DOSSIER DE VALIDATION

Ingénieur en science des données
spécialisé en infrastructure data
pour le Titre RNCP 39586

Bloc 1: Collecter, transformer et sécuriser des données

Nom Prénom	CORIN Gaëtan
Nom Prénom du tuteur	MOULIN Alexis
Niveau visé	RNCP 7
Date de la soutenance	Septembre 2025
Lieu de la soutenance	Toulouse

Table des matières

Introduction.

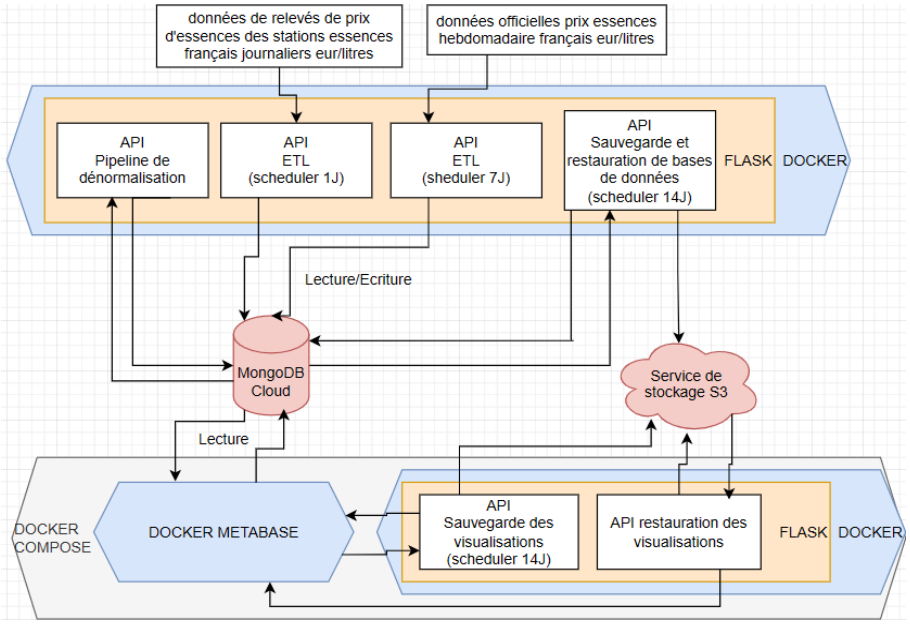
Je m'appelle Gaëtan Corin, j'ai 30 ans, et je suis en reconversion professionnelle (précédemment boulanger pâtissier). Je travaille actuellement en alternance chez Menaps, une startup sur Toulouse, où je travaille principalement sur des extractions et transformations de données, ainsi que des créations de pipelines permettant de formater les données pour des KPIs métiers. Malgré ma volonté très tôt de vouloir devenir Data-Engineer, mes années de licence 2 et 3 ont été principalement orientées vers un profil Fullstack, ce qui me permet aujourd'hui d'avoir une vision complète des projets.

Le projet que je vais vous présenter dans ce mémoire a été intégralement développé sur mon temps libre. Il est disponible sur mon github:
https://github.com/gaetancorin/Datapipeline_comparaison_official_vs_gas_stations_reporting
Ce mémoire a été intégralement écrit à la main sans générateur de texte ou LLM.

La problématique de ce projet vient d'une situation théorique, où le pôle data du gouvernement français possède un jeu de données officiel sur l'évolution du prix des différentes essences vendues en France. Ces données sont rassemblées en moyenne hebdomadaire. Malheureusement, ils ne connaissent pas la méthodologie qui a permis de calculer ces moyennes hebdomadaires à l'époque où celles-ci ont été calculées.

Ils se posent donc la question de la véracité et de la fiabilité des transformations des données anciennement recueillies, et souhaiteraient qu'un recalcul soit fait sur d'autres données journalières existantes issues des stations essence, afin de vérifier la crédibilité des transformations passées.
Ils aimeraient aussi une vision plus fine des cours des différentes essences vendues par les stations essence françaises, afin d'en tirer des conclusions et avoir une meilleure visibilité en terme journalière. Cet aperçu devra être mis à jour régulièrement.

Le cahier des charges a préalablement été réalisé dans le bloc 3 de ce titre RNCP.
Voici l'architecture du projet qui a été retenu:



C1.1.1 : Élaborer une stratégie de collecte de données en définissant les données utiles et nécessaires pour répondre à une problématique, et en identifiant les sources des données afin de cadrer le travail à réaliser pour collecter les données ciblées.

Au vu de la problématique expliquée plus haut, la stratégie déployée pour la collecte des données consiste à extraire les données de relevés de prix des stations essence journalières, puis à les traiter et les stocker en base de données.
Il faudra ensuite extraire les données officielles des prix des essences hebdomadaires, les traiter et les stocker en base de données.
Enfin, il va falloir créer une table de dénormalisation contenant les deux précédentes données transformées, afin de pouvoir facilement en faire des comparaisons et analyses lors des visualisations.

Pour les données de relevés des stations essence journalières:
Elles sont accessibles sur le site suivant: https://www.prix-carburants.gouv.fr/rubrique/opendata/

Il s'agit d'un site gouvernemental qui met à disposition ces données dans un objectif de transparence, de régulation du marché et d'information du consommateur.

Le formatage de ces données est sous format "zip", avec une URL disponible pour chaque année de données. Il faudra donc récupérer l'ensemble des données en faisant des appels API vers chacune des URLs, afin de récupérer l'intégralité des données.Il faudra ensuite dézipper chaque fichier récupéré, réaliser les transformations nécessaires, puis les stocker en base de données.

L'ensemble des données brutes est au nombre de 56 600 000 enregistrements, car chaque modification de prix au cours de la journée est répertoriée pour chaque station essence de 2007 à aujourd'hui.

