



AI MARKETPLACE

Projet Interdisciplinaire

Groupe 1

Simon Fanetti Nicolas Paschoud Luca Srdjenovic Benoist Wolleb

03 juin 2022

Table des matières

1 Contexte	3
2 Objectifs	3
3 Liste des besoins	3
4 Services d'intelligence artificielle	3
5 DevOps	4
5.1 Outils utilisés	4
5.2 Pipelines DevOps	4
5.2.1 Pipeline - Branche dev	4
5.2.2 Pipeline - Merge request	5
5.3 Stages	6
5.3.1 Linting	6
5.3.2 Preparation	7
5.3.3 Test	7
5.3.4 Build	7
5.3.5 Deploy	8
5.4 Kubernetes	9
5.4.1 Déploiement	10
5.4.2 Service	11
5.4.3 Route	11
6 MLOps	11
6.0.1 Gestion des versions des données	11
6.0.2 Pipeline	12
7 Services	13
7.1 Description	13
7.2 Orchestration	13
7.3 Procédures distantes	14
7.4 Fonctionnement asynchrone	15
7.5 Annonce périodique	15
7.6 Hors du système	15
8 Engine	16
8.1 Description	16
8.2 Fonctionnalités	17
8.3 Technologie	17
8.4 Login	17
8.5 Fonctionnement asynchrone	17
8.6 Fonctionnement sans état et persistance	18
8.7 Accès concurrents	18
9 Pipelines	20
9.1 Description	20
9.2 Définition d'une pipeline	21
9.3 Nœuds	21
9.3.1 Caractéristiques principales	21
9.3.2 Nœud générique	21
9.3.3 Nœud d'entrée	22
9.3.4 Nœud de service	22
9.3.5 Nœud HTTP	23

9.3.6 Nœud conditionnel	23
9.3.7 Nœud d'itération	23
9.4 Flux de données	24
9.5 Objets binaires	25
9.6 Code dynamique	26
9.7 Pipelines automatiques	26
10 Webapp	27
11 Organisation	30
11.1 Séparation des tâches	30
12 Conclusion	32
12.1 État actuel	32
12.2 Critique	32
13 Annexes	34
13.1 Portfolio	34
13.1.1 Luca Srdjenovic	35
13.1.2 Benoist Wolleb	36
13.1.3 Nicolas Paschoud	37
13.1.4 Simon Fanetti	38

1 Contexte

Dans le cadre du cours de Projet Interdisciplinaire (PI) du Master HES-SO, il nous a été demandé de réaliser une application mettant à disposition d'utilisateurs, différents services d'intelligence artificielle à travers une plateforme web. L'idée de ce Projet Interdisciplinaire est de faire en sorte que les membres du groupe puissent travailler ensemble, afin de développer un premier exemplaire comprenant ces différentes technologies.

2 Objectifs

L'objectif de ce projet est de réaliser un système entier permettant à des utilisateurs d'utiliser des modèles de machine learning pré-définis, entraînés et interfacés pour en faciliter leurs utilisations. Chaque modèle de machine learning sera interfacé par une API REST et considéré comme un micro-service auquel il est possible de faire des requêtes. Différents modèles pourront être interfacés et permettront notamment de détecter et décrire un visage et bien plus encore.

Différents aspects techniques doivent être utilisés pour monter un tel système. Le système doit être capable de déployer des micro-services sur un cluster Kubernetes afin d'en garantir l'accès en tout temps. Un modèle de machine learning est complexe et nécessite une unité de calcul particulière ainsi qu'une grande quantité de données pour être entraîné et le système déployé devra être capable de gérer tout cela de manière automatique en introduisant des outils de MLOps.

L'objectif visé est la modularité entre les modèles: même si chaque service est utilisable en tant que tel via son API, chaîner les unités permet de créer des pipelines offrant un traitement de l'information plus complexe. Le système mis en place devra être capable d'effectuer des appels successifs aux différents services afin de faire fonctionner une chaîne de traitement.

3 Liste des besoins

Implémenter une API REST offrant un accès utilisateur à notre système. Cet accès peut être implémenté en utilisant soit des requêtes HTTP ou via une interface web.

Mettre en place un ou plusieurs microservices encapsulant un modèle de machine learning ou un service de traitement simple interfacé par une API REST. Chaque unité dispose d'une API REST spécifique et documentée selon la norme OpenAPI.

Implémenter un moteur permettant de connecter les différents services afin d'appliquer une chaîne de traitement ML.

Mettre en place une pipeline de déploiement automatique CI/CD permettant de tester et déployer nos services dans un cluster Kubernetes.

Mettre en place une pipeline MLOps capable de réentraîner un des modèles de machine learning du système.

4 Services d'intelligence artificielle

- Description de visages : Ce modèle permet de reconnaître des humains et ainsi de décrire le visage d'une ou plusieurs de ces personnes dans une image. Pour chaque visage, nous aurons une liste comprenant ses caractéristiques propres.
- Détection d'objets (non implémenté, car l'environnement spécifique à la librairie est non reproductible dans le temps imparti) : Ce modèle permet de détecter des objets dans une image et de les lister.
- Détection de visages : Retourne la position des différents visages détectés.
- Traitement d'image : Divers options de traitement d'images
 - Blur : Floute une ou plusieurs zones de l'image

- Crop : Extrait les zones spécifiées de l'image
 - Convert : Convertit une image au format jpeg
 - Analyze : Extrait les meta-données d'une image
 - Greyscale : Convertit une image en couleurs en nuance de gris
 - Resize : Redimensionne une image à la taille voulue
- Reconnaissance de chiffres : Ce modèle permet de reconnaître des images de chiffres écrits à la main. Il se base sur le dataset MNIST et est réentrainable grâce à la pipeline MLOps.

5 DevOps

Le but de DevOps dans notre projet est de pouvoir unifier le développement (**Dev**) et de gérer en même temps l'administration système (**Ops**). Ce processus va nous permettre de pouvoir éviter des problèmes de transitions entre les quatres membres du projets que nous sommes.

L'idée ici est de se baser largement sur l'automatisation des processus, c'est-à-dire de pouvoir à chaque nouvelle modification du code, lancer par exemple chaque teste, chaque build (construction d'image docker) et chaque déploiement des images construites. Pour chaque étape, nous pouvons monitorer si tout s'est correctement déroulé et aussi récupérer des informations si besoin. C'est pour cela que nous avons besoins de créer pour ce projet des pipelines CI/CD.

5.1 Outils utilisés

Dans notre projet, plus précisément pour la partie DevOps, nous utilisons les outils suivants :

- **Gitlab** : Outil de *versionning* pour notre code. Il va nous permettre créer aussi nos pipelines CI/CD qui vont être déclencher à chaque nouvelle modification de notre projet.
- **Kubernetes** : Cluster de la HEIA-FR qui va aura le rôle de plateforme de déploiement permettant la distribution et la mise à l'échelle dans un environnement de développement/productions de divers services implémentés.

5.2 Pipelines DevOps

Nous avons décidé de séparer nos pipelines en deux parties distinctes :

5.2.1 Pipeline - Branche dev

La branche dev est la branche de développement du projet. C'est une branche que nous n'allons pas modifier directement mais uniquement via des *merge requests*. Ceci afin de garantir que tout nouvelle modification est conforme et propre pour être fusionnée dans la branche *dev*.

Dans cette branche va se trouver la pipeline qui va construire tous nos services dans des images dockers à chaque nouvelle modification. Une fois la construction des images prêtes des services, ils vont être directement déployés automatiquement avec Kubernetes.

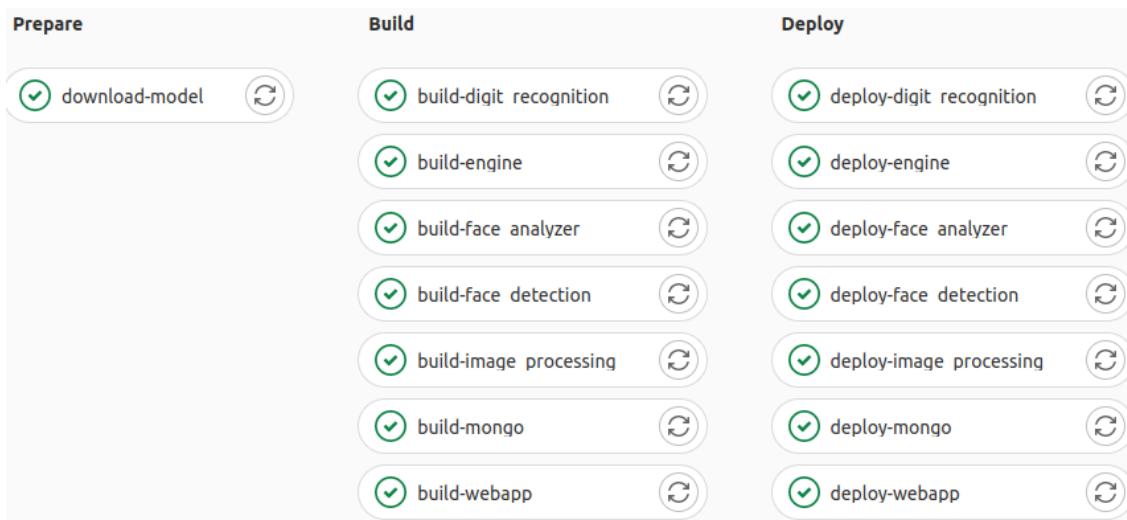


Figure 1: Pipeline de la branche dev

Sur la figure 1, nous observons que nous devons en premier préparer les modèles d'IA venant de notre serveur d'artefact MINIO. Une fois réussi, nous créons chaque image docker que nous allons stocker sur le *Registry Container* de notre projet gitlab. Une fois la construction des images terminée avec succès, nous allons finalement déployer automatiquement ces dernières sur notre environnement de développement de Kubernetes.

