1 Février 2023

Rapport de projet SOA: Machine Learning au service de l'Épitaxie par jet moléculaire

I - Objectif

Le projet a pour but d'aider Alexandre Arnoult, membre du LAAS, à réaliser des composants grâce à l'épitaxie par jet moléculaire (Molecular Beam Epitaxy ou MBE). L'épitaxie consiste à faire croître des cristaux, couche par couche, en projetant des molécules sur une surface. Ces molécules proviennent de conteneurs qui sont chauffés sous vide. Une fois que le conteneur atteint une certaine température, les molécules sont expulsées et suivent une trajectoire quasi ballistique. Le fait de contrôler à tout instant le type de molécule projeté ainsi que le pourcentage de chaque molécule permet d'aboutir à des composants quasi-parfaits. Cette technologie étant très précise et délicate, de petites différences dans les paramètres initiaux peuvent rendre le composant créé inutilisable. Or, le processus est très coûteux donc il est nécessaire de pouvoir arrêter une expérience aussi vite que possible pour réduire les pertes et donc de pouvoir surveiller le processus en détail.

C'est le but de notre projet intégrateur : créer une intelligence artificielle capable de prédire si un composant sera conforme en utilisant les données des différentes couches de l'expérience ainsi que celles des expériences passées.

Comme tout projet d'intelligence artificielle, une grande partie du temps doit être consacrée au pré-processing des données afin de les mettre sous une forme utilisable pour une intennligence artificielle. La partie SOA du projet est de créer l'architecture de preprocessing des données afin d'automatiser le processus de transformation des données de fichiers divers en base de données structurées et compatible avec un apprentissage IA. Le but est de trouver la vitesse de croissance (notre label) qui est donné pour chaque composant.

Ces données doivent être réparties par couches. En effet, nous n'avons qu'une dizaine d'expériences en tout. Parmi ces expériences, la première a été arrêtée abruptement à cause d'une erreur de manipulation et n'est donc pas très pertinente et la moitié des expériences restantes n'ont pas de label. Ce manque très important d'information nous oblige à diviser notre dataset, de sorte qu'à chaque étape de l'expérience on associé le label de l'expérience.

Le code du projet peut être trouvé içi : https://github.com/Kyriios188/mbe_preprocessing_interfaces

II - Les données

Les données que l'on a à gérer proviennent de deux sources différentes : le logiciel Crystal XE qui manipule la machine MBE, et les capteurs ajoutés en plus par les chercheurs pour comprendre plus en détail le processus.

A - Les données MBE

Les données MBE viennent sous deux formats : des fichiers textes en .log qui contiennent un récapitulatif des étapes de l'expérience et des tableaux de données en .csv qui décrivent les paramètres de la machine à chaque instant de l'expérience. Beaucoup de données ne servent à rien car elles correspondent à des processus de préparation de la machine avant de pouvoir commencer à faire un composant, comme le dégazage. Une fois ces données éliminées, une ligne des fichiers .log peut ressembler à cela :

```
0007| 2021/09/24 05:37:55,849| 270nm AlGaAs 92%
```

On retrouve le numéro d'étape, la date et l'heure où l'étape a été commencée, la taille en nanomètre de la couche que l'on souhaite produire, la molécule utilisée et son pourcentage. Cette étape correspond à une couche, mais d'autres étapes qu'il est important de surveiller existent, comme l'étape suivante :

```
0019| 2021/09/24 06:59:59,389| Descente
```

Cette étape a le même format que l'étape précédente, à l'exception qu'il n'y a pas d'informations sur la molécule ni la taille espérée de la couche. C'est parce que c'est une couche qui représente la descente en température du composant à la fin de l'expérience. Cette étape peut contenir des informations importantes pour l'apprentissage même si aucune couche n'en découle réellement.