Pour les données officielles des prix des essences hebdomadaires:
Elles sont accessibles sur le site suivant:
https://www.ecologie.gouv.fr/politiques-publiques/prix-produits-petroliers

Il s'agit d'un autre site gouvernemental qui met à disposition ces données, cette fois-ci dans un but écologique et de compréhension des différentes tranches de composition de prix des essences.

Une complexité est présente pour extraire ces données. En effet, un formulaire en Ajax doit être rempli et validé pour avoir accès à un élément HTML dynamique contenant une URL constituée d'un UUID. C'est cette URL qui devra être scrappée pour accéder aux données. La plus grande complexité se situe donc dans le formulaire en Ajax, qui est un formulaire dynamique Javascript.

Il faut donc un bot avec un simulateur de navigateur qui doit réaliser la connexion, remplir le formulaire et le valider afin de pouvoir accéder à l'URL à scrapper.
Il faudra par la suite appeler l'URL pour récupérer le fichier de données, réaliser les transformations nécessaires, puis les stocker en base de données.

Pour la table de dénormalisation:
Il s'agit d'une table constituée des deux types de données précédentes. La colonne de la date servira de jointure. Il faudra donc récupérer les deux types de données en bases de données, réaliser les éventuelles transformations dont la jointure, puis la stocker sous format de collection en base de données pour faciliter les visualisations et analyses futures.

C1.1.2 : Mettre en œuvre des techniques de collecte de données en exploitant les API externes et les bases de données disponibles, des techniques de web crawling et de web scraping afin de recueillir les données ciblées.

Trois techniques de collecte de données sont utilisées dans ce projet

La première technique est réalisée en méthode de scraping sur les données officielles.
Elle consiste à utiliser un bot réalisé avec la librairie python Selenium. Ce bot va simuler une navigation Chrome, naviguer sur le formulaire dynamique, remplir les champs les uns après les autres , puis valider le formulaire dans l'objectif d'accéder à l'élément HTML dynamique contenant l'URL constituée d'un UUID.
Il va ensuite scraper l'HTML pour récupérer cette URL.
Une vérification de résultat à chaque étape du bot nous permet de définir son bon fonctionnement, jusqu'à l'existence de l'élément HTML dynamique où nous pourrons scraper l'URL voulu. Une vérification de formatage de l'URL scrapée est effectuée afin d'assurer son exactitude, et son futur appel sera contrôlé par le status code ainsi que les colonnes récupérées, afin de vérifier l'exhaustivité.

La seconde technique est réalisée en appels API sur les données de relevés de stations essence. Elle consiste à récupérer les données en appelant les URLs des années concernées, puis à les dézipper. Pour réaliser cela, les librairies "Requests" et "Zipfile" de python sont utilisées.
Une vérification des statuts codes est réalisée sur chacune des URLs, afin d'être certain de ne pas avoir d'année manquante sur la base de données finale et d'assurer l'exactitude de ces informations. Une vérification est aussi réalisée lors de la transformation du fichier zip en fichier xml, puis csv. De plus, une vérification des colonnes ciblées est contrôlée sur chaque csv représentant une année afin de s'assurer de l'exhaustivité finale des données.

La troisième technique est réalisée en requêtes Pymongo lors de la dénormalisation des deux types de données précédentes. Ces données étant désormais stockées en bases de données, il va falloir appeler les collections concernées directement sur la base de données en réalisant des requêtes, récupérer les données de ces collections en vérifiant l'exactitude des données récupérées, les traiter en dénormalisant ces données, puis les stocker dans une nouvelle collection.

C1.1.3 : Automatiser la collecte de données en mettant en place des tâches planifiées et/ou des flux temps réel, en utilisant des logiciels d'automatisation afin de garantir l'actualisation des données.

Au sein du serveur Flask contenant la logique métier des ETLs, il y a deux planificateurs de tâches réalisés avec la librairie APScheduler de flask_apscheduler.

Le premier planificateur de tâche consiste à réaliser de manière automatique des sauvegardes de base de données vers un espace de stockage cloud (Amazon S3). Ceci ajoute une sécurité supplémentaire aux données en cas de cyberattaque, ransomware, ou bien en cas d'erreur sur la base de données. Ce planificateur se lance toutes les 2 semaines, après le chargement journalier.

Le second planificateur de tâche est celui qui va exécuter l'ensemble des fonctions d'ETLs. Pour chacune de ces fonctions ETLs, un programme est appelé juste avant l'extraction, qui va aller vérifier dans la future collection quelles sont les données existantes déjà traitées. Cela permet ainsi de ne pas charger l'entièreté des données, mais uniquement les données manquantes, dans une optique de mise à jour et dans le respect du RSE.

```
@scheduler.task('cron', id='complete_pipeline_oil_prices',year='*', month='*', day='*', week='*',
                day_of_week='*', hour='10', minute='0', second='0')
@app.route('/etl/launch_complete_pipeline_oil_prices', methods=["POST"])
def api_launch_complete_pipeline_oil_prices():
```

C1.2.1 : Élaborer la stratégie de stockage des données avec un modèle de données adéquat en intégrant les différents types de données, l'utilisation envisagée (analyse, stockage, disponibilité, accessibilité) et le volume à stocker afin d'organiser le stockage des données.

Il y a deux stratégies de stockage des données sur cette application car il y a deux types de besoins différents.

La première stratégie de stockage vient d'une volonté de **stocker les données brutes** dans la mesure du possible sur un principe de **Datalake**, afin de pouvoir être utilisées dans des cas métiers n'ayant pas encore été définis. Cette volonté de stockage ne doit pas être dirigée par un cas de visualisation spécifique que l'on souhaite implémenter.