5.2.2 Pipeline - Merge request

Pour modifier la branche *dev* nous utilisons des branches éphémères que nous allons ensuite pousser dans une *merge request* pour être après fusionnées dans la branche *dev* pour la raison évoquée ci-dessus.

Le but de la “pipeline” pour les *merge requests* est de nous permettre de pouvoir lancer la procédure de test sur chaque nouvelle modification de services afin d’identifier d’éventuels bugs de régression. Cela va nous permettre de garantir qu’il n’y aura aucune source de bug (le moins possible en tout cas) une fois que nous allons fusionner sur la branche *dev*.

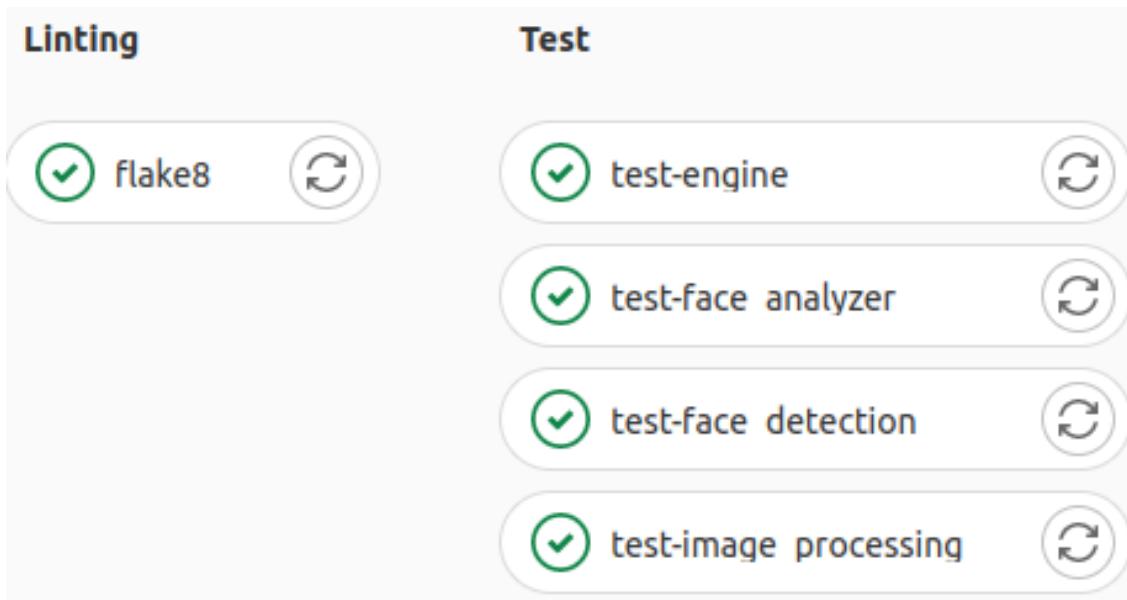


Figure 2: Pipeline d'une merge request

Sur la *pipeline* de la Figure 2, nous testons premièrement que le code respecte la convention que nous avons choisi pour ce projet afin d'y déceler d'éventuelles erreurs au niveau du code. Une fois le *linting* réussi, nous pouvons lancer les tests unitaires de chaque service modifié. Tant que la *pipeline* n'est pas valide, nous ne pouvons pas fusionner notre *merge request* dans la branche dev.

5.3 Stages

Chaque *pipeline* se décompose en plusieurs *stages*, où chacun d'entre eux exécute des jobs. Nous allons expliquer plus en détails ce que chaque stage fait.

Voici le code du fichier `.gitlab-ci.yml`

```
stages:
  - linting
  - prepare
  - test
  - build
  - deploy

include:
  - digit_recognition/digit_recognition.gitlab-ci.yml
  - engine/engine.gitlab-ci.yml
  - face_analyzer/face_analyzer.gitlab-ci.yml
  - face_detection/face_detection.gitlab-ci.yml
  - image_processing/image_processing.gitlab-ci.yml
  - mongo/mongo.gitlab-ci.yml
  - webapp/webapp.gitlab-ci.yml
```

Ce qu'il faut remarquer ici, est que nous utilisons qu'un seul projet pour tous nos différents services. Nous avons préféré la méthode d'avoir qu'un seul répertoire pour tous les services au lieu d'avoir un répertoire différent pour chaque type de service. La raison est que centraliser tous nos services dans un seul endroit est plus propre pour notre besoins parce que nous utilisons moins d'une dizaine de services que nous appellons dans notre pipeline. Sinon nous aurions dû créer une pipeline par projet ce qui était lourd.

Dans le fichier `.gitlab-ci.yml` à la racine de notre projet que *Gitlab* utilise pour lancer les pipelines, nous avons fragmenté ce fichier en plusieurs entités pour chaque service. Chaque entité aura son propre fichier `.SERVICE_NAME.gitlab-ci.yml` dans son dossier que nous inclurons ensuite sur le fichier racine. Cela nous facilite à gérer directement chaque job de chaque service et de plus, cela rend plus propre la lisibilité du code que d'avoir tout le code dans un seul fichier.

5.3.1 Linting

Le job pour la partie *linting* est d'analyser automatiquement le code pour réduire des erreurs et surtout pour améliorer la qualité de notre code. Cela va nous permettre de gagner du temps de développement et surtout de réduire le coût à trouver des erreurs dès le début.

Le *linter* choisi pour ce projet est *flake8*. C'est un *linter* pour python car tous nos services sont développés en utilisant le langage python.

A la base de notre projet se trouve un fichier de configuration `.flake8` qui va contenir toutes nos règles pour le *linter*. L'avantage aussi est qu'il extensible, c'est à dire que nous pouvons personnaliser les règles que nous voulons pour le linting. Par défaut *flake8* utilise la convention de code python appelé [PEP 8](#).

Fichier `.flake8`:

```
[flake8]
ignore = W191,E501,E701,E302,E305
exclude = .git,__pycache__,venv
max-line-length = 160
```

Si le job *linting* fait une erreur, alors la pipeline de test s'arrête. Donc, la merge request ne sera pas valide.

5.3.2 Preparation

Le but du stage *preparation* est de pouvoir récupérer un modèle d'IA de réentraînement qui est localisé dans notre serveur d'artefact MINIO pour le service *digit-recognition*. Nous en avons besoin pour la construction de l'image docker de ce service avec le modèle réentraîné et ensuite après pouvoir le déployer dans le Kubernetes.

5.3.3 Test

Ce *stage* comme son nom l'indique a pour but de tester chaque service. Il a comme prérequis que le *stage linting* réussisse afin de pouvoir être exécuté. Comme nous utilisons du python, nous avons implémenté dans chaque service ses propres tests unitaires. Ensuite il nous faut les lancer dans chaque job de chaque service via la commande :

```
python3 -m pytest --asyncio-mode=auto
```

Si tous les tests passent, alors notre *pipeline* pour une *merge request* est considérée comme valide. Dès lors, la *pipeline* dev peut-être exécutée. La praticité est que nous automatisons le plus possible nos tests afin d'éviter de les faire à la main et donc de faire des erreurs humaines. Par contre automatiser des tests à un coût en temps à prendre en considération.

5.3.4 Build

Build est la stage qui va se charger de construire les images docker de chaque service du projet. Nous avons besoin de construire ces images car pour le déploiement, notre Kubernetes va devoir les récupérer pour reconstruire nos services sur notre plateforme de déploiement. Il a comme prérequis que le stage *preparation* réussisse afin de pouvoir être exécuté.

Chaque image construite de chaque service va ensuite être stockée dans le *Container Registry* Gitlab de notre projet (voir Figure 3) :

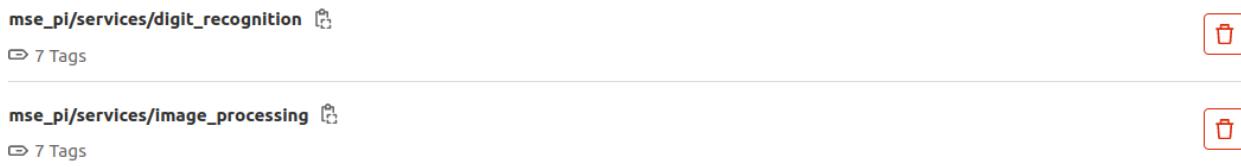


Figure 3: Container registry de gitlab

Ensuite, pour les stocker nous aurons besoin de leur attribuer un nom (**tag**) qui va nous permettre de les versionner. Par défaut, Gitlab utilise le tag *latest* pour la dernière image construite si rien n'est précisé. Nous allons créer deux tags à chaque nouvelle image (voir Figure 4):

<input type="checkbox"/> aad1f884	Published 3 days ago	
695.43 MiB	Digest: a7a24d3	

<input type="checkbox"/> latest	Published 2 hours ago	
695.67 MiB	Digest: 5c36991	

Figure 4: Tag d'image

- **\$CI_COMMIT_SHORT_SHA** : Variable prédéfinie de *Gitlab* permettant d'avoir un nom unique pour chaque nouvelle image en utilisant les huit premiers caractères de l'empreinte du *commit* dans la *pipeline*. Cela nous permet de garder un historique des dernières images avec une bonne traçabilité. Deuxième, pour déployer sur Kubernetes une nouvelle image d'un service, nous devons éviter d'avoir le même nom d'image (détails à la section 5.4.1).

- **latest**: Ce *tag* nous permet de récupérer facilement à la main la dernière version de l'image.

Au final pour une nouvelle image, nous créons deux images identiques mais avec un *tag* différent pour les raisons évoquées ci-dessus.