Le deuxième type de fichier MBE ressemble à cela :

	Date/Time	As2_VAC500.Cracker_Tip_Z2.MV	As2_VAC500.Cracker_Tip_Z2.Offset
3	24/09/2021 03:59:50	0	0
4	24/09/2021 03:59:53	0	0
5	24/09/2021 03:59:55	0	0
6	24/09/2021 03:59:57	0	0
7	24/09/2021 03:59:59	0	0
8	24/09/2021 04:00:01	0	0
9	24/09/2021 04:00:03	0	0
10	24/09/2021 04:00:05	0	0
11	24/09/2021 04:00:07	0	0
12	24/09/2021 04:00:09	0	0
13	24/09/2021 04:00:11	0	0

On y voit toujours la date et l'heure, mais aussi des données diverses relatives à chaque shutter de la machine. On peut y retrouver la température objectif d'un shutter, la température mesurée dans le shutter et si le shutter est ouvert ou non par exemple. Nous n'entreront pas dans les détails de ces données car l'équipe IA n'en a jamais exprimé le besoin et ce fichier n'a donc pas été priorisé dans le preprocessing. A cause du manque de temps, ces fichiers n'ont finalement jamais été exploités.

B - Les données des capteurs

Pour comprendre plus en détail le processus, différents capteurs ont été ajoutés sur la machine MBE. Ces capteurs obtiennent 4 types de mesure :

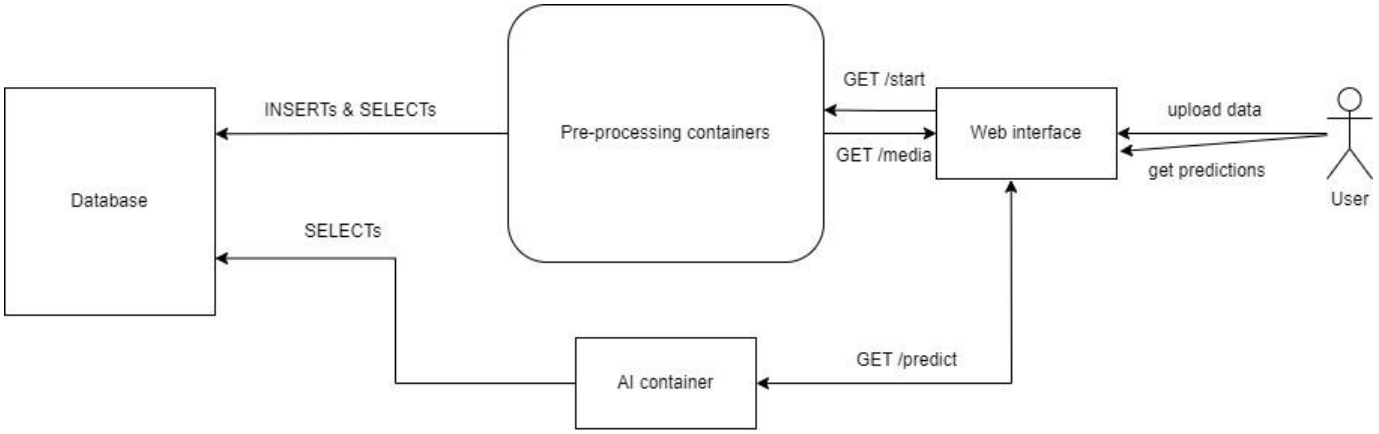
- La température du wafer
- La réflectivité
- La rugosité
- La courbure

Chacune de ces mesures est faite en fonction du temps. Dans le cas de la réflectivité, les mesures sont aussi faites en fonction de la longueur d'onde. Les données des capteurs sont stockées dans des fichiers .tdms, avec à chaque instant de mesure la valeur mesurée par le capteur.

La différence principale provient du fait que les capteurs sont enclenchés en même temps mais manuellement. Leur échelle de temps est donc relative et non absolue : on sait quelle mesure a été faite 2000 secondes après la mise en marche du capteur mais pas à quelle heure cette mesure a été prise. Heureusement, il existe des fichiers qui permettent de faire le lien entre le temps relatif et l'étape en cours. Ces fichiers sont les "Recipe Layer Number.tdms" qui contiennent à chaque nouvelle étape le temps relatif correspondant.

