# Documentation Technique – Pipeline d’ingestion de données

## Comportement global du pipeline d’ingestion

Ce pipeline d’ingestion rassemble des données provenant du site Studi, en combinant deux sources : l’API JSON de Drupal (données structurées pour métiers, formations, etc.) et le **scraping web** (contenu détaillé des pages de formation). Elle se décompose en plusieurs étapes successives :

1. **Récupération des données brutes depuis l’API Drupal** – Le pipeline interroge l’API JSON de Studi (Drupal) pour extraire tous les éléments des différents types (métiers, formations, domaines, etc.). Les données complètes sont récupérées de manière récursive (toutes les pages de résultats) et sauvegardées telles quelles en JSON brut (dossier full/).
2. **Extraction et structuration des données Drupal** – Les données brutes sont ensuite traitées pour en extraire les champs utiles (identifiants, intitulés, descriptions…) et les relations entre entités. On produit des fichiers JSON structurés pour chaque type d’entité (par ex. jobs.json pour les métiers) à partir des données brutes, en enrichissant avec certains contenus texte liés.
3. **Scraping des pages de formation** – Parallèlement aux données de l’API, le pipeline récupère des contenus détaillés qui ne sont pas fournis via Drupal (ex : le programme détaillé des formations, méthodes pédagogiques, etc.) en naviguant sur les pages web des formations. Elle identifie toutes les URLs des formations puis télécharge chaque page. Le contenu HTML est ensuite parsé pour en extraire les différentes sections (description, programme, financement, métiers liés, etc.), qui sont sauvegardées en JSON.
4. **Génération des documents et métadonnées** – À partir des JSON structurés (API Drupal) et des sections de pages web, le pipeline génère des objets Document (de LangChain) pour chaque entité. Chaque document contient un contenu textuel formaté (par ex. le titre, ou un résumé, ou les sections de formation) et des métadonnées (type d’entité, identifiants, noms associés, etc.). On obtient ainsi une collection de documents unifiés prêts pour l’indexation ou l’utilisation par un modèle de langage.
5. ***(Étape optionnelle) Génération de synthèses et questions via LLM*** – Pour enrichir les données, le pipeline peut faire appel à un modèle de langage (LLM) afin de résumer le contenu de certaines entités (notamment les formations) et générer des questions-réponses associées. Concrètement, chaque document de formation peut être découpé en morceaux avec un résumé et des questions générées automatiquement sur son contenu, ce qui facilite les cas d’usage de Q/R ou de chatbot. Les résultats sont stockés pour éviter de refaire ces appels coûteux.

Chaque étape produit des fichiers de sortie (ou charge des fichiers existants) qui servent d’entrées à l’étape suivante. Ci-dessous, chaque étape est détaillée avec son fonctionnement interne, ses entrées/sorties, et la liste des fichiers concernés.

## Étape 1 : Récupération des données brutes depuis l’API Drupal

**Fonctionnement :** Cette première étape interroge l’API JSON de Drupal (https://www.studi.com/jsonapi/) pour récupérer **toutes les données brutes** des différents types de contenu nécessaires (métiers, formations, domaines, diplômes, etc.). La classe principale utilisée est DrupalDataRetrieval (définie dans drupal\_data\_retrieval.py), qui s’appuie sur un client API DrupalJsonApiClient pour exécuter les requêtes HTTP GET.

Pour chaque type d’entité à récupérer, la logique est la suivante :

* **Requête initiale** : on appelle l’endpoint JSONAPI approprié (par ex. node/jobs pour les métiers, node/training pour les formations, taxonomy\_term/domain pour les domaines, etc.). Le client utilise requests.get pour récupérer la première page de résultats (en JSON). Si l’API nécessite une authentification, les identifiants sont fournis lors de l’initialisation du client (DrupalJsonApiClient(base\_url, user\_name, user\_password)).
* **Récupération récursive** : si les résultats sont paginés, le client détecte un lien next dans la réponse JSON et appelle récursivement les pages suivantes via la méthode get\_drupal\_data\_recursively. Cela permet d’accumuler **tous** les objets du type demandé, dans une liste complète.
* **Sauvegarde brute** : la liste complète d’objets JSON (tels que renvoyés par Drupal, sans modifications) est sauvegardée dans un fichier JSON brut dans le dossier full/. Par exemple, tous les métiers bruts sont stockés dans full/jobs.json. Ces fichiers bruts servent de cache local : si on relance le pipeline plus tard, elle vérifiera l’existence de ces fichiers pour éviter de refaire l’appel réseau (on **charge le fichier existant** au lieu de requêter l’API).

Exemple : lors de la récupération des métiers, la méthode retrieve\_jobs() vérifie d’abord si outputs/full/jobs.json existe déjà. Si oui, le contenu est chargé depuis ce fichier (données mises en cache). Sinon, DrupalJsonApiClient.get\_drupal\_data\_recursively('node/jobs') est appelé pour récupérer tous les nœuds de type métier via l’API, puis le résultat est sauvegardé dans outputs/full/jobs.json. Ce fonctionnement est répété pour chaque catégorie de données (formations, financements, diplômes, etc.).

**Entrées :** aucune donnée d’entrée locale n’est requise au premier lancement (sauf les identifiants API éventuellement). Lors des exécutions suivantes, l’étape peut tirer profit des fichiers bruts déjà présents (cache).

**Sorties :** pour chaque type d’entité, un fichier JSON brut est produit (ou rechargé s’il existe). Ces fichiers contiennent l’intégralité des objets tels que renvoyés par Drupal (y compris leurs attributs et structures de relations complexes). Ils seront utilisés comme base pour l’étape 2 (extraction des champs utiles).