Voici le code qui va lancer un job pour le service dev, par exemple le service `image_processing`:

```
build-image_processing:
  variables:
    SERVICE_NAME: image_processing
  image: docker:19.03.12
  stage: build
  services:
    - docker:19.03.12-dind
  before_script:
    - docker info
    - docker login -u gitlab-ci-token -p ${CI_BUILD_TOKEN} ${CI_REGISTRY}
  script:
    - docker pull ${CI_REGISTRY_IMAGE}/${SERVICE_NAME}:latest || true
    - docker build --network host -t
      ${CI_REGISTRY_IMAGE}/${SERVICE_NAME}:${CI_COMMIT_SHORT_SHA} -t
      ${CI_REGISTRY_IMAGE}/${SERVICE_NAME}:latest -f ./${SERVICE_NAME}/dockerfile
      ./${SERVICE_NAME}
    - docker push ${CI_REGISTRY_IMAGE}/${SERVICE_NAME}:${CI_COMMIT_SHORT_SHA}
    - docker push ${CI_REGISTRY_IMAGE}/${SERVICE_NAME}:latest
  only:
    refs:
      - dev
  changes:
    - image_processing/**/*
```

Il est à noter que nous devons avoir les droits pour avoir accès à notre *container registry*. Nous utilisons la variable prédéfinie de *gitlab* \${CI_BUILD_TOKEN} avec le nom du token `gitlab-ci-token`.

Afin d'éviter de gaspiller de l'espace mémoire sur notre *Container Registry*, nous avons activé le service *cleanup* de *Gitlab*, afin qu'il garde uniquement les cinq images les plus récentes en plus de l'image *latest*. En gardant les cinq dernières images, nous pouvons garder une traçabilité de nos modifications et si besoin nous pouvons toujours revenir en arrière en redéployant une ancienne image si besoin.

5.3.5 Deploy

Une fois que nos images sont construites et stockées pour chaque service dans notre *container registry*, nous allons passer la dernière étape qui est le stage *deploy*. Ce stage a pour but de déployer automatiquement chaque service qui a été modifié directement sur la plateforme Kubernetes. Il a comme prérequis que le stage *build* réussisse afin de pouvoir être exécuté.

Voici un exemple du code de déploiement pour le service `image_processing` :

```
deploy-image_processing:
  variables:
    SERVICE_NAME: image_processing
  image: lachlanenvenson/k8s-kubectl:latest
```

```

stage: deploy
tags:
  - k8s
# when: manual
script:
  - echo $KUBE_CONFIG | base64 -d > kubeconfig.txt
  - export KUBECONFIG=$(pwd)/kubeconfig.txt
# Check if KUBECONFIG is configured
  - kubectl config get-contexts
# Change name of
  - sed -i "s~IMAGE_NAME~$CI_REGISTRY_IMAGE/$SERVICE_NAME:$CI_COMMIT_SHORT_SHA~g"
    ./${SERVICE_NAME}/kubernetes/deployment.yml
  - kubectl apply -f ./${SERVICE_NAME}/kubernetes/deployment.yml -n project-dev
# Create services
  - kubectl apply -f ./${SERVICE_NAME}/kubernetes/service.yml -n project-dev
only:
  refs:
    - dev
changes:
  - image_processing/**/*

```

Pour pouvoir déployer sur Kubernetes une image, nous devons en premier récupérer les droits pour y avoir accès. Pour cela, nous devons aller sur [Rancher](#) sur le *local dashboard* et récupérer le fichier **Kubeconfig File**.

Le problème est que nous ne pouvons pas donner le fichier à *Gitlab* dans la CI/CD. Pour palier à ce problème nous créons une variable *Gitlab* appellée **KUBE_CONFIG** dans laquelle nous allons insérer notre fichier de configuration Kubernetes encodé en base64. Ensuite dans le job, nous avons simplement besoin d'appeler notre variable **\$KUBE_CONFIG** et de la décoder en base64 pour récupérer notre configuration. Une fois que notre configuration Kube est paramétrée, nous testons si nous avons bien accès à notre Kubernetes via la commande `kubectl config get-contexts`.

Maintenant, nous pouvons déployer notre service. Nous devons donner le bon nom d'image dans le fichier de déploiement pour qu'il puisse être déployé. C'est pourquoi nous modifions notre fichier `deployment.yml` pour remplacer le nom de l'image nommé **IMAGE_NAME** par celle avec le *tag* **\$CI_COMMIT_SHORT_SHA** (expliqué dans la section [5.3.4](#)). Une fois que le bon nom d'image est paramétré dans le fichier de déploiement, nous pouvons exécuter la création d'une nouvelle entité de notre service sur Kubernetes (appelé **Pod**) ou simplement l'application de changement de ce Pod déjà existant. Un pod Kubernetes est un ensemble composé d'un ou plusieurs conteneurs Linux (ici nos images Docker construites). Après cela, Nous devons créer le service Kubernetes qui va être lié au Pod déployé précédemment. Si le service Kubernetes pour le Pod existe déjà, alors il n'y aura aucun changement appliqué.

5.4 Kubernetes

Dans cette partie, nous allons expliquer les configurations pour Kubernetes afin de compléter la phase du déploiement automatique de nos services.

Dans chaque dossier d'un service se trouve un dossier **kubernetes** contenant :

- monservice/
 - kubernetes/
 - * deployment.yml
 - * service.yml

Une exception est faite pour les services **engine** et **webapp** qui ont eux besoin d'un fichier de configuration appelé **ingress.yml** en plus afin d'être accessible depuis l'extérieur du Kubernetes à travers une route sur internet.

Pour le projet, nous avons créé un *namespace* `project-dev` dans Kubernetes où seront répertoriés tous nos déploiements, services et routes (*ingresses*) Kubernetes (voir Figure 5).

Namespace: project-dev			
<input type="checkbox"/>	Active	digit-recognition 	registry.forge.hefr.ch/mse_pi/services/digit_recog... 1 Pod / Created 9 days ago / Pod Restarts: 0 1 
<input type="checkbox"/>	Active	engine  80/http	registry.forge.hefr.ch/mse_pi/services/engine:473... 1 Pod / Created a month ago / Pod Restarts: 0 1 
<input type="checkbox"/>	Active	face-analyzer 	registry.forge.hefr.ch/mse_pi/services/face_analyz... 1 Pod / Created a month ago / Pod Restarts: 0 1 
<input type="checkbox"/>	Active	face-detection 	registry.forge.hefr.ch/mse_pi/services/face_dete... 1 Pod / Created a month ago / Pod Restarts: 0 1 
<input type="checkbox"/>	Active	image-processing 	registry.forge.hefr.ch/mse_pi/services/image_pro... 1 Pod / Created 15 days ago / Pod Restarts: 0 1 
<input type="checkbox"/>	Active	mongo 	registry.forge.hefr.ch/mse_pi/services/mongo:473... 1 Pod / Created a month ago / Pod Restarts: 0 1 
<input type="checkbox"/>	Active	webapp  80/http	registry.forge.hefr.ch/mse_pi/services/webapp:47... 1 Pod / Created a month ago / Pod Restarts: 0 1 

Figure 5: Liste de nos services déployés dans Kubernetes sous le *namespace* `project-dev`

5.4.1 Déploiement

Le fichier `deployment.yml` contient la configuration nous permettant de déclarer les changements devant être appliqués aux Pods. Le nom et le label du Pod Kubernetes auront le même nom de que le service, e.g. `engine` sera aussi `engine`.

Il y a deux spécifités à prendre en compte :

Spécifité 1 - Utilisation d'un Access Token Gitlab

Nous avons utilisé un *Access Token* de *Gitlab* pour une question de sécurité. En effet, à chaque fois que nous appelons un déploiement, le fichier `deployment.yml`, a besoin des accès *Gitlab* pour récupérer l'image Docker se trouvant sur notre *container registry*. L'utilisation du jeton a pour but d'éviter de rentrer nos identifiants *Gitlab* manuellement de chaque membre. Il est aussi plus simple d'utiliser un jeton d'authentification pour plusieurs personnes.

Nous avons créé cet *Access Token* dans le groupe `mse_pi` → `settings` → `repository` → `Access Token`.

Une fois le jeton créé, nous aurons un *token user* et *token secret*. Sur *Rancher*, nous le rajoutons dans un secret Kubernetes pour tous les *namespaces* de notre groupe pointant sur notre *container registry* de *gitlab*.

Namespace: All			
<input type="checkbox"/>	Active	mysecretdeployer	All registry.forge.hefr.ch gitlab+deploy-token-82 

Figure 6: Crédation du secret kubernetes

Nous n'avons plus qu'à le spécifier dans le fichier `deployment.yml`

```
imagePullSecrets:
- name: mysecretdeployer
```

Spécifité 2 - Modification du nom d'image

Comme explicité au job `build` de la *pipeline*, nous passons le nom d'image avec le *tag* de l'empreinte du *commit*. Pour que les modifications de l'image soit acceptées sur Kubernetes, nous devons rajouter l'option `imagePullPolicy: Always`

Fichier `deployment.yml` :

```
image: IMAGE_NAME
#image: registry.forge.hefr.ch/mse_pi/services/image_processing:latest
imagePullPolicy: Always
```

La raison est que si nous utilisons le même nom d'image avec le même *tag* (e.g. `latest`), alors même si l'image est nouvelle, Kubernetes ne fera pas la distinction et ne déployera pas la nouvelle image donnée stipulant qu'il n'y a aucun changement apporté. Pour remédier à ce soucis, il nous faut lui donner un nom d'image avec un *tag* différent à chaque nouveau déploiement. De plus, cela nous permettra aussi si besoin la possibilité d'instancier plusieurs pods pour faire de la répartition de charge.