Pour cette première stratégie, les données officielles des prix des essences n'ont pratiquement subi aucune transformation. La colonne de date a été modifiée, les noms des colonnes ont été modifiés, puis les données ont directement été chargées en base de données.
Ayant une quantité de données relativement faible de données (2000 enregistrements), aucune modification supplémentaire n'a été nécessaire.

{
_id: 685ea7d9984bc1281b1967ad (type ObjectId)
Date: 2008-01-04T00:00:00.000+00:00 (type Date)
official_ttc_GAZOLE_eur_liter: 1.2226 (type float)
official_ttc_SP95_eur_liter: 1.3701 (type float)
official_ttc_E10_eur_liter: NaN (type float)
official_ttc_SP98_eur_liter: 1.4007 (type float)
official_ttc_E85_eur_liter: NaN (type float)
official_ttc_GPLC_eur_liter: 0.7472 (type float)
}

En revanche, sur les données relevées par les stations essence, le format xml d'origine est un fichier en architecture où chaque station essence possède un bloc de données correspondant à l'ensemble de ces relevés de prix pour chaque type d'essence. Il s'agit d'une quantité énorme de données par station essence, car nous sommes à 56 millions d'enregistrements pour 13 600 stations sur la période de 2007 à 2025.

Afin d'alléger la quantité d'enregistrements et garantir l'intégrité des données, un nettoyage des données non pertinentes, ainsi que la conservation uniquement des derniers prix relevés par jour et par type d'essence ont été réalisés. Cela nous fait désormais passer à 39 millions de données. Il y a ensuite une division de ce jeu de données en deux entités.

La première représente les stations essence. Ces données changent rarement, et ne sont pas cumulées mais mises à jour en base de données par date de parution.

La deuxième entité représente les données des modifications de prix par jour et type d'essence sur une station essence. Chaque nouvel enregistrement est relié à la station essence par son id_station_essence unique, et ces données sont accumulées en base de données.

{
_id: 685ea3f5984bc1281bbdba53 (type ObjectId)
Id_station_essence: 1000001 (type int)
Adresse: "596 AVENUE DE TREVOUX" (type string)
Ville: "SAINT-DENIS-LèS-BOURG" (type string)
Cp: 1000 (type int)
Latitude: 4620100 (type int)
Longitude: 519800 (type int)
Derniere_maj: 2025-06-25T00:00:00.000+00:00 (type Date)
}

{
_id: 685ea3fd984bc1281bbdd80c (type ObjectId)
Date: 2009-01-05T00:00:00.000+00:00 (type Date)
Id_station_essence: 1000001 (type int)
Nom: "Gazole" (type string)
Valeur: 0.909 (type float)
Heuremin: "08:53" (type float)
}

La seconde stratégie de stockage a pour objectif de réaliser des transformations sur les données dans le Datalake afin de les formater pour être adaptées en analyses et en visualisation. Le travail sur ces données est donc réalisé sur un principe de **dénormalisation**.

La première dénormalisation a lieu sur la collection des prix des stations essence par type d'essence, jour et station essence. Ce jeu de données contient encore 39 millions d'enregistrements. L'objectif est donc de rassembler ces données en un seul enregistrement par jour, contenant la moyenne des prix de toutes les stations essence sur chaque type d'essence afin de n'en faire plus qu'un seul enregistrement, créant un total de 6 700 enregistrements.

{
_id: 685ea7df984bc1281b196b3d (type ObjectId)
Date: 2009-01-01T00:00:00.000+00:00 (type Date)
station_ttc_E85_eur_liter: 0.8279 (type float)
station_ttc_GPLC_eur_liter: NaN (type float)
station_ttc_GAZOLE_eur_liter: 1.10647 (type float)
station_ttc_SP95_eur_liter: 1.24179 (type float)
station_ttc_E10_eur_liter: 1.13348 (type float)
station_ttc_SP98_eur_liter: NaN (type float)
}

Enfin, la seconde dénormalisation consiste à rassembler ensemble les données des prix d'essences officiels avec les données des prix d'essences des stations essence dénormalisées. Pour faciliter les analyses et la visualisation, des colonnes supplémentaires sont aussi ajoutées.

{
_id: 685eaa1a984bc1281b198bdd (type ObjectId)
Date: 2008-03-28T00:00:00.000+00:00 (type Date)
official_ttc_GAZOLE_eur_liter: 1.2587 (type float)
official_ttc_SP95_eur_liter: 1.3656 (type float)
official_ttc_E10_eur_liter: NaN (type float)
official_ttc_SP98_eur_liter: 1.3993 (type float)
official_ttc_E85_eur_liter: NaN (type float)
official_ttc_GPLC_eur_liter: 0.7494 (type float)
station_ttc_GAZOLE_eur_liter: 1.23422 (type float)
station_ttc_SP95_eur_liter: 1.36403 (type float)
station_ttc_E10_eur_liter: NaN (type float)
station_ttc_SP98_eur_liter: NaN (type float)
station_ttc_E85_eur_liter: 0.81043 (type float)
station_ttc_GPLC_eur_liter: 0.73473 (type float)
Day_of_week: "Friday" (type string)
Month: "March" (type string)
Year: "2008" (type string)
DayMonth: "28march" (type string)
}

Pour chaque ETL mis en œuvre, les données sont mises à jour en base de données de manière incrémentale en ne chargeant que les données manquantes. Cela permet d'éviter de recharger l'ensemble des données à chaque lancement de scheduler, et permet d'assurer l'intégrité des données en évitant les valeurs en doublons.

Concernant les moyens de manipulation de ces données, il est obligatoire de passer par les différents points d'APIs afin de réaliser des transformations. Enfin, en cas de désir de réinitialisation d'une base de données ou d'une collection, des APIs dédiées permettent de supprimer les données voulues de manière sécurisée sans toucher directement à la base de données.