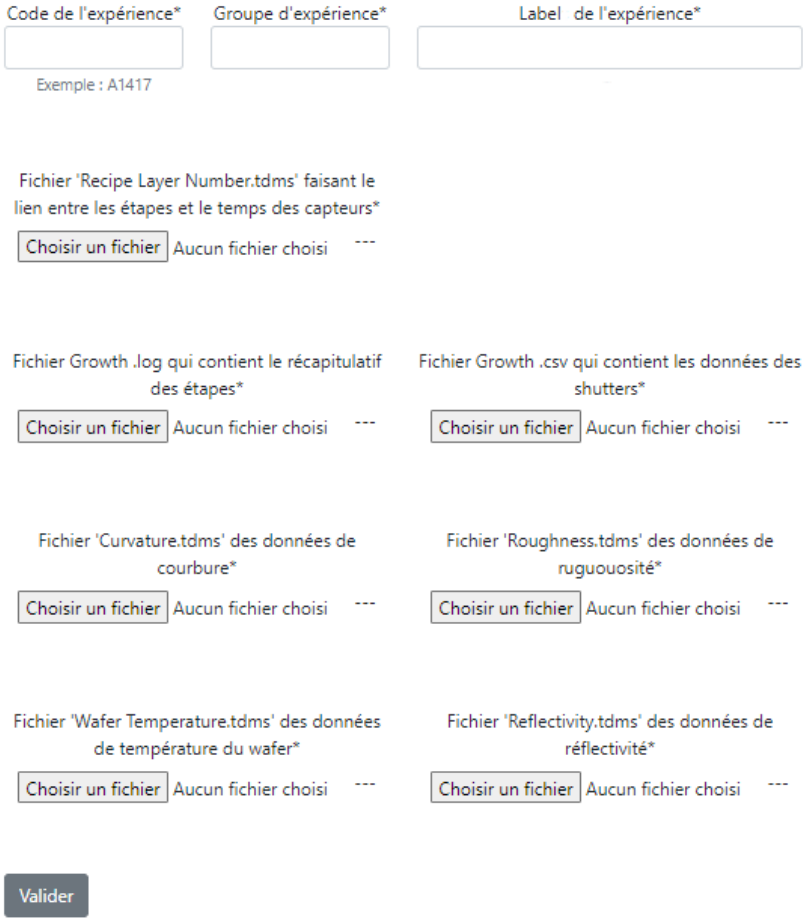
III - Vue globale de l’architecture

L'architecture de pré-processing est décrite par le schéma suivant :



L’utilisateur a accès à une interface web à partir de laquelle il peut soit entrer les données d’une expérience, soit prédire un résultat. La page d’accueil ressemble à cela :

S’il décide d’ajouter des données, un formulaire s’affiche. Le formulaire demande les fichiers décrits précédemment qui correspondent à l’expérience à ajouter dans la base de données. Le code de l'expérience correspond à la chaîne de cinq caractères qui décrivent de façon unique une expérience. Par exemple, la première expérience de notre dataset a pour code “A1417”. Le groupe d’expérience permet de regrouper les expériences similaires. Par exemple, toutes les expériences consistant à créer un miroir de Bragg. Ce groupement a été décrit comme “inutile pour l’instant” par l’équipe d’intelligence artificielle et n’a donc pas encore été implémenté.