### Fichiers générés/chargés à l’étape 1

| **Nom du fichier** | **Chemin (dossier)** | **Statut** | **Rôle dans l’étape 1** |
| --- | --- | --- | --- |
| **jobs.json** (brut) | outputs/full/jobs.json | Généré si absent, sinon **chargé** | Données brutes de tous les métiers récupérés via l’API Drupal (node/jobs). Contient la liste complète des nœuds métiers. |
| **fundings.json** (brut) | outputs/full/fundings.json | Généré si absent, sinon **chargé** | Données brutes de tous les financements (node/funding). |
| **trainings.json** (brut) | outputs/full/trainings.json | Généré si absent, sinon **chargé** | Données brutes de toutes les formations (node/training). |
| **diplomas.json** (brut) | outputs/full/diplomas.json | Généré si absent, sinon **chargé** | Données brutes de tous les diplômes (node/diploma). |
| **certifications.json** (brut) | outputs/full/certifications.json | Généré si absent, sinon **chargé** | Données brutes de tous les éléments de taxonomie “certification” (taxonomy\_term/certification). |
| **domains.json** (brut) | outputs/full/domains.json | Généré si absent, sinon **chargé** | Données brutes de tous les domaines (taxonomy\_term/domain). (Inclut également les sous-domaines imbriqués dans la structure.) |
| **certifiers.json** (brut) | outputs/full/certifiers.json | Généré si absent, sinon **chargé** | Données brutes de tous les certifieurs (taxonomy\_term/certifier). |

**Note :** Le dossier de sortie de ces fichiers peut être configuré via le paramètre outdir passé à DrupalDataRetrieval. Dans cet exemple, on suppose outdir="outputs/". Les fichiers sont nommés d’après le type de données, préfixés par le sous-dossier full/ pour indiquer qu’il s’agit de données brutes complètes.

## Étape 2 : Extraction et structuration des données Drupal

**Fonctionnement :** Une fois les données brutes disponibles (en mémoire ou via les fichiers de l’étape 1), le pipeline extrait les champs pertinents et structure les données de manière simplifiée pour chaque entité. Cette étape utilise encore DrupalDataRetrieval mais se concentre sur le **post-traitement** des objets JSON bruts en vue de produire des JSON plus légers, directement exploitables par la suite.

Les principales opérations pour chaque objet :

* **Attributs principaux** : on extrait les champs textuels importants (titre, nom, description…) depuis la section attributes de chaque objet Drupal. Ces valeurs sont insérées dans une structure Python plus simple (target\_node). La méthode utilitaire extract\_common\_data\_from\_nodes (définie dans DrupalJsonApiClient) automatise cela : pour chaque élément, elle crée un dictionnaire contenant l’id, le type de l’objet, puis ajoute si présents les champs title, name, description et la date de modification (changed).
* **Liens “related”** : si l’objet JSON original possède un lien links.related (fourni par Drupal JSON:API pour accéder à l’URL publique de l’entité), ce lien est conservé. Il est stocké dans le champ related\_url de l’objet simplifié, afin de garder une trace du lien web associé.
* **Identifiants de relations** : pour certaines propriétés relationnelles (par exemple le domaine rattaché à un métier, ou les métiers liés à un domaine), on enregistre seulement les identifiants des relations plutôt que tout l’objet. La fonction extract\_common\_data\_from\_nodes gère cela via le paramètre included\_relationships\_ids. Par exemple, pour un métier on stocke l’ID du domaine parent dans related\_ids.domain. Ces identifiants serviront plus tard pour recouper les entités (ex: retrouver le nom du domaine depuis son ID).
* **Relations à récupérer** : certains champs contiennent de longs textes structurés qui sont fournis par des entités liées (notamment les **Paragraphes** Drupal). C’est le cas par exemple des descriptions détaillées pouvant apparaître sur les fiches métiers ou financements. Pour obtenir le contenu de ces paragraphes, le pipeline utilise la méthode parallel\_get\_items\_related\_infos du client API. Cette méthode lance en parallèle des requêtes HTTP pour chaque URL listée dans related\_url (par ex. field\_paragraph) et extrait récursivement tous les textes présents. Le résultat (texte du paragraphe) est rassemblé dans related\_infos.paragraph au sein de chaque objet. Ces appels sont parallélisés (pool de threads) pour accélérer l’extraction de tous les textes liés.
* **Nettoyage** : les textes extraits sont nettoyés des caractères spéciaux ou encodages particuliers (via txt.fix\_special\_chars et UnicodeHelper). On s’assure aussi de supprimer d’éventuelles redondances dans les listes de phrases extraites (par exemple, éviter des doublons très similaires en utilisant un filtre basé sur BM25).

Après ce traitement, on obtient pour chaque type d’entité une **liste d’objets simplifiés** (avec champs primaires et éventuellement du texte additionnel). Ces listes sont ensuite sauvegardées en JSON dans des fichiers séparés (sans le préfixe full/).

**Entrées :** les fichiers JSON bruts de l’étape 1 (ou les structures en mémoire) pour chaque type. Si ces fichiers existent, l’étape 2 les charge; sinon, elle utilise directement les données retournées par l’API. Par exemple, retrieve\_jobs() chargera full/jobs.json puis traitera ces données.

**Sorties :** un fichier JSON structuré par type d’entité, contenant une liste d’entrées épurées. Ces fichiers sont beaucoup plus légers que les bruts et contiennent uniquement les champs nécessaires (y compris des relations simplifiées). Ils serviront d’**entrée à l’étape 4** (génération des documents LangChain).