Afin d'assurer l'intégrité des données, différents droits sont donnés suivant les besoins. Le docker d'ETL aura le droit de lecture et d'écriture sur l'ensemble des bases de données, tandis que le docker de visualisation n'aura que le droit de lecture, et cela uniquement sur les tables de dénormalisation destinées à la visualisation.

C1.2.2 : Construire une base de données en sélectionnant la technologie (SQL, NoSQL), un système de gestion de base de données (SGBD) et/ou une solution de stockage BIG DATA, en assurant le paramétrage et l'implémentation afin de mettre en œuvre le modèle de données qui garantit la disponibilité et l'intégrité des données.

Les données seront réparties en deux bases de données.

La première base de données a une fonction de type **"Datalake"**. Les prérequis de cette base de données sont qu'elle doit supporter tout type de données, qu'il soit possible de créer des clusters de données si la taille du datalake le demande, et que la recherche soit réalisée par indexation afin d'obtenir rapidement des résultats dans des grandes quantités de données. Les données traitées peuvent être relationnelles mais les queries ne sont pas censées être réalisées directement sur la base de données, car le stockage est optimisé pour le volume et non pour la performance.

La seconde base de données a une fonction de stockage de données **"Dénormaliser"**. Elle doit être disponible pour les outils de visualisation et d'analyse en ayant de nombreux connecteurs existants entre ces outils et la base de données. Elle doit aussi avoir la possibilité de séparer les droits de lecture uniquement sur cette base de données sans pouvoir accéder au Datalake, et d'avoir un cloud facile d'accès. Ces tables ne sont pas censées être relationnelles, mais plutôt directement prêtes à l'emploi pour la visualisation et l'analyse.

Au vu des besoins et des attentes, MongoDB semble être le système de gestion de bases de données répondant le mieux aux besoins nécessaires au stockage dans la base de données "Datalake", ainsi qu'aux besoins nécessaires pour la visualisation dans la base de données "Denormaliser".

Ce SGBD aura donc deux users. Le premier, destiné au docker des ETLs, qui aura un droit de lecture et d'écriture sur toutes les bases de données, et le second, destiné au docker de visualisation, qui n'aura que le droit de lecture sur la base de données de dénormalisation.

C1.3.1 : Sélectionner les technologies et les outils de traitement de données en identifiant les solutions existantes et en comparant leurs avantages et leurs inconvénients afin de traiter efficacement les données collectées.

Voici un tableau répertoriant les différentes possibilités de langages pour cette application.

	Python	Scala	SQL
Avantage	Syntaxe simple, bonne flexibilité avec de nombreuses bibliothèques	traitement parallèle, traitements distribués	manipulation directe de bases relationnelles, Technologie mature
Inconvénient	moins performant que scala sur les traitements distribués	complexité syntaxique, faibles ressources pédagogiques	limité aux bases de données relationnelles

Comme nous l'avons vu précédemment, le système de gestion de bases de données contient un datalake qui doit contenir tous types de données. Le SQL, ne fonctionnant que sur des bases de données relationnelles avec une rigidité du schéma des données, cela va très vite poser des limites sur le datalake qui n'est pas souhaitables dans notre application.

Scala est un excellent outil avec ses traitements parallèles et distribués et aurait été très utile lors du chargement des 39 millions de données des relevés de prix de stations essence. Mais il faut retenir que ce chargement n'a lieu qu'au premier lancement de l'application lorsque la collection de ces données est encore vide, afin de récupérer de l'année 2007 a 2025. Par la suite, le chargement se fait quotidiennement et les données se comptent simplement en milliers d'enregistrements par jour. Ce serait donc une complexification excessive de déployer Scala avec du traitement distribué pour ce cas d'usage.

Python, avec sa facilité d'utilisation, sa bonne documentation et ses nombreuses bibliothèques, semble donc être le candidat idéal pour cette application. Lors de la première exécution, un système de batch devra être mis en place pour éviter la saturation de la mémoire lors du chargement, transformations et chargement des 39 millions de données. Par la suite, il sera parfaitement adapté à la tâche demandée.

Il a ensuite fallu faire le choix entre Python Spark et Python Pandas.

Et pour la même raison que pour Scala et Python, les avantages que propose Python Spark sont des fonctionnalités de traitement distribuées intéressantes lors du premier chargement, mais qui n'a plus d'utilité lors de tâches journalières.
Python Pandas semble donc être un meilleur candidat afin d'éviter la complexité excessive qui pourrait devenir source d'erreurs.

Le serveur Flask a aussi été choisi en comparaison à Django car sur ce projet, il n'y a pas de nécessité d'avoir un frontend ni d'interface administration.
Le choix de la simplification a donc été porté sur Flask, un framework pour API plus simple et flexible répondant aux cas d'usages demandés.

C1.3.2 : Transformer les données à l'aide de langage de programmation ou en utilisant des outils dédiés (Talend, Spark…) afin d'obtenir des données nettoyées exploitables.

Voici les différentes étapes de transformations réalisées sur les 56 millions de données de relevés de stations essence jusqu'au chargement en base de données:

En premier lieu, une vérification en base de données permet de connaître les dernières données existantes sur la collection des relevés de stations essence.

Ayant connaissance de ces dernières données existantes, on détermine les dates des données dont nous avons besoin, soit l'ensemble des données entre la dernière date chargée et la date d'aujourd'hui.
Malheureusement, les données relevées par les stations essence sont récupérées via les fichiers zip annuels, et il n'y a pas de moyen existant pour ne récupérer que les dates voulues. Il faut donc récupérer l'année ou les années que nous souhaitons traiter.
Par la suite, le ou les fichiers sont dézippés, puis transformés en fichier csv. Nous filtrons ensuite uniquement sur les dates voulues.