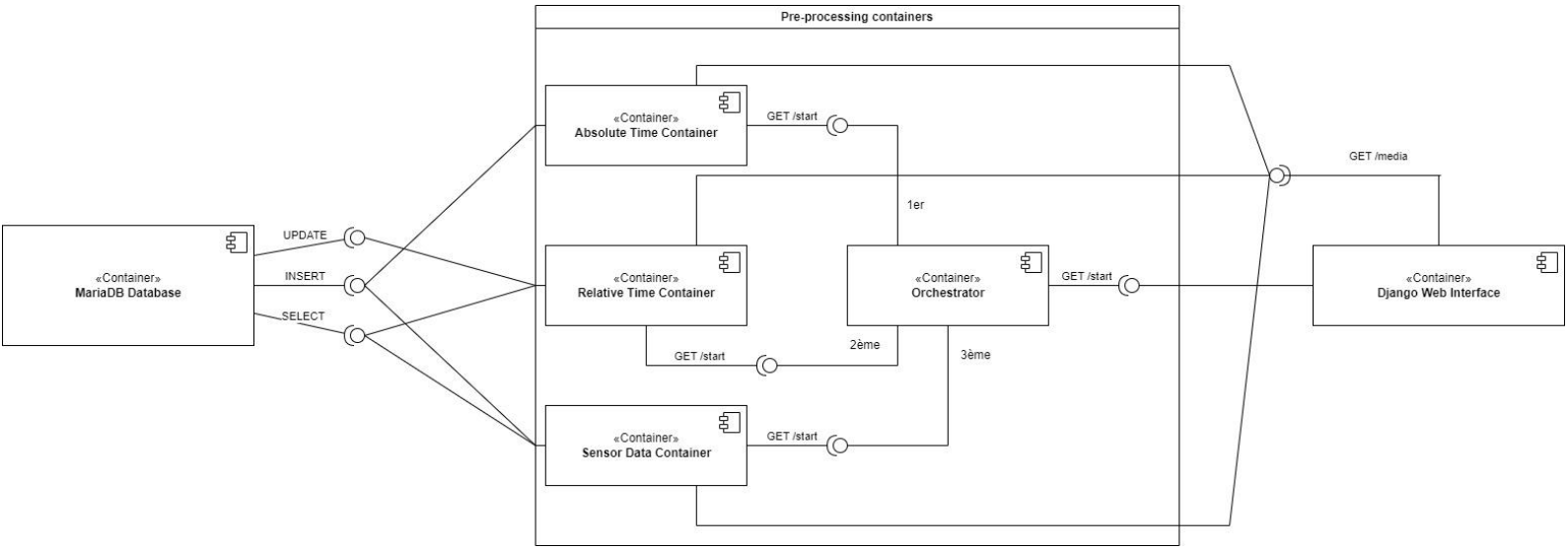
Une fois validé, le formulaire est vérifié puis les fichiers sont renommés et stockés dans le dossier media du container Docker de l'interface web. Ces fichiers sont servis par l'interface et donc téléchargeable par une simple requête GET avec le bon nom de fichier. L'interface va ensuite lancer le processus de pré-processing grâce à une requête GET vers les containers de pré-processing qui seront expliqués plus en détails dans la partie IV. Ces containers vont télécharger les fichiers dont ils ont besoin en utilisant des requêtes GET vers l'interface web. L'interface web utilise Django, un framework Python pour le web.

Les containers de pré-processing vont ensuite exploiter les documents dans un certain ordre précis pour extraire les données importantes et les insérer dans une base de données SQL qui sera décrite dans la partie V.

A cause d'un manque très prononcé de données utilisables pour l'apprentissage, il est nécessaire de créer et de compléter des données à partir de formules connues pour avoir un dataset exploitable. Malheureusement, cela implique une intervention humaine manuelle conséquente ce qui empêche le processus d'apprentissage d'être intégré dans cette architecture. Ainsi, le container d'apprentissage n'a pas été fait et le bouton "prédire un résultat" n'a actuellement pas de fonction.

IV - Les containers de pré-processing

Le fonctionnement des containers de pré-processing est décrit par le diagramme de structure composite suivant :



Comme expliqué dans la partie précédente, l'interface web Django va envoyer une requête GET pour déclencher le pré-processing. Cette requête est adressée à l'orchestrator, et contient comme arguments le code de l'expérience, son groupe et son label. L'orchestrator va accepter la requête et créer un thread enfant pour lancer le pré-processing. Le processus parent va répondre immédiatement à la requête de l'interface web afin de ne pas faire patienter l'utilisateur (on rappelle que tout se déroule après avoir appuyé sur la bouton "valider" du formulaire). L'orchestrator va ensuite appeler successivement les containers qui mettent à jour la base de données. Pour chaque appel, si le résultat de la requête est une erreur, l'orchestrator s'arrête en renvoyant une erreur à l'interface web.

Le pré-processing est divisé en trois étapes :

1. Utilisation du fichier .log qui contient le récapitulatif des étapes pour créer les objets "Expérience" et les objets "Étape" dans la base de données. Cela permet d'avoir la structure de l'expérience. Ce container s'appelle "Absolute Time Container" car il permet de faire le lien entre les expériences et leur temps de début absolu.
2. Utilisation du fichier "Recipe Layer Number.tdms" pour faire le lien entre les étapes et leur instant de début et de fin relatif.
3. Utilisation de tous les autres fichiers .tdms pour insérer les données des capteurs dans l'étape à laquelle ils correspondent.