5.4.2 Service

Le but de créer un service est que nous voulons éviter d'utiliser l'adresse IP d'un Pod car ce n'est pas le plus stable sur le temps (destruction du pod, changement d'adresse IP). L'intérêt du service est que nous n'avons plus besoin de passer l'adresse IP du Pod directement mais uniquement le nom DNS du service dans lequel il se trouve. Ce processus est automatiquement géré par Kubernetes. Grâce à cette technique, quand nous voulons qu'un service communique en local avec un autre service, nous avons juste besoin de donner le nom DNS du service Kubernetes pour atteindre notre Pod directement. Voici un exemple d'utilisation :

- name: APP_ENGINE
 value: "http://service-engine.project-dev.svc.cluster.local"

Dans le fichier `service.yml`, nous spécifions `service-` + le nom du service, par exemple `service-engine` qui va sélectionner tous les pods avec le label `engine`.

5.4.3 Route

Pour autoriser l'accès sur internet d'un service depuis l'extérieur du cluster Kubernetes, nous définissons une route HTTP/HTTPS utilisant les **Ingresses**. Nous déclarons un *ingress* pour seulement les services `webapp` pour qu'il soit accessible pour nos utilisateurs et `l'engine`.

Dans le fichier `ingress.yml`, nous déclarons un nom d'hôte, par exemple pour le service `webapp` `pi-webapp.kube.isc.heia-fr.ch` qui va exposer un port (ici le port 80) du service `service-webapp` dans lequel se trouve le Pod avec le label `webapp`.

Voici les deux routes pour le service `webapp` et `engine` :

- `webapp` : <https://pi-webapp.kube.isc.heia-fr.ch/>
- `engine` : <https://pi-engine.kube.isc.heia-fr.ch/>

6 MLOps

Le principe MLOps s'applique uniquement à un service particulier dans le projet. Il s'agit du service de reconnaissance de chiffres. L'ensemble des données utilisées pour entraîner est basé sur l'ensemble de MNIST qui contient des chiffres dessiné à la main.

6.0.1 Gestion des versions des données

Afin de pouvoir traquer l'état des données utilisé lors d'un commit, il n'est pas très adapté d'envoyer les données de manière traditionnelle avec Git. C'est pourquoi il est possible d'utiliser DVC, un outil de gestion

de version, permettant d'identifier l'état des données au moment d'un commit à l'aide d'un hash des données. Ainsi en ajoutant à Git uniquement le hash, il est possible de connaître l'état des données au même moment sans pour autant les sauvegarder dans un commit. A la place, nous sauvegardons les données sur un serveur S3 d'où nous pourrons les récupérer à l'aide du hash dans le commit Git. De plus, il est possible de préparer au préalable les opérations à effectuer lors des pipelines CI/CD afin de préparer les données pour les modèles de Machine Learning ou de traquer ces performances à l'aide d'artéfacts.

6.0.2 Pipeline

La pipeline MLOps couvre de la création du nouvel ensemble de données, jusqu'à la sauvegarde du nouveau modèle préentraîné sur le serveur S3. Dans les principes MLOps, le but est de sauvegarder les nouvelles données sur le serveur S3 à l'aide de DVC et d'ajouter l'artéfact en résultant au Git à l'aide d'un commit. L'artéfact ne contient que quelques informations permettant d'identifier les données et est donc très léger. Ce principe est utile lorsque les données peuvent évoluer, hors dans notre cas, nous nous basons toujours sur les données MNIST et sélectionnons les données voulues sur demande. C'est pourquoi nous avons décidé de ne pas signaler les artéfacts de DVC, mais d'en garder une trace dans les pipelines CI de GitLab. Ceci facilite la création du nouvel ensemble de données et l'entraînement du nouveau modèle.

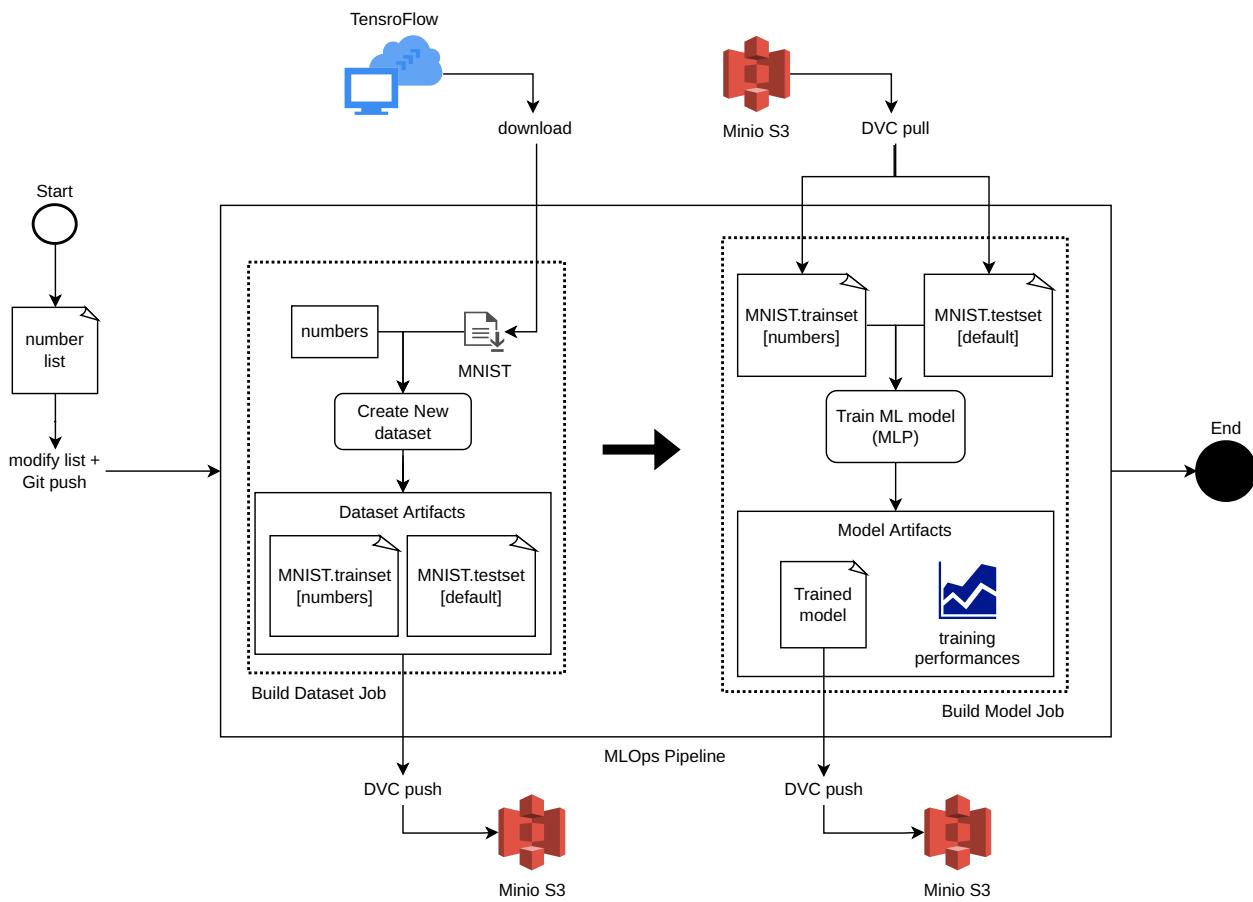


Figure 7: MLOps Pipeline

Dans le schéma ci-dessus, nous pouvons observer la pipeline MLOps déclenchée lors de la modification de la liste des nombres voulus dans le nouvel ensemble de données. Lors du premier job, un script python télécharge les données MNIST et crée le nouvel ensemble en ne gardant que les nombres voulus dans l'ensemble d'entraînement, tout en gardant aussi les nombres dans l'ensemble de test. Ceci nous permet de contrôler les résultats pour tout les nombres de base à prédire. Ces deux ensembles exportés au format csv sont envoyés au

serveur Minio S3 à l'aide de DVC, qui nous produit un fichier de meta-données par ensemble de données qui nous permet d'identifier les ressources sur le S3. Les fichiers de meta-données sont conservés comme artéfacts sur le serveur Git.

Une fois le premier job terminé, le deuxième commence et télécharge les ensembles à l'aide des artéfacts du premier job. Ensuite, un script python entraîne le modèle avec le nouvel ensemble de données et le teste avec l'ensemble correspondant. Finalement, les performances de l'entraînement sont conservées sous forme de graphiques et la justesse testée grâce à l'ensemble de test est affiché dans la console. Ces informations sont conservés en tant qu'artéfacts sur le serveur Git. Quant au modèle entraîné qui en ressort, il est extrait sous forme de fichier `h5` puis envoyé au serveur S3 sans passer par DVC avec pour nom, `mnist_model_` suivit de la liste des nombres présent dans l'ensemble d'entraînement (où les nombres sont séparés par des '_').

7 Services

7.1 Description

Dans notre système, un service est par définition un composant fournissant une ou plusieurs fonctionnalités qui lui sont propres via une interface HTTP. Ces services peuvent faire appel à des technologies de *machine learning*, comme évoqué à la section 4 ou non, ils ne sont pas différenciés depuis l'extérieur. Ils sont par contre caractérisés par un mode de fonctionnement similaire que nous allons détailler ci-dessous.

7.2 Orchestration

La première question que nous nous sommes posée lors de la conception de notre système était: Comment répartir et orchestrer les sous-tâches que les services doivent traiter?

Afin de choisir la meilleure solution, nous avons posé plusieurs critères:

- L'appel aux services ne doit pas être bloquant. En effet, ceux-ci vont potentiellement mettre du temps à traiter une sous-tâche, particulièrement dans le cas de technologies de *machine learning* où l'inférence peut être lente, ou simplement pour supporter une charge de travail importante.
- L'ordonnancement doit fonctionner de manière événementielle, nous ne voulons pas de *polling* ni du côté des services (en attente d'une tâche) ni du côté de l'engine (en attente d'un résultat).
- La solution doit permettre une montée en charge horizontale de manière relativement simple et transparente, afin de s'adapter au trafic.
- La répartition de charge doit d'opérer le plus automatiquement possible, nous ne voulons pas implémenter trop de logique nous-même dans l'engine pour gérer des services redondants.
- Les services doivent maintenir un couplage le plus faible possible avec le reste du système afin de permettre une éventuelle utilisation hors de ce contexte.
- L'interface / le canal de communication doit utiliser une technologie standard et peu contraignante permettant de créer des procédures distantes (RPC). En effet, les fonctionnalités offertes par les services peuvent être vues comme des RPC, utilisables par les utilisateurs ou depuis d'autres composants.