### Fichiers générés/chargés à l’étape 2

| **Nom du fichier** | **Chemin (dossier)** | **Statut** | **Rôle dans l’étape 2 (données structurées)** |
| --- | --- | --- | --- |
| **jobs.json** | outputs/jobs.json | **Généré** à partir de *full/jobs.json* | Données structurées de tous les métiers : pour chaque métier, inclut id, type, titre du métier, description, date modif, et l’ID du domaine lié (related\_ids.domain). (Contenu texte additionnel : si le métier possède des paragraphes descriptifs, ils sont récupérés et stockés dans *related\_infos.paragraph*.) |
| **fundings.json** | outputs/fundings.json | **Généré** | Données structurées de tous les financements : id, type, titre du financement, description éventuelle, etc. Si des paragraphes sont liés aux financements, leur contenu est collecté dans related\_infos.paragraph. |
| **trainings.json** | outputs/trainings.json | **Généré** | Données structurées de toutes les formations : id, type, titre de la formation, champs de description courts. Ce JSON contient également les identifiants de relations vers d’autres entités (ex : related\_ids.job, related\_ids.domain, related\_ids.diploma, etc. pour lier vers les métiers, domaines, diplômes correspondants). **Remarque :** Les contenus détaillés des formations (programme, etc.) ne sont pas inclus via l’API Drupal, ils seront récupérés dans l’étape 3 par scraping. |
| **diplomas.json** | outputs/diplomas.json | **Généré** | Données structurées de tous les diplômes : id, type, titre du diplôme, etc. Pour chaque diplôme, on extrait aussi le texte descriptif associé. Via related\_infos.paragraph, on stocke les contenus de paragraphe liés (par ex. explication du diplôme). |
| **certifications.json** | outputs/certifications.json | **Généré** | Données structurées de toutes les entrées de taxonomie "certification" : id, type, nom de la certification. (Les certifications étant des termes de taxonomie assez simples, on ne récupère ici que le nom et l’id.) |
| **domains.json** | outputs/domains.json | **Généré** | Données structurées de tous les domaines : id, type, nom du domaine, etc. Un domaine peut contenir une liste de noms de sous-domaines associés (subdomains\_names). L’étape 2 sépare les domaines racine des sous-domaines enfants en se basant sur la relation parent fournie par Drupal. Seuls les domaines racine sont listés dans ce fichier, chacun avec la liste de ses sous-domaines. |
| **subdomains.json** | outputs/subdomains.json | **Généré** | Données structurées de tous les sous-domaines : id, type, nom du sous-domaine, + références à leur domaine parent. Chaque entrée comprend domain\_name (le nom du domaine parent) et domain\_id (l’ID du domaine parent), facilitant les jointures ultérieures. |
| **certifiers.json** | outputs/certifiers.json | **Généré** | Données structurées de tous les certifieurs : id, type, nom du certifieur. (Ce sont les organismes certificateurs, extraits de la taxonomie certifier.) |

**Note :** Les fichiers de cette étape sont systématiquement générés à partir des données brutes. S’ils existaient déjà d’une exécution précédente, ils seront écrasés (mise à jour) lors d’une nouvelle récupération. Ces fichiers JSON épurés seront utilisés tels quels dans l’étape 4 pour construire les documents finaux.

## Étape 3 : Scraping des contenus détaillés des formations

**Fonctionnement :** Cette étape complète les données issues de l’API en récupérant le **contenu HTML des pages web** de chaque formation sur studi.com, puis en en extrayant les sections utiles. Le script website\_scraping\_retrieval.py définit la classe WebsiteScrapingRetrieval qui gère ce processus. Il procède en deux sous-parties : d’abord collecter toutes les URLs des pages de formation, puis télécharger et parser chaque page.

* **Collecte des URLs de formation** : la méthode scrape\_all\_trainings\_links(max\_pagination) parcourt le site d’annuaire des formations. Le site Studi liste les formations page par page (paramètre ?training[page]=N). Le script utilise Selenium (navigateur headless Chrome) pour charger chaque page et la bibliothèque BeautifulSoup pour extraire les liens des formations. Le paramètre max\_pagination (par défaut 17) indique le nombre de pages à parcourir, afin de couvrir l’ensemble des formations. Les URLs collectées (de la forme https://www.studi.com/fr/formations/nom-de-la-formation) sont enregistrées dans un fichier JSON. Si ce fichier all\_trainings\_links.json existe déjà, le script le recharge directement au lieu de refaire le parcours (gain de temps).
* **Scraping des pages de détail** : une fois la liste des liens obtenue, la méthode parallel\_scrape\_content\_missing\_webpages(urls) va visiter **chaque URL de formation** pour en télécharger le contenu HTML complet. Afin d’accélérer, les pages sont traitées par lots en parallèle (via un ThreadPool). Par défaut, seules les pages manquantes sont effectivement téléchargées (only\_missing\_webpages\_to\_scrape=True signifie qu’on vérifie pour chaque formation si un fichier de contenu existe déjà; si oui, on ne la retélécharge pas). Chaque page HTML récupérée est sauvegardée telle quelle (avec son contenu intégral) dans un fichier JSON individuel dans le dossier scraped-full/. Par exemple, pour une formation dont l’URL se termine par /bac+3-chef-de-projet, le fichier sera scraped-full/bac+3-chef-de-projet.json contenant : { "name": "<slug>", "url": "<url>", "content": "<html...>" }.
* **Extraction des sections** : après avoir toutes les pages HTML (ou au fur et à mesure), le script analyse le HTML pour en extraire des sections de contenu structurées. La fonction save\_sections\_from\_scraped\_pages() charge chaque fichier de scraped-full/, parse le HTML avec BeautifulSoup, et isole les blocs pertinents par leurs identifiants/classes CSS. Par exemple, il recherche les sections <section class="lame-bref"> (description brève), "lame-programme" (programme de formation), "lame-financement" (options de financement), "lame-modalites", etc., ainsi que la section identifiée par id="jobs" (débouchés métiers). Chaque section trouvée est nettoyée : le script reconstruit le texte, gère les sauts de ligne (<br> remplacés par \n), déplie les éléments d’accordéon (les sections de programme découpées en sous-parties), et formate les listes à puces hiérarchiques. Le résultat est un dictionnaire Python où chaque clé est le nom de la section (ex: bref, programme, financement, metiers…) et la valeur est le contenu texte concaténé (souvent sous forme de liste à puces ou paragraphe). On ajoute également le titre de la formation (title) et éventuellement le niveau académique (academic\_level) si présent. Ce dictionnaire de sections est ensuite sauvegardé en JSON dans le dossier scraped/. Chaque fichier porte le même nom que le slug de la formation. Par exemple scraped/bac+3-chef-de-projet.json contiendra : { "title": "...", "academic\_level": "...", "bref": "...", "programme": "...", "financement": "...", "metiers": "...", "url": "..." }.