Les containers ne peuvent être appelés que dans cet ordre car chaque container a besoin des résultats du container précédent. En effet, le container de temps relatif a besoin d'avoir

les objets étapes pour les relier à leur temps relatif et le container des capteurs a besoin d'avoir les temps relatifs des étapes pour leur associer les données des capteurs. Idéalement, il faudrait ajouter un container qui gère les fichiers contenant les données des conteneurs.

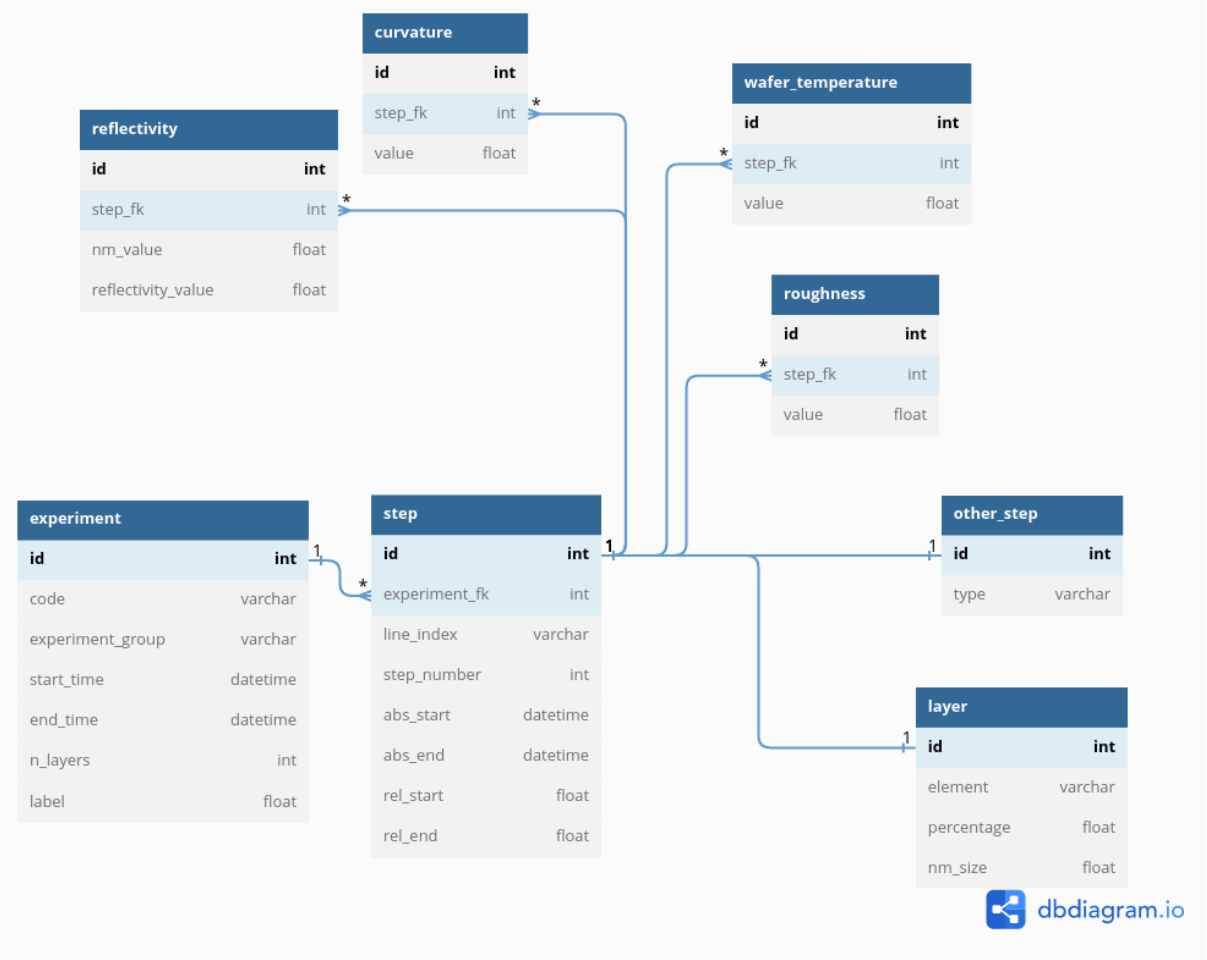
Chacun de ces containers suit le même processus. Ce sont tous des applications web Flask (Python) qui vont commencer par télécharger le ou les fichier(s) nécessaire(s). Ce téléchargement est fait grâce à une requête web sur l'url /media de l'interface Django avec ensuite le chemin vers le fichier demandé. Si la requête échoue, le container renvoie le code d'erreur de la requête vers l'orchestrator. Par exemple, le container de temps relatif va faire faire une requête GET avec l'url suivante : "/media/recipe_layer_number_tdms/{code}_recipe_layer_number.tdms" où le code est celui de l'expérience (exemple : A1417).