Nous avons envisagé plusieurs solutions et technologies permettant de satisfaire certains points, comme:

- Utiliser *gRPC* pour l'interface et les canaux de données
- Utiliser un bus de message au niveau du système, comme *Apache Kafka*
- Utiliser un socket TCP ou HTTP (*websocket*) comme canal de communication entre l'engine et les services
- Des combinaisons plus ou moins intelligentes de ces solutions

Mais aucune ne permettait de faire efficacement ce dont nous avions besoin. Nous avons donc choisi de rester sur une structure basée sur HTTP uniquement, sans bus au niveau du système, et fonctionnant sur le principe des *callback*:

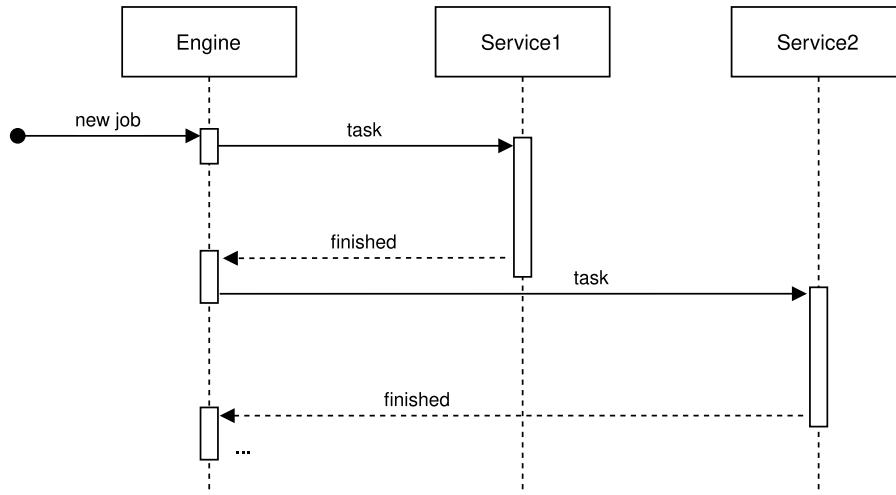


Figure 8: Renvoi des résultats

Ce choix, évidemment imparfait, nous satisfait sur la plupart des critères que nous voulions respecter:

- Les appels ne sont pas bloquants, l'utilisateur d'un service envoie ses données et le service quittance immédiatement la requête.
- Envoyer une tâche à un service se fait au moyen d'une requête, et à la fin du traitement, les résultats sont retransmis par une requête, aucun *polling n'est nécessaire*.
- Un service fonctionne de manière unitaire, indépendamment du reste du système, il est donc possible de simplement dupliquer le service pour répartir la charge.
- Comme décrit à la section 5, le système est prévu pour être déployé dans un *cluster Kubernetes*. Si une politique de montée en charge est mise en place pour les services, plusieurs *pods* d'un même service peuvent être instanciés et adressés de manière transparente, la répartition sera alors opérée au niveau du moteur de Kubernetes lui-même.
- Avec cette méthode, aucun composant global du système n'est nécessaire, les services fonctionnent donc de manière indépendante et il est possible de les faire fonctionner et de les utiliser hors du contexte de notre système.
- En définissant une interface HTTP, les services peuvent être interrogés par une multitude d'acteurs de manière standard.

La conséquence directe de ce choix est que les services ne sont pas accessibles directement depuis l'extérieur, car nous souhaitons éviter que des callbacks puissent être faits sur n'importe quelle adresse externe. Dans le cadre de notre projet, les services doivent être utilisés obligatoirement depuis l'engine, lequel devra les connaître et proposer un point d'accès dédié.

Cette méthode apporte cependant aussi son lot d'inconvénients qui sont détaillés dans les sections suivantes.

7.3 Procédures distantes

Le choix de l'orchestration étant fait, les services doivent donc décrire leur interface et proposer leurs fonctionnalités via HTTP. Malheureusement, HTTP n'a pas tout à fait été prévu pour implémenter des RPC. En REST par exemple, ce sont les verbes HTTP qui sont utilisés pour définir une opération de base (création, modification, suppression) d'une ressource. Cependant, une RPC est plus générique et on arrive rapidement aux limites de ce qu'HTTP est capable de faire.

Dans notre cas, les fonctionnalités de certains services consomment et produisent plusieurs types de paramètres à la fois: une ou des images, un ou des paramètres typés. On pense notamment aux différentes opérations de notre service de traitement d'image: l'opération de floutage prend en entrée une image et une liste de zone

et produit une image modifiée en sortie. Son prototype pourrait ressembler à: `Image blur(Image image, List[List[int]] areas)`

Pour accomplir ceci en HTTP, trois choix s'offrent à nous:

- Plusieurs requêtes, la première contenant les paramètres typés, sérialisés en json, et les suivantes les paramètres binaires.
- Une seule requête contenant tous les paramètres, les binaires encodés de manière textuelle (hexadécimal, base64), le tout sérialisé en json.
- Une seule requête en *multipart*, chaque paramètre étant une *part*, les paramètres typés sérialisés en json dans une *part* spécifique.

Nous avons retenu la troisième solution qui a l'avantage d'être faite en une seule requête et permettant l'envoi de binaires arbitrairement gros, car l'envoi de ceux-ci peut être fait sous forme de flux par la plupart des clients HTTP. Elle a le désavantage de ne malheureusement pas être très explicite dans l'interface, car les paramètres typés ne sont pas différenciés des binaires.

7.4 Fonctionnement asynchrone

Comme évoqué précédemment, le traitement d'une tâche par un service peut être relativement long, il faut donc éviter de bloquer le fonctionnement du service pendant que celui-ci est en train de travailler. Nous avons donc décidé que le traitement des tâches et le renvoi des résultats seraient effectués par des composants asynchrones, connectés via un bus de messages:

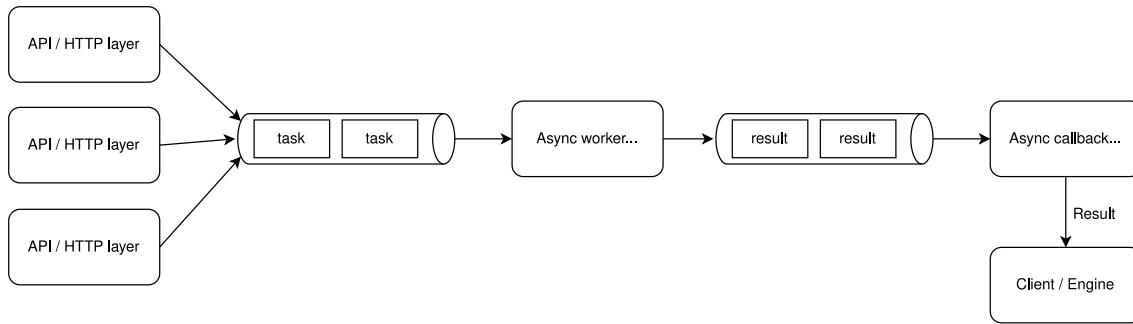


Figure 9: Composants asynchrones

7.5 Annonce périodique

Le service étant cloisonné dans le système, celui-ci doit donc être rendu disponible au travers de l'engine. Un mécanisme permettant de déclarer le service et son interface a donc été créé.

- Au démarrage, ainsi que périodiquement (toutes les 30 secondes par défaut), le service s'annonce auprès de l'engine.
- L'interface de chaque point d'accès est déclarée à l'engine en utilisant la même syntaxe que le noeud d'entrée d'une pipeline (voir [6.0.2](#)).
- Chaque point d'accès déclaré est rendu disponible par l'engine comme s'il s'agissait d'une pipeline.
- Si un service ne s'annonce plus, son point d'accès est automatiquement supprimé de l'engine

7.6 Hors du système

Un de nos critères durant la conception était que le service doit être aussi faiblement couplé que possible au reste du système. Il est donc possible de les faire fonctionner hors de ce contexte et les utiliser directement:

- Les services possèdent tous une recette *Docker* (un *Dockerfile*) permettant de construire une image encapsulant tout le nécessaire à leur fonctionnement, il est donc aisément de les construire et de les faire fonctionner dans n'importe quel contexte.
- Le mécanisme d'annonce utilise quelques variables d'environnement afin de connaître l'adresse de l'engine, il suffit donc de les omettre afin de désactiver cette fonctionnalité.
- L'API du service est automatiquement documentée à son point d'accès `/docs`.
- Utiliser une fonctionnalité du service nécessite de lui passer les paramètres de requête `callback_url` et `task_id` qui seront utilisés pour retourner le résultat.

Ce dernier point est potentiellement le seul qui peut gêner l'utilisation du service hors contexte, car il nécessite de disposer d'un serveur HTTP permettant de récolter les résultats du service. Il est néanmoins relativement facile de permettre au service de conserver les résultats d'une tâche afin de permettre leur récupération ultérieure, mais ça n'est pas implémenté pour le moment:

- La création d'une tâche attribue déjà un identifiant unique si `task_id` n'est pas renseigné.
- Les résultats doivent être stockés, en mémoire ou enregistrés dans un dossier temporaire s'il s'agit de binaires.
- Un point d'accès doit être ajouté afin de récupérer les résultats en fournissant l'identifiant de tâche, similaire à ce qui est implémenté sur l'engine.
- Un mécanisme permettant de nettoyer périodiquement les résultats non récupérés doit être implémenté, similaire à ce qui est implémenté sur l'engine.