**Entrées :** aucune entrée locale directe n’est requise initialement. En cours de route, l’étape utilise les fichiers produits par elle-même :

* outputs/all\_trainings\_links.json si déjà présent (pour éviter de reparcourir les pages index de formations).
* Les fichiers HTML déjà scrapés dans scraped-full/ si présents (pour ne scraper que les nouveautés ou compléter les manquants).

**Sorties :**

* La liste complète des URLs de formation dans all\_trainings\_links.json (à conserver pour usage ultérieur).
* Pour chaque formation, un fichier JSON brut de page HTML dans scraped-full/ (utilisé comme archive du contenu complet de la page).
* Pour chaque formation, un fichier JSON de sections de contenu dans scraped/, qui sera réutilisé lors de la génération des documents (étape 4).

### Fichiers générés/chargés à l’étape 3

| **Nom du fichier** | **Chemin (dossier)** | **Statut** | **Rôle dans l’étape 3 (scraping web)** |
| --- | --- | --- | --- |
| **all\_trainings\_links.json** | outputs/all\_trainings\_links.json | Généré si absent, sinon **chargé** | Liste de toutes les URLs des pages de formation. Sert de point de départ pour scraper chaque page. Le fichier est conservé pour éviter d’avoir à re-parcourir le site à chaque exécution. |
| **<slug>.json** (page HTML brute) | outputs/scraped-full/*<slug>*.json | Généré (un par formation) | Contenu HTML complet de la page de formation correspondant à <slug> (identifiant unique dans l’URL). Ces fichiers sont générés en masse via Selenium pour analyse ultérieure. Si un fichier existe déjà pour une formation, on ne la télécharge pas de nouveau (sauf demande explicite). |
| **<slug>.json** (sections extraites) | outputs/scraped/*<slug>*.json | **Généré** (un par formation) | Contenu texte structuré extrait de la page de formation <slug>. Comprend le titre, niveau académique, et différentes sections (description brève, programme, méthodes, modalités, financements, métiers liés, etc.). Ce fichier sert d’**entrée** pour la création des documents de formation à l’étape 4. Chaque exécution du scraping recrée ce fichier à partir de la version brute correspondante. |

**Exemple :** Pour la formation « Chef de projet digital (Bac+3) », le slug pourrait être chef-de-projet-digital-bac3. Le scraping produira outputs/scraped-full/chef-de-projet-digital-bac3.json (HTML) puis outputs/scraped/chef-de-projet-digital-bac3.json (sections texte contenant, par exemple, "bref": "Cette formation en quelques mots...", "programme": "• Module 1: ..." etc.). Ces informations textuelles seront fusionnées avec les données de l’API lors de l’étape suivante.

## Étape 4 : Génération des documents et métadonnées unifiés

**Fonctionnement :** À cette étape, on fusionne les données issues de l’API Drupal (étape 2) et les contenus détaillés des formations (étape 3) pour produire des **documents unifiés**, c’est-à-dire des instances de langchain.schema.Document. Ces objets contiennent un champ page\_content (texte) et un champ metadata (dictionnaire de métadonnées). L’objectif est d’avoir un document par élément de connaissance (métier, formation, etc.), prêt à être indexé dans un moteur de recherche sémantique ou une base de connaissances.

Le script generate\_documents\_and\_metadata.py fournit la classe GenerateDocumentsAndMetadata qui gère ce processus. La méthode principale est load\_all\_docs\_as\_json(path) -> List[Document]. Elle va successivement traiter chaque type de données :

* **Chargement des JSON structurés** : pour chaque catégorie (certifieurs, certifications, diplômes, domaines, sous-domaines, financements, métiers, formations), on charge le fichier JSON correspondant (créé en étape 2) depuis le chemin indiqué (path). Par exemple, certifiers.json, domains.json, etc.
* **Conversion en Document – cas simples** : pour les entités simples (certifieurs, certifications, domaines, sous-domaines), on crée un Document par entrée en utilisant principalement le nom/titre comme contenu. Exemple : un certifieur nommé “ABC organisme” donnera un Document(page\_content="ABC organisme", metadata={ "doc\_id": ..., "type": "certifieur", "name": "ABC organisme", ... }). On collecte également tous les noms dans une liste (pour éventuellement générer des listes globales). Les fonctions process\_certifiers, process\_certifications, process\_domains, process\_sub\_domains font ce travail : elles itèrent sur les entrées JSON, créent des objets Document et compilent la liste des noms uniques.
* **Documents diplômes et financements (avec texte)** : pour les diplômes et les financements, il existe du texte descriptif récupéré via related\_infos.paragraph à l’étape 2. Les fonctions process\_diplomas et process\_fundings intègrent ces informations : le contenu du Document est construit en concaténant le titre et le texte du paragraphe. Par exemple, pour un diplôme, le page\_content sera "<Titre du diplôme>\n<Paragraphes descriptifs...>". Les métadonnées incluent toujours l’id, le type ("diplôme" ou "financement"), le nom et la date de modification.
* **Documents métiers** : pour chaque métier, on génère un texte court du type **« Métier : '<nom\_du\_métier>'. Domaine : <nom\_du\_domaine>. »**. L’objectif est de fournir un petit contexte liant le métier et son domaine. Le nom du domaine est obtenu en cherchant dans la liste des domaines (chargée plus haut) via l’ID stocké en related\_ids.domain. Les métadonnées du métier incluent l’id, "type": "métier", le nom du métier et éventuellement les identifiants reliés (rel\_ids) regroupant toutes les relations (ici juste le domaine).
* **Documents formations** : le cas des formations est plus riche, combinant les données de l’API et du scraping. Pour chaque formation (chaque entrée de trainings.json), la fonction process\_trainings récupère d’une part les métadonnées de base (id, type "formation", nom, date, et tous les identifiants reliés comme domaine, sous-domaine, certification, diplôme, métier, financement, objectifs…). Ensuite, elle intègre les **détails de formation** obtenus par scraping : on utilise la correspondance par titre de formation pour retrouver le dictionnaire de sections (chargé depuis outputs/scraped/<slug>.json). On enrichit les métadonnées de la formation avec l’URL de la page (url) et le niveau académique (academic\_level) si disponibles. À partir de là, la génération se fait en deux temps pour chaque formation :
  + **Document “summary”** : un document de résumé basé uniquement sur les données Drupal. On prend tous les champs textuels disponibles dans training\_data['attributes'] (exemple de champs possibles : description courte, objectifs, prérequis… fournis via l’API) et on les concatène sous forme de sections Markdown (la fonction utilitaire get\_french\_section donne un titre en français à chaque champ, par ex. "Objectifs", "Pré-requis"). On précède ces sections par le titre de la formation et un lien vers la page. Ce Document porte une métadonnée supplémentaire "training\_info\_type": "summary" pour indiquer qu’il s’agit d’un résumé issu de l’API.
  + **Documents par section détaillée** : pour chaque section provenant du scraping (par exemple bref, programme, financement, metiers…), on crée un Document séparé contenant le texte de cette section. Le contenu commence par un en-tête indiquant qu’il s’agit de cette section de la formation donnée (ex: "## Programme de la formation : <Nom> ##\n<contenu du programme>"). La métadonnée "training\_info\_type" est définie au nom de la section (ex: "programme", "financement", etc.). Ainsi, plusieurs Documents peuvent correspondre à une même formation, distingués par ce type d’information. On utilise metadata\_common (métadonnées de base de la formation) et on y ajoute le champ training\_info\_type spécifique avant d’instancier chaque Document.
  + Remarque : Si une formation n’a pas de détails de scraping (cas improbable si toutes les formations du JSON ont une page web correspondante), le script le signale par un warning et ne créera que le document “summary”. Dans la plupart des cas, chaque formation génère donc **un document résumé + plusieurs documents de sections**.
* **Listes globales de noms (facultatif)** : à la fin, si l’option write\_all\_lists=True, la méthode écrit dans un sous-dossier all/ un fichier JSON par type contenant la liste de tous les noms d’entités rencontrés (ex : all\_jobs\_names.json, all\_trainings\_names.json, etc.). Ces listes peuvent être utiles pour alimenter des menus déroulants, de l’autocomplétion ou simplement pour vérifier le contenu. Elles ne sont pas directement utilisées dans les étapes suivantes de le pipeline.

