder() -> str`

Ouvre une boîte de dialogue graphique (Tkinter) permettant à l'utilisateur de sélectionner un dossier.

- **Utilité dans le projet** : principalement utilisé en mode interactif (hors Streamlit), par exemple pour sélectionner un répertoire d'entrées ou d'outputs.
- **Exemple d'usage** : choisir manuellement un dossier inputs/ contenant un fichier de spécifications.

`delete_project(log, project_path: str) -> None`

Supprime un dossier projet donné et gère les exceptions fréquentes (inexistant, permissions, erreurs système).

Cette fonction interagit également avec **Streamlit** pour notifier l'utilisateur via des messages visuels (st.success, st.warning, st.error).

- **Utilité dans le projet** : réinitialiser un projet généré en cas d'erreur ou de relance.
- **Exemple d'usage** : lorsqu'une génération échoue et que le projet partiellement créé doit être supprimé avant de relancer le pipeline.

`safe_serialize(log, value, indent=None, ensure_ascii=None)`

Tente de sérialiser un objet en JSON.

Si l'objet n'est pas sérialisable (exemple : objets complexes ou non standards), la fonction renvoie sa représentation sous forme de chaîne (str(value)), et logge un avertissement.

- **Utilité dans le projet** : garantir la robustesse lors de la sauvegarde de l'état (State) ou des résultats intermédiaires.
- **Exemple d'usage** : éviter qu'un objet non JSON-compatible bloque l'écriture d'un état de graphe ou de logs.

`load_prompt_template(path: str, context: dict) -> str`

Charge un fichier **template Jinja2** et l'exécute en injectant les variables fournies via context.

- **Utilité dans le projet** : personnaliser dynamiquement les prompts envoyés au LLM en fonction des données d'entrée (colonnes, descriptions, plan, etc.).
- **Exemple d'usage** : charger un fichier formats/prompt_generate_plan.j2 et injecter les colonnes du fichier utilisateur pour générer un prompt complet destiné au modèle.

ANNEXE – EXEMPLE POUR INDICER OU TROUVER LES INFORMATIONS SNOWFLAKE

The screenshot illustrates the process of finding Snowflake account information. It shows the Snowflake navigation bar on the left and a detailed view of account settings on the right.

Navigation Bar (Left):

- Switch Role (dropdown: MY CUSTOM ROLE)
- Account (dropdown: MYACCOUNT)
 - [View account details](#) (highlighted with a red box)
 - Sign Into Another Account
- My profile
- Support
- Appearance
- Connect a tool to Snowflake
- Client download
- Documentation
- Privacy notice
- Classic console
- Sign Out

Account Details Page (Right):

Header: Account Details

Tabs: Account (selected), Config File, Connectors/Drivers, SQL Commands

| NAME | VALUE |
|-----------------------------------|--|
| Account Identifier ⓘ | SNOWFLAKE_ACCOUNT |
| Data Sharing Account Identifier ⓘ | SNOWFLAKE_ACCOUNT (avec un point au lieu d'un tiret) |
| Organization Name | Préfix du SNOWFLAKE_ACCOUNT (avant tiret) |
| Account Name | Suffixe du SNOWFLAKE_ACCOUNT (après tiret) |
| Account/Server URL | SNOWFLAKE_ACCOUNT.snowflakecomputing.com |
| User Name ⓘ | SNOWFLAKE_USER |
| Role | SNOWFLAKE_ROLE |
| Account Locator | SNOWFLAKE_ACCOUNT_LOCATOR (pas utile ici) |
| Cloud Platform | AWS |
| Edition | Standard |