Notons enfin que les fichiers `readme.md` du dépôt contenant les services de notre projet (voir section 12) documentent comment ceux-ci peuvent être construits et exécutés.

8 Engine

8.1 Description

Un des objectifs principaux de ce projet est de permettre d'interfacer / de chaîner les services disponibles dans le système afin de créer des chaînes de traitement que nous appelons des *pipelines*. Si chaque service offre des fonctionnalités unitaires basiques, les combiner permet de définir des fonctionnalités plus riches. Par exemple, en disposant des services “détection de visage” et “traitement d’image”, les chaîner permet de flouter les zones contenant des visages dans une image:

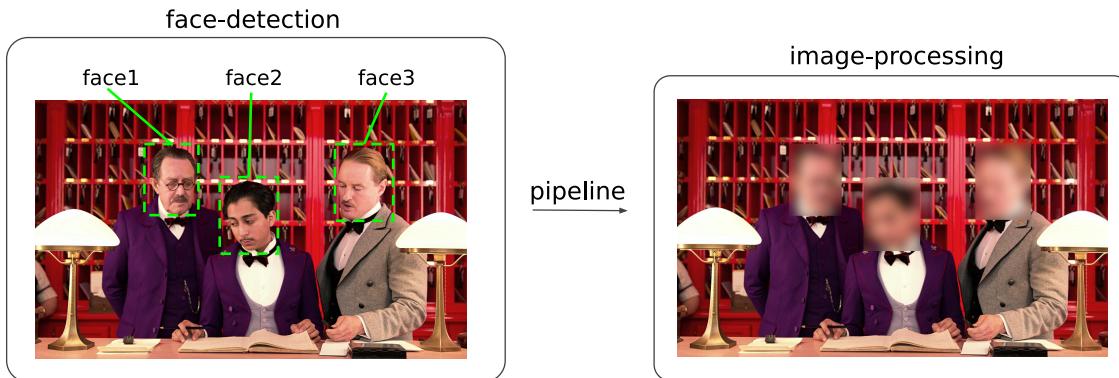


Figure 10: Floutage des visages

Toute la difficulté est de faire transiter les données d'un service à un autre. Dans l'exemple ci-dessus, la sortie du service de détection de visages est une simple liste de zones, donc une liste de liste de 4 entiers, et le service

de traitement d'image consomme une image ainsi que des paramètres qui varient selon l'opération à effectuer. Il s'agit de la première mission de notre moteur, que nous appelons *engine*.

C'est en effet sur ce composant que nous avons choisi de mettre l'accent dans ce projet, car cela nous intéressait de concevoir un système permettant de créer des chaînes de traitement de manière flexible et dynamique.

8.2 Fonctionnalités

L'*engine* est responsable des opérations suivantes:

- Collecter les services disponibles dans le système et en faire des pipelines permettant d'interagir directement avec chacun (voir chapitre 7)
- Permettre la définition de pipelines faisant appel à plusieurs services (voir chapitre 9)
- Gérer la création de tâches, la réception de leurs paramètres et le fonctionnement de la tâche dans le graphe de la pipeline associée
- Gérer le stockage des données binaires pour chaque tâche, y compris leur durée de vie selon une politique de rétention
- Offrir une interface HTTP de type REST permettant d'utiliser les fonctionnalités ci-dessus depuis l'extérieur

8.3 Technologie

Comme évoqué dans le chapitre 7, les services ne sont pas accessibles depuis l'extérieur du système, l'*engine* est donc le seul point d'accès externe pour l'utilisation des fonctionnalités. Les technologies que nous avons choisies pour implémenter ce système sont *python* et *FastAPI*.

Un langage haut niveau tel que python permet de créer un système complexe en fonctionnalités, facilement testable, et son aspect interprété permet d'exécuter du code dynamiquement à l'intérieur des pipelines. Le module FastAPI permet de créer une interface REST de manière très simple, mais reste riche en fonctionnalités.

8.4 Login

L'appel aux fonctionnalités de l'*engine*, y compris l'utilisation, la création et la suppression de pipelines, n'est pas soumise à un login. Cependant, l'impact est relativement limité, car même si n'importe qui peut supprimer des pipelines existantes, il est très difficile d'accéder à des données d'une tâche de quelqu'un d'autre sans connaître son identifiant exact, ceux-ci étant des UUID v1. Nous avons donc considéré qu'il s'agissait d'une fonctionnalité optionnelle qui dépassait le cadre du projet.

8.5 Fonctionnement asynchrone

Étant donné que:

- Une pipeline représente l'exécution de n services
- Chacun de ces services peut potentiellement avoir besoin de temps pour traiter les données, on pense particulièrement aux services de *machine learning* qui ont parfois besoin d'un temps non négligeable pour l'inférence
- L'interface pour utiliser le système se base sur HTTP

Il est fort possible que, lors d'une utilisation synchrone du système, le temps de traitement dépasse la limite d'attente du client HTTP et la connexion est fermée. Nous avons donc décidé de le faire de manière asynchrone, ce qui permet au système de fonctionner correctement même en cas de charge de travail importante:

- 1) L'utilisateur effectue une requête sur le point d'accès d'une pipeline en envoyant les données dans le corps de la requête
- 2) L'*engine* stocke les données, crée la tâche et renvoie immédiatement son identifiant à l'utilisateur

- 3) L'utilisateur effectue périodiquement une requête à l'engine afin de connaître l'état d'avancement de la tâche en fournissant son identifiant
- 4) Lorsque la tâche se termine, son état change
- 5) L'utilisateur peut récupérer le résultat associé

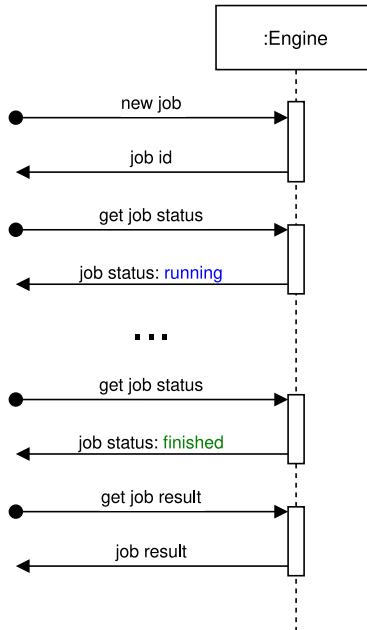


Figure 11: Floutage des visages

8.6 Fonctionnement sans état et persistance

C'est ici un choix important que nous avons fait. En effet, comme nous avons développé le projet en mode agile, avec des déploiements automatiques très fréquents, nous voulions que cela n'impacte pas le fonctionnement global du système. L'engine fonctionne donc sans état (*stateless*), aucun état fonctionnel n'est conservé en mémoire hormis lorsqu'une requête est en cours de traitement. Cela permet aussi de pouvoir relancer le service en cas de panne dans que le système ne soit perturbé.

Ceci signifie qu'à chaque requête, les variables et états doivent être chargés depuis un espace de stockage approprié. Pour ce faire, nous avons choisi de déployer un serveur de base de données non relationnelle orientée document, [MongoDB](#). Celui-ci permet en effet de manière très efficace de stocker des documents de type clé-valeur et il existe de très bons modules python permettant d'y accéder.

Enfin, comme plusieurs de nos services et pipelines peuvent consommer et produire des images, tous les fichiers binaires qui transitent par l'engine sont eux aussi stockés dans un espace de stockage dédié. Pour cela, nous avons choisi d'utiliser le protocole S3 qui est le standard actuel *de facto* pour le stockage objet.

8.7 Accès concurrents

L'engine peut recevoir des requêtes sur son API autant depuis l'extérieur que depuis l'intérieur du système:

Depuis l'extérieur:

- Création d'une nouvelle pipeline
- Suppression d'une pipeline
- Création d'une tâche dans une pipeline
- Interrogation de l'état d'une tâche

- Récupération des résultats d'une tâche

Depuis l'intérieur:

- Déclaration d'un nouveau service
- Fin de traitement d'une sous-tâche par un service du système

Comme plusieurs sous-tâches peuvent fonctionner en parallèle dans une pipeline, il est nécessaire de prévenir les accès concurrents. En effet, la fin de traitement d'une sous-tâche va amener à modifier son état, et il est possible que deux accès concurrents mènent à un état incohérent du système:

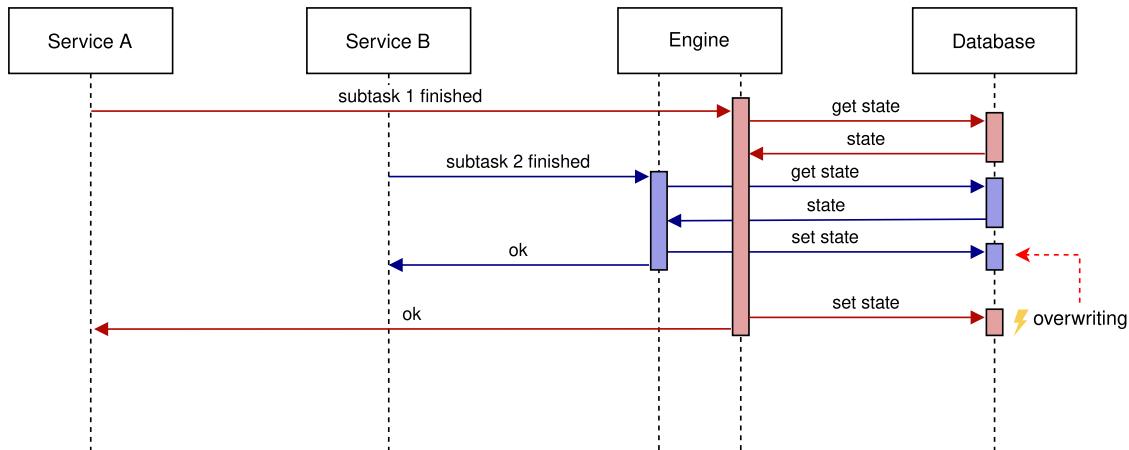


Figure 12: Accès concurrent problématique

- A: Fin de la sous-tâche 1, récupération de l'état de la tâche dans la pipeline
- B: Fin de la sous-tâche 2, récupération de l'état de la tâche dans la pipeline
- B: Modification de la pipeline avec la sous-tâche 2 terminée
- A: Modification de la pipeline avec la sous-tâche 1 terminée

À la fin de cet enchaînement, la modification opérée par *A* a écrasé la modification de *B* et la pipeline sera alors bloquée, car le système sera en attente de la fin de la sous-tâche 2 alors que celle-ci est bien terminée.