En fin d’étape 4, on dispose en mémoire d’une liste globale de Documents (all\_docs). Ceux-ci représentent l’intégralité de la base de connaissances fusionnée (chaque métier, chaque formation avec ses infos, etc.), prêts à être indexés. Si besoin, on peut sérialiser ces Documents ou les injecter dans un vecteur de recherche.

**Entrées :**

* Les fichiers JSON structurés issus de l’étape 2 (\*.json dans outputs/, ou autre dossier passé en paramètre path). Ils sont **tous chargés** en début de processus.
* Les fichiers de sections de formation issus de l’étape 3 (outputs/scraped/\*.json). La méthode load\_trainings\_details\_as\_json(path) charge tous les JSON de scraped/ et crée un dictionnaire { titre\_de\_formation -> sections }. Ce dictionnaire est utilisé par la génération des documents de formation.

**Sorties :**

* Principalement, une liste de Document en mémoire (non directement écrite sur disque dans ce script). Cette liste combine des documents de types variés (métier, formation, etc.).
* Fichiers **liste de noms** dans outputs/all/ (ou <path>/all/) pour chaque catégorie, si activé. Ces fichiers JSON contiennent par exemple ["Nom Métier1", "Nom Métier2", ...].

### Fichiers chargés/générés à l’étape 4

| **Nom du fichier** | **Chemin (dossier)** | **Chargé/Généré** | **Utilisation dans l’étape 4** |
| --- | --- | --- | --- |
| **certifiers.json** | outputs/certifiers.json | **Chargé** | Données structurées des certifieurs (étape 2). Sert à créer un Document par certifieur. |
| **certifications.json** | outputs/certifications.json | **Chargé** | Données structurées des certifications. Sert à créer un Document par certification. |
| **diplomas.json** | outputs/diplomas.json | **Chargé** | Données structurées des diplômes. Utilisé pour créer les Documents diplômes (avec leur texte descriptif). |
| **domains.json** | outputs/domains.json | **Chargé** | Données structurées des domaines. Sert à la création des Documents domaines et pour associer les métiers à leur nom de domaine. |
| **subdomains.json** | outputs/subdomains.json | **Chargé** | Données structurées des sous-domaines. Utilisé pour créer les Documents sous-domaine et aider à associer formations <-> sous-domaines/domaines. |
| **fundings.json** | outputs/fundings.json | **Chargé** | Données structurées des financements. Sert à créer les Documents financements (texte descriptif inclus). |
| **jobs.json** | outputs/jobs.json | **Chargé** | Données structurées des métiers. Sert à créer les Documents métiers (avec mention du domaine). |
| **trainings.json** | outputs/trainings.json | **Chargé** | Données structurées des formations (champs basiques et IDs relations). Sert de base pour chaque Document formation (métadonnées de base, résumé API). |
| **<slug>.json** (sections formation) | outputs/scraped/*<slug>*.json | **Chargé** (tous) | Détails de chaque formation obtenus par scraping (étape 3). L’étape 4 charge tous ces fichiers et les indexe par titre de formation. Ces données alimentent le contenu des Documents formation (sections détaillées). |
| **all\_certifiers\_names.json** | outputs/all/all\_certifiers\_names.json | **Généré** | Liste de tous les noms de certifieurs présents (écrite si write\_all\_lists=True). |
| **all\_certifications\_names.json** | outputs/all/all\_certifications\_names.json | **Généré** | Liste de tous les noms de certifications. |
| **all\_diplomas\_names.json** | outputs/all/all\_diplomas\_names.json | **Généré** | Liste de tous les titres de diplômes. |
| **all\_domains\_names.json** | outputs/all/all\_domains\_names.json | **Généré** | Liste de tous les noms de domaines. |
| **all\_subdomains\_names.json** | outputs/all/all\_subdomains\_names.json | **Généré** | Liste de tous les noms de sous-domaines. |
| **all\_fundings\_names.json** | outputs/all/all\_fundings\_names.json | **Généré** | Liste de tous les titres de financements. |
| **all\_jobs\_names.json** | outputs/all/all\_jobs\_names.json | **Généré** | Liste de tous les intitulés de métiers. |
| **all\_trainings\_names.json** | outputs/all/all\_trainings\_names.json | **Généré** | Liste de tous les titres de formations. |