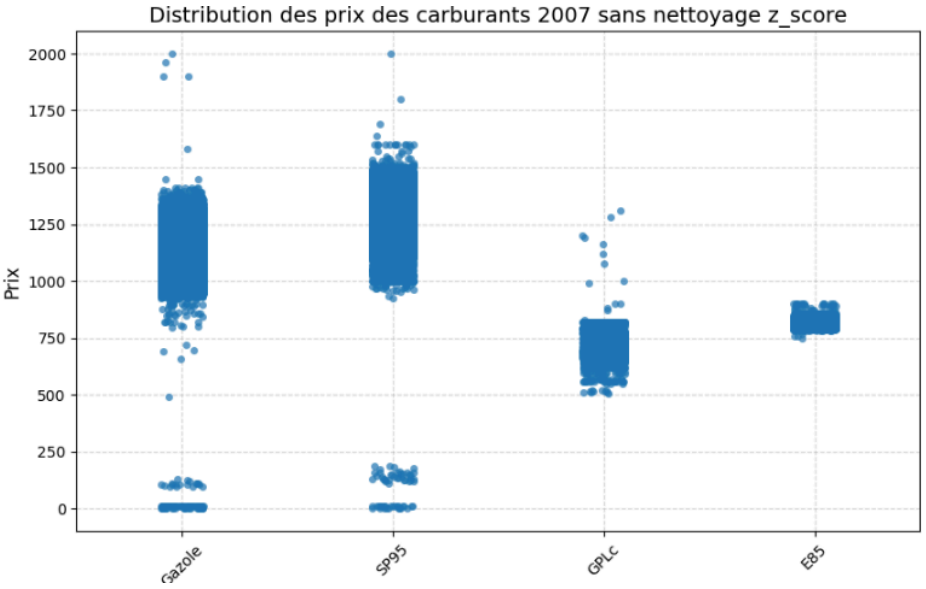
L'essentiel des transformations est ensuite réalisé en Python Pandas.
Un nettoyage de différentes colonnes est réalisé, comme par exemple le retrait des retours à la ligne sur la colonne "adresse", l'ensemble des prix des essences en type float ainsi que la standardisation de trois types de format de dates différents suivant les années en un seul.

Il semble il y avoir une erreur sur un enregistrement d'une station essence sur l'année 2008, car la station essence avec l'id" "35200004" au code postal "35***",n'a jamais eu de relevé de prix d'essence, et n'apparaît qu'une seule fois en cette année 2008. Elle n'apparaît plus sur les autres années. Le retrait de cette donnée est donc réalisé car jugée peu pertinente.

Le prix des essences était compris entre 500 à 2500 de la date de 2007 à 2021, puis de 0.5 à 2.5 de 2022 à aujourd'hui. je formate donc l'ensemble des valeurs entre 0.5 et 2.5, dans un souci de logique métier.

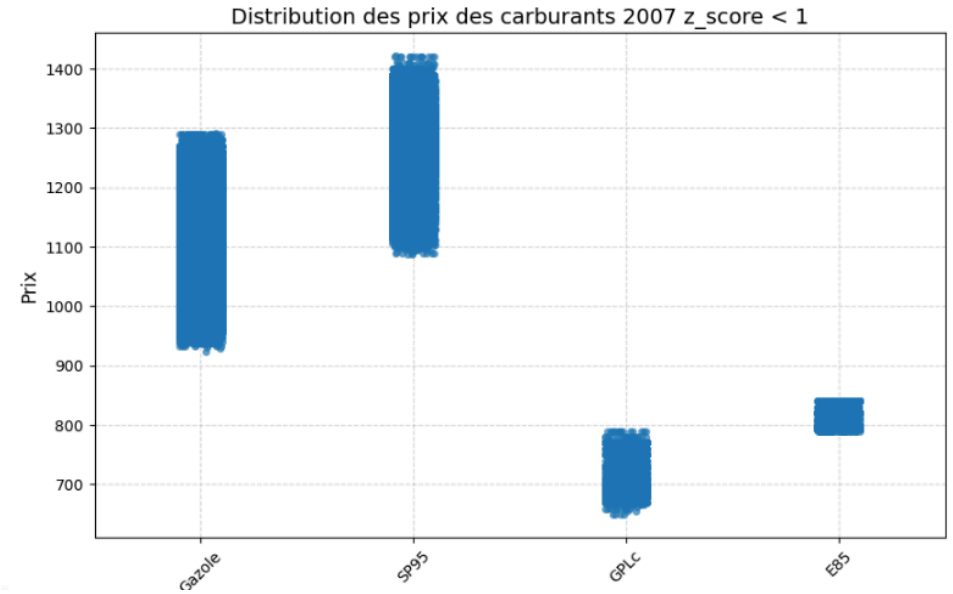
Enfin, un problème de cohérence apparaît dans mon jeu de données.
Les données relevées par les stations essence ne sont pas réellement les prix vendus, mais

bien les prix affichés sur le panneau de présentation. Chaque modification sur le panneau est relevée de manière automatique et se retrouve dans le jeu de données.
Il arrive donc des situations où les pompistes réalisent des tests sur leurs panneaux en affichant 0, 10 centimes ou 2 euros, et ces données finissent dans le jeu de données.



Heureusement, les données métiers des prix des essences sont très homogènes et pour chaque journée, le prix d'un type d'essence est relativement bien défini sur l'ensemble des stations essence de France, avec une amplitude de prix assez faible.

Il est donc très efficace de réaliser un écart-type entre les valeurs pour une même journée et un même type d'essence, puis de retirer toutes les valeurs s'écartant un peu trop de cet écart-type (z-score). Le résultat est très efficace.



Enfin, on réduit la taille du jeu de données en ne conservant que les dernières données de chaque journée pour chaque type d'essence et chaque station.