Ce fichier est stocké dans le dossier /files/ du container. On lance ensuite l'exploitation du fichier, fait en général avec pandas et npTDMS. L'exploitation se fait avec des requêtes INSERT, SELECT et UPDATE. Le container de temps absolu ne fait pas de requêtes SELECT car il est le premier à insérer les données et n'a donc rien d'intéressant à trouver dans la base de données. Le container de temps relatif est le seul à faire des requêtes UPDATE et ne fait pas d'INSERT car il modifie les tables des étapes pour ajouter le temps relatif correspondant à chaque étape.

Les fichiers wafer temperature ont la particularité qu'aucune librairie Python n'existe capable de les ouvrir. En effet, ils contiennent le type ExtendedFloat des fichiers .tdms mais la gestion de ce type n'a pas été implémentée dans le module npTDMS et forcer le type à être un DoubleFloat dans le code source de npTDMS résulte en des données incohérentes. Il est aussi impossible de transformer les fichiers en .csv pour la même raison. Nous n'avons donc jamais pu exploiter les fichiers de wafer temperature automatiquement.

V - La base de données

Ci-dessous un diagramme de la base des données.



On représente les étapes (table “step”) comme soit des couches (table “layer”) soit comme des étapes de contrôle diverses (table “other_step”). Il est techniquement possible avec ce schéma qu’à une étape corresponde une entrée dans la table “layer” et dans la table “other_step” mais c’est impossible en pratique dans le code.

A une entrée dans la table “experiment”, on associe autant d’entrées dans la table “step” que voulu et pour chaque entrée dans la table “step”, on associe autant de mesures des capteurs que voulu.

A chaque mesure des capteurs, on associe une unique entrée dans la table “step”. La table “reflectivity” diffère des autres car la réflectivité est fonction du temps et de la longueur d’onde.

VI - Les micro-services

Comme vu précédemment, il y a 5 micro-services dans l'architecture. Voici un résumé de leur rôle et de leur fonctionnement. Les ports sont tous exposés donc il est possible d'accéder aux interfaces depuis le navigateur avec l'adresse 172.0.0.1.

Orchestrator :

- **IP :** 172.20.0.3
- **Port :** 8003
- **Interface exposée :** /start
- **Rôle :** Appeler dans le bon ordre les containers de preprocessing (absolute_time_container, relative_time_container, sensor_data_container)
- **Arguments nécessaires pour la requête :** code (le code l'expérience, str)
- **Fichiers à télécharger :** Aucun
- **Renvoie :** 200
- **Actions en durée de processus :** appel des autres containers de preprocessing
- **Actions en fin de processus :** appels vers l'interface /end du container Django avec le code 201 ou 500 selon les réponses ses containers de preprocessing

Absolute Time Container :

- **IP :** 172.20.0.4
- **Port :** 8002
- **Interface exposée :** /start
- **Rôle :** Insérer dans la BDD les informations structurelles de l'expérience (nombre d'étapes, informations sur les étapes) dont les temps absolus de début et de fin des étapes.
- **Arguments nécessaires pour la requête :** code (le code l'expérience, str)
- **Fichiers à télécharger :** Le fichier .log qui contient le récapitulatif des étapes.
- **Renvoie :** 201 si tout s'est bien passé, 500 si le processus de preprocessing a rencontré un problème, le code d'erreur éventuel de la requête GET pour télécharger le fichier.
- **Actions en durée de processus :** Parcours du fichier .log, appels SQL pour insérer des données
- **Actions en fin de processus :** Envoie le code de statut comme réponse à l'orchestrator.