**Exemple de Document généré :** Un métier “Développeur Web” (id 123) rattaché au domaine “Informatique” produira un Document avec page\_content="Métier : 'Développeur Web'. Domaine : Informatique." et metadata={"doc\_id": "123", "type": "métier", "name": "Développeur Web", "changed": "...", "rel\_ids": "..."} .  
Pour une formation “Chef de projet digital (Bac+3)”, on obtiendra plusieurs Documents : un résumé (contenant les informations de l’API comme objectifs, prérequis, etc.) avec training\_info\_type: "summary", et des Documents “bref”, “programme”, “financement”, etc., contenant chacun la section correspondante du contenu de la page web (issus du JSON de scraped).

## Étape 5 (optionnelle) : Génération de synthèses et questions via LLM

**Fonctionnement :** Cette étape n’est pas toujours activée, mais elle permet d’enrichir les documents (surtout les formations) en utilisant un **Large Language Model (LLM)** pour générer des résumés plus synthétiques et des questions sur le contenu. Le fichier summary\_chunks\_with\_questions\_documents.py fournit la classe de service SummaryWithQuestionsByChunkDocumentsService pour orchestrer ces opérations. L’idée est de transformer chaque document de formation en une version condensée et interactive : on crée un résumé concis, puis on découpe le contenu en “chunks” accompagnés de questions pertinentes sur chaque chunk.

Les principales étapes internes (asynchrones) :

1. **Construction des documents d’entrée** : on peut partir des Document créés à l’étape 4, en particulier ceux liés aux formations (puisque ce sont eux qui contiennent beaucoup de texte). Le service propose aussi des fonctions build\_all\_but\_trainings\_documents et build\_trainings\_docs qui rechargent directement les JSON de l’étape 2 et 3 pour recréer les Documents. (En pratique, l’étape 5 peut donc reconstruire ses propres Documents de formation à partir des fichiers, ce qui évite d’avoir à passer la liste de l’étape 4 en paramètre.) On obtient une liste de Documents “raw” pour les formations, potentiellement avec toutes leurs sections dans le contenu.
2. **Appels LLM – résumé et chunk** : Pour chaque Document de formation, on utilise des prompts prédéfinis (fichiers ressource .french.txt) pour demander au LLM de générer : (a) un **résumé global** du document, et (b) un découpage en **chunks** (morceaux thématiques ou paragraphes) avec une série de questions pour chaque chunk. Il existe plusieurs modes implémentés : un mode en une seule étape (un seul prompt pour tout faire) ou en deux/trois étapes (d’abord un résumé, puis un chunking, puis des questions). Le service utilise typiquement la méthode en trois étapes pour fiabilité :
   * Étape LLM 1 : Générer un résumé du document de formation. (Prompt du type "Voici le contenu d’une formation, produis-en un résumé concis en français…") – Le résultat est un texte court qui synthétise la formation.
   * Étape LLM 2 : À partir du résumé obtenu, demander au LLM de découper le contenu complet en segments cohérents (chunks) et de donner pour chacun un titre ou un sous-résumé. (Prompt du type "Découpe le document suivant en sections thématiques et fournis chaque section", etc.) – Le LLM renvoie une liste de chunks (textes).
   * Étape LLM 3 : Pour chaque chunk, on sollicite le LLM afin de générer une ou plusieurs **questions pertinentes** portant sur ce segment. (Prompt du type "En te basant sur le contenu du segment ci-dessous, génère 2 questions quiz auxquelles ce segment répond.") – Le résultat est une liste de questions par chunk (éventuellement sous forme JSON suivant un modèle Pydantic attendu).  
     Ces appels sont effectués de manière asynchrone et parallélisée par lot pour accélérer le traitement de l’ensemble des formations. Le code utilise des méthodes utilitaires (Llm.invoke\_parallel\_prompts\_with\_parser\_batches\_fallbacks\_async) pour gérer les appels aux LLM (principal + modèle de repli au besoin) et parser directement les réponses JSON selon des modèles définis (DocWithSummaryChunksAndQuestionsPydantic etc.).
3. **Assemblage des résultats** : Pour chaque formation, on construit un objet de type DocWithSummaryChunksAndQuestions contenant le contenu d’origine, le résumé, et la liste des chunks avec leurs questions. Ensuite, ces objets peuvent être convertis de nouveau en Documents LangChain. Par exemple, on peut créer pour chaque chunk un Document dont le page\_content contient le texte du chunk suivi des questions et réponses (ou juste questions) associées. L’option merge\_questions\_with\_data permet de choisir si on fusionne les questions avec le contenu ou si on les garde séparés. Par défaut, la méthode to\_langchain\_documents\_chunked\_summary\_and\_questions(True, True) va créer des documents qui combinent chaque chunk et ses questions, formattés de manière lisible (par ex. en ajoutant une section "### Questions ###" suivie d’une section "### Réponses ###" dans le texte). Ces nouveaux Documents représentent donc une version **augmentée** de la formation : chaque Document résultant couvre un segment de la formation et intègre des Q/R pour faciliter l’entraînement d’un agent conversationnel ou d’un quiz.
4. **Sauvegarde** : les objets complets (résumé + chunks + questions) générés pour chaque formation sont **sérialisés en JSON** dans un fichier unique (par exemple trainings\_summaries\_chunks\_and\_questions\_objects.json). Ce fichier sert de cache pour cette étape coûteuse : si on relance le pipeline, on peut recharger ces résultats au lieu de solliciter de nouveau le LLM.

**Entrées :**

* Les Documents de formation générés à l’étape 4 (pouvant être reconstruits en interne). Les autres documents (métiers, etc.) ne sont généralement pas traités par le LLM (pas besoin de résumés pour de simples noms), bien que le service les charge éventuellement pour constituer un ensemble complet.
* Le modèle de langage configuré (llm\_and\_fallback) et les prompts de résumé/questions pré-définis (fichiers texte). Ces derniers ne sont pas des fichiers d’entrée de données mais plutôt des ressources fixes pour les requêtes.