On divise ensuite le jeu de données en deux dataframes différents.

Le premier sera les informations sur les stations essence avec les colonnes *'Id_station_essence', 'Adresse', 'Ville', 'Cp', 'Latitude', 'Longitude', 'Date'*

Le second sera les informations de relevés des prix par stations essence avec les colonnes *'Date', 'Id_station_essence', 'Nom', 'Valeur'* (la colonne Id_station_essence permet de faire le lien avec le dataframe de station essence)

Les deux dataframes sont ensuite prêts à être exploités. Ils sont donc désormais stockés vers leurs bases de données respectives:

- "gas_stations_infos" pour les stations essence, sur un principe de mise à jour des données ou la donnée la plus récente prend le droit d'écriture.
- "gas_stations_price_logs_eur" pour les prix des essences qui sont stockés sur la base de données sur un système incrémental.

C1.3.3 : Développer un processus ETL en identifiant les bénéfices des technologies ETL (ex : facilité de développement), en exploitant la technologie ETL préalablement sélectionnée afin d'automatiser l'extraction, la transformation et le chargement de données.

Pour réaliser les différentes étapes d'extraction, de transformation et de chargement en base de données, le framework Flask est utilisé, avec le langage Python et la bibliothèque Pandas.

En effet, Flask est un framework très simple, facile à prendre en main et très modulaire. Il peut faire fonctionner de très nombreux outils en Python, ce qui lui donne la capacité à s'adapter à la plupart des situations et des besoins tout en offrant ses modules d'APIs ainsi que ses outils internes.

Fonctionnant en Python, l'import d'outils et de librairies destinés à la gestion de données est réalisé de manière très simple. Flask permet aussi de facilement intégrer du code Python qui aurait été développé sur des fichiers notebooks par exemple, ce qui permet de facilement passer d'un code en version Proof of Concept à un code en version Industrialisation.

Le Framework Flask fonctionne très bien avec la librairie Python Pandas, mais il a aussi la possibilité de travailler avec d'autres librairies de traitement de données en traitement parallèle ou traitement distribué comme Python Spark par exemple. Cela offre une grande modularité et adaptabilité.

Flask possède aussi sa librairie interne pour réaliser des planificateurs de tâches, avec l'outil "APScheduler" de la librairie "flask_apscheduler". Lors de déploiements de pipelines ETLs,

cela permet de lancer la pipeline de données de manière automatique suivant la période définie.

Le framework Flask fonctionne très bien avec les fichiers .env, ainsi qu'avec les fichiers requirements.txt qui permettent de créer des environnements virtuels où Flask va fonctionner. Il est donc très facile de construire un projet d'architecture facilement dockerisable, et déployable chez le client.

Voici l'architecture des fonctions du Flask qui est actuellement déployé sur la partie ETL: (Il y a deux types de fonctions, celles permettant d'être appelées en API en bleu, et celles qui sont lancées par les APIs en noir)

/etl/launch_complete_pipeline_oils_prices
- /etl/launch_etl_gas_stations_oils_prices
  - determine_dates_to_load_from_mongo()
  - extract_new_gas_stations_oils_prices()
  - transform_gas_stations_oils_prices()
  - load_gas_stations_oils_prices_to_mongo()
- /etl/launch_etl_official_oils_prices
  - determine_dates_to_load_from_mongo()
  - extract_new_official_oils_prices()
  - transform_official_oils_prices()
  - load_official_oils_prices_to_mongo()
- /dataviz/denormalize_station_prices_for_dataviz
  - determine_dates_to_load_from_mongo()
  - extract_new_station_prices_from_mongo()
  - transform_and_denormalize_station_prices()
  - load_denormalized_station_prices()
- /dataviz/merge_denorm_station_vs_official_prices
  - determine_dates_to_load_from_mongo()
  - extract_new_denorm_station_prices_from_mongo()
  - extract_new_official_oils_prices_from_mongo()
  - transform_merge_station_vs_official_prices()
  - load_denorm_station_vs_official_prices()

/mongo/drop_one_collection
/mongo/drop_one_bdd
/utils/save_mongo_dump_to_S3
/utils/list_S3_contents
/utils/restore_mongo_dump_from_S3
/is_alive

L'indentation d'appels APIs permet de comprendre que certaines fonctions qui sont déjà dans une API globalisée peuvent elles aussi être appelées en API séparément.

Sur cette architecture de projet ETLs, nous retrouvons bien distinctement les étapes ETLs avec les fonctions commençant par le mot “extract”, “transform” ou “load”. Ces étapes ont pour chacune leur propre API afin de laisser à l'utilisateur la possibilité de ne réaliser que ces transformations. Elles peuvent aussi être appelées avec l'API globale qui permet de tout lancer.

Avant chaque extraction de données, nous retrouvons la fonction “determine_dates_to_load_from_mongo()” qui permet de définir les dates manquantes entre la dernière donnée sur la collection et la date d’aujourd’hui. Cela représente donc les données actuellement manquantes en bases de données et qui doivent être extraites, transformées et chargées sur la collection.

Les planificateurs de tâches sont déclenchés sur deux APIs:

Le premier planificateur de tâches est installé sur la fonction de l’appel API “/etl/launch_complete_pipeline_oils_prices”. C’est lui qui va réaliser l’intégralité des parties ETLs, et s’assurer que l’ensemble des données sont mis à jour de manière quotidienne.

Le second planificateur de tâches est installé sur la fonction de l’appel API “/utils/save_mongo_dump_to_S3”. C’est lui qui va réaliser des sauvegardes automatiques des collections de MongoDB vers l’espace de stockage S3.