Relative Time Container :

- **IP :** 172.20.0.5
- **Port :** 8003
- **Interface exposée :** /start
- **Rôle :** Ajouter pour chaque étape les temps de début et de fin relatifs.
- **Arguments nécessaires pour la requête :** code (le code l'expérience, str)
- **Fichiers à télécharger :** Le fichier "Recipe Layer Number.tdms" qui fait la correspondance entre le numéro des étapes et leur temps de début relatif..
- **Renvoie :** 201 (voir conclusion)
- **Actions en durée de processus :** Parcours du fichier "Recipe Layer Number.tdms", appels SQL pour insérer des données
- **Actions en fin de processus :** Envoie le code de statut comme réponse à l'orchestrator.

Sensor Data Container :

- **IP :** 172.20.0.6
- **Port :** 8004
- **Interface exposée :** /start
- **Rôle :** Ajouter les données des capteurs dans la base.
- **Arguments nécessaires pour la requête :** code (le code l'expérience, str)
- **Fichiers à télécharger :** Les fichiers TDMS "Wafer Temperature", "Reflectivity", "Curvature" et "Roughness".
- **Renvoie :** 201 (voir conclusion)
- **Actions en durée de processus :** Parcours des fichiers pour insérer leurs données dans la BDD
- **Actions en fin de processus :** Envoie le code de statut comme réponse à l'orchestrator.

MariaDB Database :

- **IP :** 172.20.0.10
- **Port :** 3306
- **Rôle :** Stocke les données

VII - Déploiement

Tous les éléments décrits sont déployés dans des containers Docker. L'image de la base de données est mariadb:10.4 et toutes les autres images ont leur propre Dockerfile. Le déploiement est régi par un docker-compose.yaml.

Dans le docker-compose.yaml, on décrit pour chaque container un service associé. Chaque service correspond à un dossier dans lequel se trouve le requirements.txt du serveur Python (Django ou Flask), un Dockerfile et les fichiers de code. On décrit les dépendances de chaque service et on associe le container à une adresse IP fixe issue d'un réseau privé Docker en 172.20.0.0/16. On n'utilise pas l'adresse 172.20.0.1 qui est réservée à Docker. Pour chaque service, on expose le port sur lequel il écoute afin de pouvoir faire du débogage. Le docker-compose va build l'ensemble des containers puis les lancer.

Le seul élément particulier est que comme l'image de la base de données est un mariadb:10.4 générique, il faut créer notre schéma manuellement. On crée le mot de passe root et la base de données grâce à des variables d'environnement mais il faut aussi créer les tables et leurs relations. Pour cela, on fait en sorte que le point d'entrée de l'interface web Django soit un script Bash. Ce script bash va commencer par envoyer une requête à la base de données sur son port d'écoute 3306 grâce à netcat afin de savoir si la base est capable de répondre à une requête. Comme notre container est très simple, soit mariadb n'écoute pas encore sur le port, soit mariadb est capable d'accepter des requêtes. C'est la façon la plus simple qui a été trouvée de faire le healthcheck de la base de données.

Une fois que la commande netcat obtient une réponse, on insère les tables grâce à un fichier .sql obtenu au préalable avec la commande mysqldump et on peut lancer l'interface web.

VIII - Conclusion

Comme on a pu le constater au fur et à mesure du rapport, il reste des problèmes à régler. La liste des problèmes est présente dans le README du projet. Ceci étant dis, nous avons créé une architecture capable de convertir les fichiers MBE (à l'exception du fichier TDMS de température) en données dans la BDD SQL. Cette architecture peut être exploitée par les futurs groupes qui travailleront sur ce projet pour accélérer leur avancement mais va tout de même nécessiter de nombreuses modifications pour être dans un état professionnel.

Il reste comme piste d'amélioration majeur le fait que l'interface idéale est une interface qui fait des prédictions basées sur les données d'une expérience en temps réel. L'architecture actuelle n'est absolument pas capable de réaliser cela. Il faudrait pouvoir intégrer le processus de transformation des données dans l'architecture du LAAS de sorte à ne plus passer par des fichiers TDMS mais bien par les données en temps réel des capteurs.