**Sorties :**

* Un fichier JSON regroupant les résultats LLM pour les formations, ex: **trainings\_summaries\_chunks\_and\_questions\_objects.json**. Il contient pour chaque formation un objet JSON avec son résumé et ses chunks + questions. Ce fichier peut être rechargé directement pour recréer les documents augmentés sans refaire les appels LLM.
* Une liste de nouveaux Documents (en mémoire) correspondant aux chunks de formations avec questions. Si l’on souhaite les exploiter, on peut les ajouter à la liste globale de documents. Par exemple, la fonction generate\_summaries\_and\_questions\_for\_documents\_async retourne all\_documents qui inclut tous les documents initiaux inchangés **+** les documents chunkés/quiz des formations. Ces documents peuvent ensuite être indexés ou utilisés dans des cas d’usage de Q/R automatique.

### Fichiers chargés/générés à l’étape 5

| **Nom du fichier** | **Chemin (dossier)** | **Chargé/Généré** | **Utilisation dans l’étape 5 (LLM)** |
| --- | --- | --- | --- |
| **trainings\_summaries\_chunks\_and\_questions\_objects.json** | outputs/trainings\_summaries\_chunks\_and\_questions\_objects.json | Chargé si existant | Cache des résultats LLM pour les formations. Si ce fichier existe, l’étape 5 le charge pour reconstruire les objets de résumé/questions sans faire de nouveaux appels au modèle. S’il n’existe pas, il sera généré après les appels LLM et sauvegardé pour réutilisation ultérieure. |
| (Divers fichiers de prompt LLM) | common\_tools/helpers/ressource\_helper (resources) | Chargés (lecture) | Prompts pré-définis utilisés pour les requêtes LLM (ex: document\_summarize.french.txt, document\_create\_chunks\_and\_corresponding\_questions.french.txt, etc.). Ces fichiers contiennent les gabarits de question en français avec des variables, mais ne font pas partie des données métier de le pipeline. |
| **Documents augmentés** (chunks + Q) | (non sauvegardés en tant que fichiers individuels par défaut) | Générés en mémoire | Documents LangChain résultant de la transformation LLM. Chaque formation donne typiquement plusieurs documents (un par chunk) dont le contenu combine segment de cours et Q/R associées. (On peut les fusionner avec la liste globale ou les sauvegarder si besoin spécifique.) |

**Exemple :** Pour la formation “Chef de projet digital (Bac+3)”, le LLM pourrait générer un résumé global de quelques lignes, puis découper le contenu en 3 chunks (par ex. Présentation générale, Programme, Débouchés). Pour chaque chunk, il produit 2-3 questions. Un chunk “Programme” pourrait donner un Document avec un contenu du style : "### Programme de la formation : Chef de projet digital ###\n...<contenu du chunk>...\n\n### Questions ###\n- Quelles sont les principales compétences développées dans cette formation ?\n- Combien de projets sont à réaliser en fin de cursus ?\n\n### Réponses ###\nLes principales compétences incluent la gestion de projet agile, ...\nIl faut réaliser 2 projets majeurs en fin de cursus.". Ce document enrichi pourra servir à un chatbot pour répondre aux questions ou tester les connaissances.

## Gestion des fichiers selon les cas d’usage

En fonction des besoins, on peut choisir de conserver certains fichiers générés pour accélérer des exécutions futures, ou au contraire de les supprimer pour forcer une mise à jour complète. Voici, pour chaque étape, quels fichiers sont à **garder** ou à **supprimer** dans deux scénarios courants :

* **Réutiliser les données existantes (ne pas relancer les étapes coûteuses)** : on conserve tous les fichiers de sortie de l’étape précédente afin que le code les recharge au lieu de tout recalculer.
* **Refaire complètement une étape** : on supprime les fichiers cache correspondants, puis on relance la classe/fonction de l’étape pour regénérer les données à neuf.

### Étape 1 : Récupération des données brutes Drupal

* **Réutiliser les données existantes** : Conserver tous les fichiers outputs/full/\*.json (jobs, trainings, etc.). Lors d’une nouvelle exécution, DrupalDataRetrieval.retrieve\_all\_data() détectera ces fichiers et les chargera au lieu d’appeler l’API. Cela évite des requêtes réseau inutiles.
* **Refaire complètement l’étape** : Supprimer (ou renommer) les fichiers dans outputs/full/ avant d’exécuter la récupération. Ainsi, le client refera les appels API et téléchargera les données fraîches. Pour lancer l’étape manuellement, on peut utiliser la classe DrupalDataRetrieval : par exemple, exécuter DrupalDataRetrieval(outdir="outputs/").retrieve\_all\_data() pour tout récupérer, ou retrieve\_jobs() etc. pour une catégorie spécifique.

### Étape 2 : Extraction/structuration des données Drupal

(Remarque : cette étape est automatiquement couplée à l’étape 1 dans l’implémentation.)

* **Réutiliser** : Si on a conservé les fichiers JSON bruts (full/\*.json), la ré-exécution de l’extraction se fera rapidement en mémoire. Les fichiers structurés (outputs/\*.json) seront recréés à chaque fois, mais ce traitement est léger. On peut aussi conserver les fichiers structurés eux-mêmes si on souhaite éviter même ce recalcul, bien qu’il soit rapide. En pratique, garder les bruts suffit, car l’extraction sera refaite automatiquement.
* **Refaire** : Pour forcer une restructuration (par exemple si le code d’extraction a changé) sans refetcher l’API, on peut simplement supprimer les fichiers structurés (outputs/\*.json comme jobs.json, etc.) tout en gardant les bruts. Ensuite, exécuter à nouveau DrupalDataRetrieval.retrieve\_all\_data(): il va charger les bruts existants et **regénérer** les fichiers structurés mis à jour. Si l’on souhaite tout reprendre de zéro, supprimer également les bruts (ce qui équivaut à l’étape 1 refaite) puis lancer la récupération comme indiqué ci-dessus.