C1.4.1 : Définir la politique de sécurisation des données en évaluant les risques, en qualifiant leur niveau de sensibilité, en identifiant les droits d’accès selon les rôles des différentes parties prenantes et en respectant les exigences légales (ex : RGPD) afin de garantir la bonne utilisation et l’intégrité des données.

La politique de sécurisation des données doit être adaptée aux données qui sont traitées et de leur sensibilité.

Sur mon application, deux types de données sont traités:

- Le premier type de données représente des moyennes françaises de prix d'essence en données mensuelles. Le fait que ces données soient des moyennes anonymise les différents acteurs ayant participé à cette moyenne. Les données respectent donc naturellement les règles du RGPD de la confidentialité des données individuelles et des entreprises. De plus, le fait que ces données soient des données fournies par le gouvernement en OpenData nous valide que les règles précédemment énoncées sont respectées.
- Le second type de données représente les prix des stations essences de France de manière journalière. Si nous allons sur le site gouvernemental où ces données sont récupérés: https://www.prix-carburants.gouv.fr/rubrique/opendata/, sur la foire aux questions, à la “question 5 Ce qui est exclu des données”, nous apprenons que ces données publiques ne mettent pas à disposition “le nom des stations” ainsi que “la marque de stations”. Le site gouvernemental nous laisse donc suggérer que ces données ont déjà été anonymisées afin de respecter les exigences légales. De plus, le fait que

ces données soient là aussi des données fournies par le gouvernement en OpenData nous valide que les règles de RGPD sont respectées.

De manière censée, nous pouvons donc en conclure que des traitements de prix d'essence, si on ne connaît pas la source exacte de ces données, sont naturellement considérés comme un faible risque en termes de données sensibles.

En prenant en considération l'étude des données que l'on vient de réaliser, nous pouvons en déduire qu'il n'y a pas de nécessité d'anonymiser ou de faire de chiffrement de données. Une vérification de la sensibilité de chaque colonne des jeux de données utilisés doit quand même être réalisée afin de confirmer cette étude.

Enfin, pour garantir la bonne utilisation et l'intégrité des données, une différenciation de rôle est réalisée sur la connexion et la disponibilité des données en base de données MongoDB:

- La partie dockerisée qui réalise les ETLs aura un accès MongoDB avec un rôle permettant d'accéder à l'ensemble des bases de données en droit d'écriture et de lecture sur lequel elle doit réaliser des traitements.
- La partie dockerisée qui réalise la visualisation avec Métabase aura un accès limité à MongoDB, avec un rôle permettant uniquement d'accéder aux données dénormalisées destiné à la visualisation. Cette partie n'aura donc pas accès à la partie datalake. De plus, les droits seront uniquement en lecture afin de garantir l'intégrité des données.

Un autre moyen permettant de sécuriser les données consiste à réaliser des sauvegardes régulières de la base de données vers un environnement externe, tel qu'un espace de stockage cloud. Cela permet d'assurer la sauvegarde des données en cas de cyberattaque ou de ransomware par exemple.
Pour cette application, une sauvegarde complète et historisée des données est réalisée en format de fichier zip sur le cloud Amazon S3. Un planificateur de tâches permet de réaliser cette tâche toutes les deux semaines et de l'historiser dans le cloud avec le nom du contenu et la date en nom de fichier zip.

Enfin, le fait que lors de la conception des serveurs Flask, aucune API ne retourne de données permet une sécurisation forte des données.

C1.4.2 : Concevoir une architecture sécurisée et robuste, en intégrant des mesures de sécurité multicouches et des contrôles d'accès stricts, en mettant en place des solutions de protection des données (ex : chiffrement) pour sécuriser les données en transit et au repos, en veillant à l'anonymisation des données personnelles afin de répondre aux normes de protection de la vie privée et de sécurité des données.

Voici le schéma d'architecture de sécurité des données.

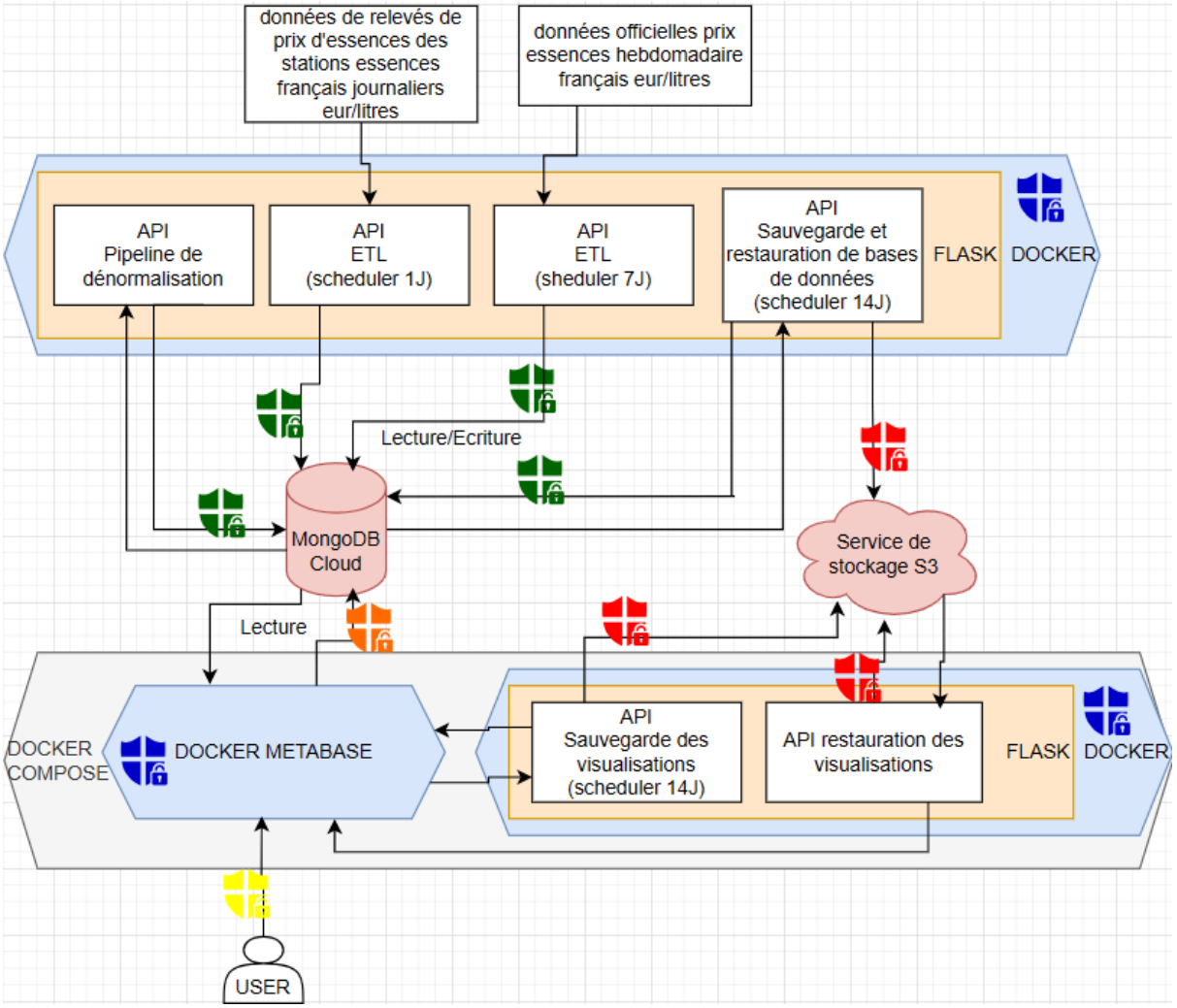
Ce schéma contient les jeux de données sources, le dockerfile contenant le serveur Flask des ETLs avec ses fonctions ainsi que ses flux, la base de données en cloud MongoDB