Buttons: Learn More, Close

ANNEXE – MODELES DISPONIBLES SUR GROQ (LIEN CLIQUABLE)

| MODEL ID | RPM | RPD | TPM | TPD | ASH | ASD |
|---|-----|-------|------|------|------|-------|
| allam-2-7b | 30 | 7K | 6K | 500K | - | - |
| deepseek-r1-distill-llama-70b | 30 | 1K | 6K | 100K | - | - |
| gemma2-9b-it | 30 | 14.4K | 15K | 500K | - | - |
| groq/compound | 30 | 250 | 70K | - | - | - |
| groq/compound-mini | 30 | 250 | 70K | - | - | - |
| llama-3.1-8b-instant | 30 | 14.4K | 6K | 500K | - | - |
| llama-3.3-70b-versatile | 30 | 1K | 12K | 100K | - | - |
| meta-llama/llama-4-maverick-17b-128e-instruct | 30 | 1K | 6K | 500K | - | - |
| meta-llama/llama-4-scout-17b-16e-instruct | 30 | 1K | 30K | 500K | - | - |
| meta-llama/llama-guard-4-12b | 30 | 14.4K | 15K | 500K | - | - |
| meta-llama/llama-prompt-guard-2-22m | 30 | 14.4K | 15K | 500K | - | - |
| meta-llama/llama-prompt-guard-2-86m | 30 | 14.4K | 15K | 500K | - | - |
| moonshotai/kimi-k2-instruct | 60 | 1K | 10K | 300K | - | - |
| moonshotai/kimi-k2-instruct-0905 | 60 | 1K | 10K | 300K | - | - |
| openai/gpt-oss-120b | 30 | 1K | 8K | 200K | - | - |
| openai/gpt-oss-20b | 30 | 1K | 8K | 200K | - | - |
| playai-tts | 10 | 100 | 1.2K | 3.6K | - | - |
| playai-tts-arabic | 10 | 100 | 1.2K | 3.6K | - | - |
| qwen/qwen3-32b | 60 | 1K | 6K | 500K | - | - |
| whisper-large-v3 | 20 | 2K | - | - | 7.2K | 28.8K |
| whisper-large-v3-turbo | 20 | 2K | - | - | 7.2K | 28.8K |

MESURES :

- **RPM:** Requests per minute
- **RPD:** Requests per day
- **TPM:** Tokens per minute
- **TPD:** Tokens per day
- **ASH:** Audio seconds per hour
- **ASD:** Audio seconds per day

ANNEXE – ERREURS LIEES AU STATE

On retrouve deux erreurs qui ont été récurrentes dans l'affichage et le téléchargement du state, variable centrale du graphe qui permet de connaître pléthore d'information et qui est par conséquent utile pour le débogage.

- **Erreur 1 :** Les fichiers sélectionnés dans app.py sont des instances de la classe UploadedFile, comme suit :

```
UploadedFile(file_id='\c546ce82-7807-4a21-9591-3f65f4f73191',
name='saved_plan.json', type='application/json', size=8024,
_file_urls=file_id: "c546ce82-7807-4a21-9591-
3f65f4f73191"\nupload_url: "/_stcore/upload_file/f65cf54b-df0b-
40ff-adbf-963c53676953/c546ce82-7807-4a21-9591-
3f65f4f73191"\ndelete_url: "/_stcore/upload_file/f65cf54b-df0b-
40ff-adbf-963c53676953/c546ce82-7807-4a21-9591-3f65f4f73191"\n)
```

Lorsqu'on les stocke dans un dictionnaire, l'erreur suivante apparaît :

- ▶ Type error: Object of type function is not JSON serializable

Cela est dû aux **double quotes** et **parenthèses** (ici surlignées) qui ne sont **pas compatibles** avec les **dictionnaires** en l'état.

Une solution a été de caster cet object en string à la fin de la génération, mais la présence de simples et doubles quotes originalement ne le permettait pas facilement.

- **Erreur 2 :** La fonction qui gère le graphe est également sujette à un bug dans le state. Un warning est déclenché :

[WARNING] ⚠ The value of the key 'update_graph_status' is not json compatible: Value type = '<class 'function'>'

Comme précédemment, cela est sûrement dû aux quotes et parenthèses. La même solution que précédemment a été utilisée pour la fonction. Cependant à l'affichage des logs sur Streamlit, il y a simplement son nom car l'objet n'était pas affichable. Cela aurait dû être comme suit, mais par soucis de temps et de praticité (ce n'est pas important), il n'a pas été fait plus que ça.

```
<function update_graph_status at 0x000001D4A5C5CEA0>
```