### Étape 3 : Scraping des pages de formation

* **Réutiliser les données existantes** : Conserver outputs/all\_trainings\_links.json ainsi que tout le dossier outputs/scraped-full/ et outputs/scraped/. Ainsi, en ré-exécutant WebsiteScrapingRetrieval.scrape\_all\_trainings(), le script va : charger les liens existants (sans reparcourir les pages index), puis pour chaque formation détecter que le fichier brut existe déjà et **ne pas re-télécharger** la page (grâce à only\_missing\_webpages\_to\_scrape=True). Il reconstruira quand même les fichiers scraped/\*.json (sections) à partir des bruts, assurant que les sections sont à jour. Ce processus est très rapide comparé au scraping initial.
* **Refaire complètement l’étape** : Pour forcer un nouveau scraping complet, on peut supprimer all\_trainings\_links.json (pour reprovoquer la collecte des liens) et/ou vider les dossiers scraped-full/ et scraped/. En particulier, la suppression des fichiers dans scraped-full/ obligera le script à re-télécharger chaque page HTML. Ensuite, on exécute WebsiteScrapingRetrieval().scrape\_all\_trainings(). Cela relancera la collecte des liens (si vide) puis le téléchargement de chaque page (car elles seront manquantes) et l’extraction des sections. Note : Si on ne souhaite pas supprimer les fichiers mais tout de même re-scraper (par exemple pour actualiser du contenu), on peut appeler directement parallel\_scrape\_content\_missing\_webpages(..., only\_missing\_webpages\_to\_scrape=False) en modifiant ce paramètre dans le code pour ignorer le cache et re-télécharger toutes les pages.

### Étape 4 : Génération des documents et métadonnées

* **Réutiliser les données existantes** : Cette étape est peu coûteuse, il n’est donc généralement pas nécessaire de la skipper. Cependant, si on a déjà construit les Documents et éventuellement indexé une base vectorielle, on peut éviter de la relancer tant que les fichiers JSON source (étape 2 et 3) n’ont pas changé. Il suffit de garder les fichiers outputs/\*.json (données structurées) et outputs/scraped/\*.json (sections formations). La fonction GenerateDocumentsAndMetadata.load\_all\_docs\_as\_json(path) peut être appelée à tout moment pour recréer la liste d’objets Document en mémoire. Si on dispose déjà d’une sortie persistante (par ex. une base d’index), on peut ne pas rappeler cette fonction. Les fichiers outputs/all/\*.json (listes de noms) ne sont utiles que si on en a un usage spécifique, sinon leur réécriture est sans conséquence.
* **Refaire l’étape** : Si les JSON source ont été mis à jour (suite à une refetch de l’API ou un nouveau scraping), il est conseillé de regénérer les Documents pour refléter les changements. On peut simplement rappeler la fonction load\_all\_docs\_as\_json du module, qui lira les fichiers existants et créera de nouveaux Documents en conséquence. Inutile de supprimer quoi que ce soit manuellement dans cette étape – il faut surtout s’assurer que les entrées (JSON) sont à jour, puis lancer la génération. En résumé, pour refaire cette étape il suffit de relancer le code associé (par exemple via un script principal qui instancie GenerateDocumentsAndMetadata ou en appelant directement les fonctions process\_\* sur de nouveaux JSON si besoin).

### Étape 5 : Génération de synthèses et questions (LLM)

* **Réutiliser les données existantes** : Les appels à un LLM étant coûteux, on souhaite **absolument** réutiliser les résultats si on les a déjà. Il faut donc conserver le fichier de cache outputs/trainings\_summaries\_chunks\_and\_questions\_objects.json. Lors d’une nouvelle exécution, on veillera à ce que le paramètre load\_existing\_summaries\_and\_questions\_from\_file=True soit utilisé (par défaut c’est le cas). Ainsi la méthode chargera ce JSON et reconstruira les objets de résumé/questions sans appeler le LLM. On récupérera directement la liste des objets ou documents enrichis précédemment calculés.
* **Refaire complètement l’étape** : Si on veut forcer un nouveau passage par le LLM (par exemple, si le modèle ou les prompts ont changé, ou si le contenu source a beaucoup évolué), on doit supprimer (ou déplacer) le fichier de cache trainings\_summaries\_chunks\_and\_questions\_objects.json. De plus, on s’assurera que les Documents d’entrée sont à jour (donc refaire l’étape 4 si nécessaire). Ensuite, on exécute la génération via la méthode asynchrone appropriée, par exemple SummaryWithQuestionsByChunkDocumentsService.build\_trainings\_objects\_with\_summaries\_and\_chunks\_by\_questions\_async(...). Cette méthode va détecter l’absence du fichier de cache et en conséquence appeler le LLM pour chaque formation. Après calcul, un nouveau fichier cache sera écrit. NB : Cette étape requiert aussi la présence des fichiers scraped/ et JSON d’API puisqu’elle peut reconstruire les documents de formation en interne. Assurez-vous donc de ne pas les supprimer avant de lancer l’étape 5.

En suivant ces recommandations, on peut accélérer les exécutions du pipeline en évitant de recalculer ce qui n’est pas nécessaire (en profitant des caches sur disque), ou au contraire invalider le cache pour rafraîchir l’ensemble des données lorsqu’on le souhaite. Chaque étape du pipeline a donc ses propres artefacts à gérer, mais en règle générale **il est conseillé de garder tous les JSON produits** (bruts, structurés, scrapés, résultats LLM) tant qu’on ne souhaite pas une mise à jour complète, afin de pouvoir relancer facilement des parties du pipeline sans coût excessif. En cas de doute, on peut isoler les dossiers de chaque étape (full/, scraped-full/, scraped/, all/, etc.) et ne nettoyer que ceux nécessaires en fonction de l’objectif (par ex., ne vider que scraped-full/ si on veut re-scraper les pages web mais garder les données API intactes). Ainsi, le pipeline d’ingestion reste modulable et efficace, évitant les recalculs inutiles tout en permettant une régénération complète si besoin.