ainsi que l'espace de stockage cloud S3, avec pour chacun leurs flux respectifs.
Il contient aussi le docker-compose contenant deux containers dockers:

- Le premier étant le container de Métabase contenant l'ensemble des graphiques et des visualisations
- et le second étant le container du serveur flask avec les fonctions de gestion de sauvegarde de ces graphiques et visualisations de Métabase vers l'espace de stockage S3.

Nous avons là aussi pour chacun les différents flux de données traités.
Enfin nous avons l'utilisateur final qui n'a accès qu'à Métabase avec son flux respectif.

Voici les différents moyens mis en oeuvres pour sécuriser les données, représentés par les boucliers de couleurs par chaque type de sécurisation :



bouclier vert: Ces boucliers sont sur les flux entre le serveur Flask des ETLs et la base de données MongoDB. Ils représentent la sécurisation de la base de données avec une

connexion en user ayant le droit de lecture et d'écriture sur l'ensemble des bases de données concernées par les ETLs.

 bouclier orange: Ce bouclier est sur le flux entre le container Métabase et la base de données MongoDB. Il représentent la sécurisation de la base de données avec une connexion en user ayant le droit de lecture uniquement, et seulement sur la base de données gérant les collections dénormalisées destinées à la visualisation.

 bouclier rouge: Ces boucliers sont sur les flux entre le serveur Flask des ETLs et le service de stockage S3, ainsi que entre le serveur Flask de la gestion des sauvegardes de visualisations Métabase et le service de stockage S3. Ils représentent la sécurisation du service de stockage avec l'obligation de s'identifier avec une clé et un mot de passe pour pouvoir avoir un accès d'écriture et de lecture sur le S3.

 bouclier bleu: Ce bouclier est directement dans le fonctionnement natif des containers dockers et des docker-compose. En effet, les containers dockers et docker-compose ne permettent d'accéder que sur les ports qui sont ouverts lors de la création de ces containers. Ainsi, cela permet de sécuriser l'ensemble des autres ports qui ne sont pas censés être accessibles aux personnes et outils externes aux containers.

 bouclier jaune: Ce bouclier est situé sur le flux entre le user et le container docker de Métabase. Ce flux représente l'accès aux visualisations de Métabase pour l'utilisateur, et il est obligé de se créer un compte avec un identifiant et un mot de passe pour avoir accès aux bases de données que celui-ci a paramétrées. Cela rajoute donc une sécurité supplémentaire sur l'accès aux données.

Quant à l'anonymisation des données, comme nous l'avons vu précédemment, les données traitées ont été vérifiées et contrôlées afin de respecter l'ensemble des règles RGPD afin de garantir l'intégrité de celles-ci dans un respect des normes de la vie privée et de la sécurité des données.

Conclusion.

Durant la réalisation de ce projet, j'ai mis un effort supplémentaire sur la partie d'interconnexion entre différents outils cloud, dans l'objectif de créer une vraie architecture semblable à un réel projet d'entreprise.

J'ai beaucoup appris sur ce projet, que cela soit sur la recherche de données OpenData, sur la partie de conception des ETLs, mais aussi sur toute la partie de visualisation, dont j'ai fait peu de réalisations durant mon alternance.

Le traitement et analyse du jeu de données de 56 millions d'enregistrements des relevés de prix de stations essence a aussi été très pertinent.

L’aide de Large Language Model n'a pas été utilisée lors de l’écriture de ce mémoire, mais ils m’ont aidé lors de difficultés d’implémentation sur le projet. Par exemple, lors de l’import de MongoDB-tools sur le dockerfile. Cela m’a fait gagner beaucoup de temps sur certains points, et cela me permet de mieux me concentrer sur les problèmes d’architecture ou de problèmes métiers.

Je vous remercie du temps que vous avez passé à lire mon mémoire.