Pour régler ce problème dans le fonctionnement interne de l'engine, toutes les opérations qui sont amenées à modifier l'état d'une tâche sont faites de manière séquentielle via un bus de message:

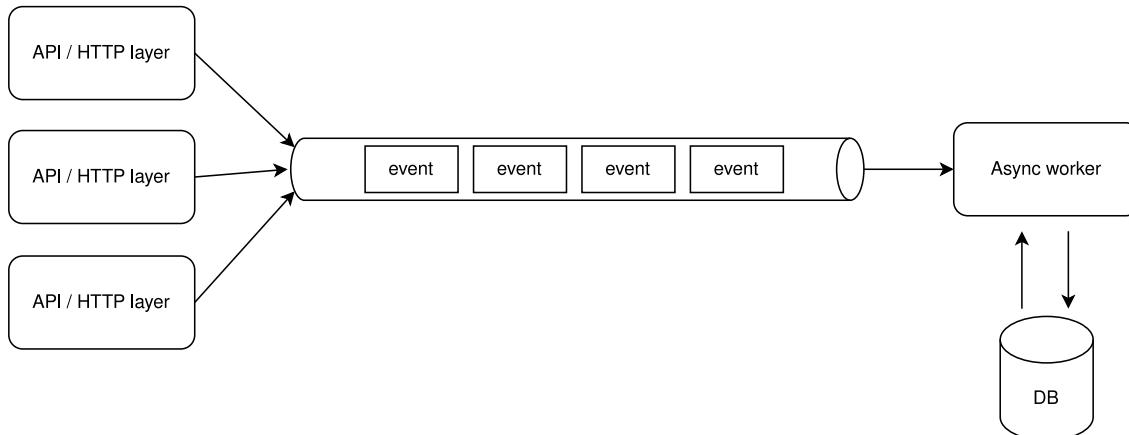


Figure 13: Bus de message

Le même scénario que ci-dessus devient donc:

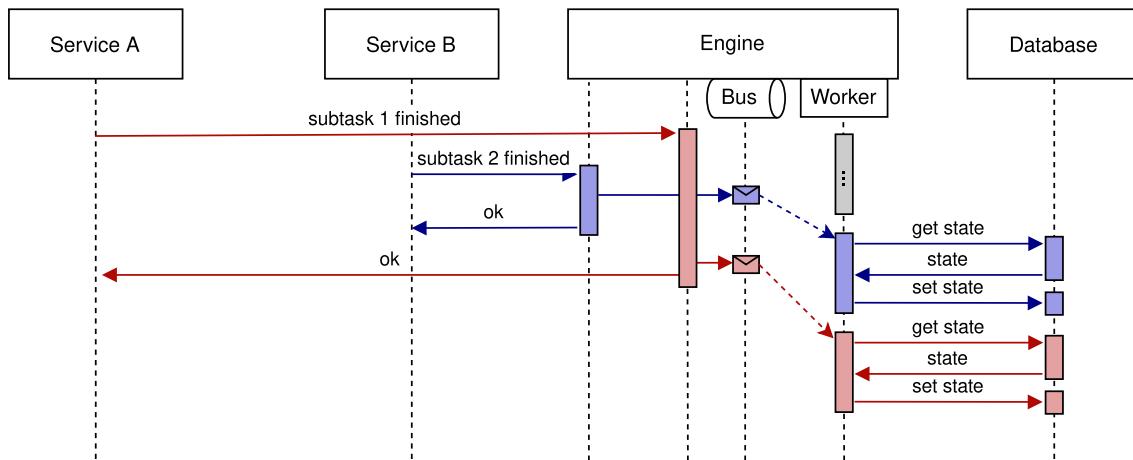


Figure 14: Suppression de l'accès concurrent

Tous les événements qui vont modifier une tâche sont placés dans une file et un unique consommateur les traite, en évitant ainsi de modifier la même donnée en parallèle. Cependant, cette méthode ne garantit l'atomicité de modification qu'à l'intérieur d'une instance de l'engine. S'il devait y en avoir plusieurs, il faudrait utiliser un mécanisme plus haut niveau, comme un bus de message au niveau du système par exemple. Par conséquent, l'engine est le seul service qui n'est pour l'instant pas possible de faire fonctionner en redondance.

9 Pipelines

9.1 Description

Comme évoqué de nombreuses fois, le but de l'engine est de permettre la définition et d'utiliser des pipelines. Celles-ci représentent une suite d'opération à effectuer pour accomplir une tâche, de manière assez similaire à un langage de programmation, mais plutôt orientée sous forme de flux de données. On parle alors du *pattern "Pipe and filter"*.

En effet, une pipeline peut être représentée sous la forme d'un graphe orienté, où chaque noeud est une opération. Pour la plupart des cas d'utilisation, le graphe est sans circuit (*DAG*), mais il est tout de même possible de créer des boucles pour des cas d'utilisation bien spécifiques, voir [9.3](#).

Les noeuds peuvent s'effectuer en parallèle et la dépendance est assurée, par défaut un noeud ne peut commencer que si tous ses prédecesseurs sont terminés.

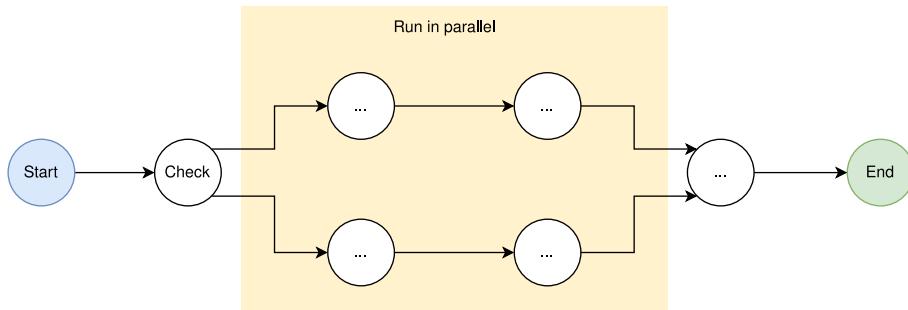


Figure 15: Les branches sont exécutées en parallèle

9.2 Définition d'une pipeline

La création d'une pipeline s'effectue en envoyant sa définition à l'engine en POST sur le point d'accès `/services`. La structure de la pipeline doit être sérialisée en json dans le corps de la requête et est spécifiée de la manière suivante:

```
{
  "type": "pipeline",
  "nodes": [NODES]
}
```

Où le champ `nodes` est la liste de tous les noeuds que comporte la pipeline (voir ci-dessous pour la spécification des noeuds) quel que soit l'ordre. En effet, la précédence des noeuds est spécifiée dans un champ spécifique. Des exemples de pipelines fonctionnelles sont présents dans le dépôt du projet.

9.3 Nœuds

9.3.1 Caractéristiques principales

- Une pipeline possède forcément un noeud d'entrée (`entry`) et de sortie (`end`), afin que l'engine sache où commence et quand se termine une tâche.
- Un noeud ne peut commencer que si tous ses prédecesseurs sont terminés, comportement modifiable.
- Un noeud prend automatiquement en entrée les données de sortie de ses prédecesseurs, comportement modifiable par une directive appropriée.
- Un noeud peut définir un ou plusieurs successeurs, ce qui permet de créer des branches exécutées en parallèle.

De plus, plusieurs champs d'un noeud peuvent contenir du code python qui sera exécuté dans le contexte de la pipeline, voir [9.6](#). Ces caractéristiques sont définies dans la classe `Node` qui est parente de tous les autres types de noeuds.

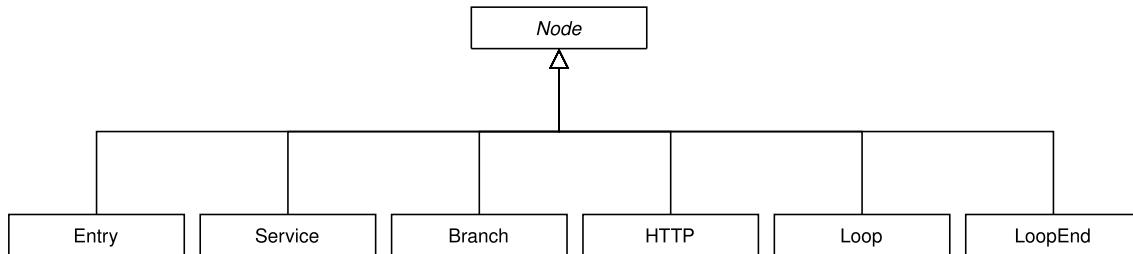


Figure 16: Classes de noeuds

9.3.2 Nœud générique

Un noeud générique est une instance de la classe `Node`, parente de tous les autres noeuds. Elle permet principalement d'exécuter du code python arbitraire lors de son exécution ou simplement de transmettre des flux de données en profitant du comportement de transit par défaut qui va automatiquement transférer les données de sortie de tous ses prédecesseurs dans sa sortie.

Sa représentation est un objet clé-valeur comportant les champs suivants, ceux en gras étant obligatoires:

- **`id`**: Identifiant unique du noeud.
- **`type: node`**
- **`params`**: Dictionnaire pouvant contenir des valeurs arbitraires, celles-ci pourront être utilisées par la suite pour les injecter en entrée, voir [9.4](#).

- **input:** Directives définissant les entrées du nœud (voir 9.4), si non présent, le nœud prend automatiquement les sorties de ses prédécesseurs.
- **next:** Liste d'identifiants des successeurs directs.
- **ready:** Code python évalué dynamiquement dans le contexte de la pipeline (voir 9.6) afin de modifier la condition de démarrage du nœud, par exemple pour accepter qu'un seul de ses prédécesseurs soit terminé avant de commencer.
- **before:** Code python exécuté dynamiquement dans le contexte de la pipeline (voir 9.6) juste après la préparation des données d'entrée, avant le démarrage de traitement du noeud.
- **after:** Code python exécuté dynamiquement dans le contexte de la pipeline (voir 9.6) juste après l'exécution (réussie) du nœud.

9.3.3 Nœud d'entrée

Ce noeud spécial de type **entry** définit le point de départ de la pipeline reconnu par l'engine. Ce nœud permet aussi de définir l'API de la pipeline, son point d'accès et les données attendues. Son unique champ spécifique supplémentaire est **api** qui doit être un objet clé-valeur contenant les champs suivants:

- **route:** Point d'accès qui sera créé pour cette pipeline dans l'engine `/services/<route>`.
- **body:** Description des données que prend la pipeline en entrée.
- **summary:** Description courte de la pipeline qui sera affichée dans la documentation automatique de l'engine.
- **description:** Description de la pipeline qui sera affichée dans la documentation automatique de l'engine.

Le champ **body** peut être de plusieurs formes:

- Un objet clé-valeur, contenant les champs attendus et une valeur d'exemple: `{"name": "Myself", "age": 31}`. Dans ce cas, le point d'accès attend un objet json qui respecte la structure spécifiée et les types seront validés.
- Une chaîne de caractères, par exemple **image**. Dans ce cas, le point d'accès attend une requête *multipart* contenant un champ **image** qui sera interprété dans la suite de la pipeline comme une ressource binaire.
- Une liste de chaînes de caractères, par exemple `[image1, image2]` qui définissent, comme le point précédent, une liste de champs binaires attendus en *multipart*.

9.3.4 Nœud de service

Ce noeud de type **service** permet d'utiliser les services présents dans le système, c'est-à-dire en envoyant la requête en HTTP à ceux-ci contenant un identifiant de tâche et une adresse de *callback*, comme spécifié à la section 7. Le nœud ne se termine que lorsque le résultat ou l'erreur revient par le callback. Son unique champ spécifique supplémentaire est **url** qui définit la route à appeler pour contacter le point d'accès voulu du service.

La requête effectuée au service est toujours de type POST et son corps contient automatiquement les données que le nœud reçoit en entrée, donc soit de manière automatique en reprenant les sorties des prédécesseurs, ou spécifié par la directive **input**. Trois situations sont possibles:

- Le nœud n'a reçu que des données typées, dans ce cas celles-ci sont sérialisées dans un objet json et envoyé dans le corps de la requête avec un "Content-Type" `application/json`.
- Le nœud reçoit un ou plusieurs champs binaires, ceux-ci sont alors transmis dans une requête *multipart*.
- Le nœud reçoit à la fois des binaires et des données typées, les binaires sont alors transmis en *multipart* et les données sont sérialisées en json et transmises dans une *part* appelée **data**.

Si le service ne peut pas être contacté ou si une erreur HTTP est détectée à l'envoi de la sous-tâche, le nœud sera ré-exécuté plusieurs fois avec un délai grandissant avant de se mettre en état d'erreur et interrompre la pipeline:

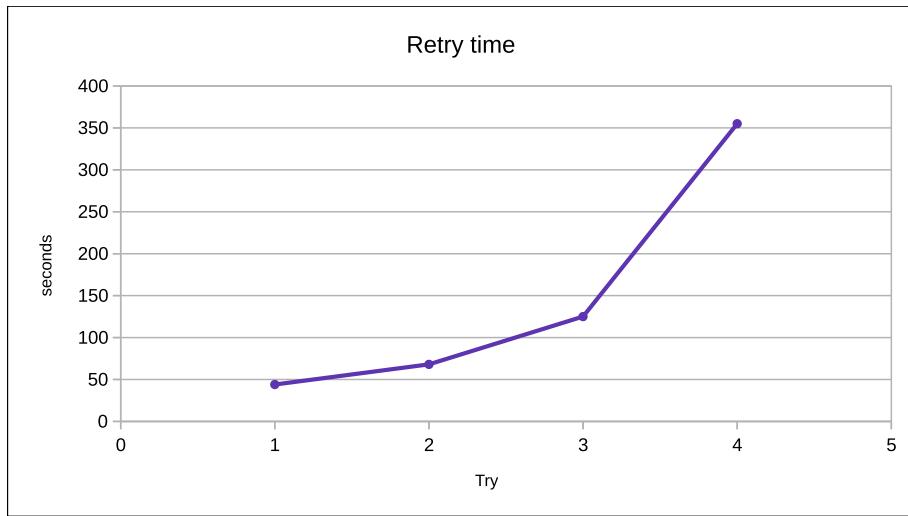


Figure 17: Delais d'attente pour chaque essai

9.3.5 Nœud HTTP

Ce nœud fonctionne quasiment de la même manière que le nœud de service, à l'exception qu'il est fait pour interroger une ressource HTTP de manière synchrone. En effet, si le nœud de service ne se termine que lorsque le *callback* est appelé, celui-ci attend que la requête se termine, il permet donc de s'interfacer avec des systèmes plus classiques. Ses paramètres spécifiques sont les suivants:

- **type: http**
- **url:** Adresse à interroger.
- **verb:** Verbe HTTP à utiliser, GET par défaut.

9.3.6 Nœud conditionnel

Comme son nom l'indique, ce nœud permet de modifier le comportement de la pipeline de manière conditionnelle lorsqu'il est exécuté. C'est notamment avec celui-ci que l'on peut créer des boucles dans le graphe. Ses paramètres spécifiques sont:

- **type: branch**
- **if:** Code python évalué dynamiquement dans le contexte de la pipeline (voir 9.6) afin de vérifier la condition du nœud, le résultat de cette exécution va directement exécuter les directives des branches **then** ou **else**.
- **then:** Branche à exécuter si l'évaluation de la condition est vraie.
- **else:** Branche à exécuter si l'évaluation de la condition est fausse.

Une branche d'exécution est un objet clé-valeur pouvant contenir les directives suivantes:

- **next:** Liste des prochains noeuds à exécuter dans la pipeline si cette branche est choisie, cela permet d'effectuer des embranchements conditionnels, y compris des boucles, dans le graphe de la pipeline.
- **out:** Directive similaire à **input** qui va permettre de transmettre les données spécifiées si la branche est choisie, ce qui permet par exemple de filtrer le flux de données.
- **exec:** Exécuter du code python arbitraire dans le contexte de la pipeline si la branche est choisie.

9.3.7 Nœud d'itération

Ce nœud permet d'itérer sur une ressource et de créer une sous-structure de la pipeline pour chaque élément. Pratiquement, lorsqu'il est exécuté, ce nœud va instancier autant de branches contenant les nœuds spécifiés,

où chacun sera responsable de traiter un élément. Toutes ces branches sont des successeurs du nœud de boucle et ils sont tous connectés à un nœud de fin de boucle. Ce la permet ainsi d'attendre la fin de traitement de chaque élément en utilisant le mécanisme de base de la pipeline.

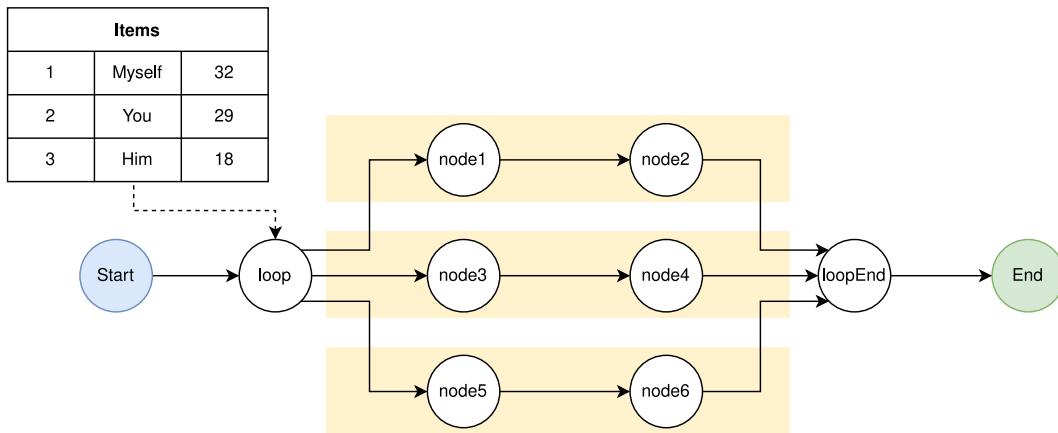


Figure 18: Sous-graphe généré pour 3 éléments

Les paramètres spécifiques à ce nœud sont:

- **type: loop**
- **items:** Identifiant de la ressource sur laquelle itérer, celle-ci doit être une liste.
- **nodes:** Liste de nœuds qu'il faut instancier pour chaque élément de l'itération.
- **collect:** Identifiant des données à collecter dans chaque branche, cela permet de regrouper tous les résultats et de les placer en sortie du noeud de boucle.

Afin que l'ensemble fonctionne, il y a quelques spécificités sur les nœuds instanciés automatiquement:

- Le premier nœud de la liste est automatiquement celui de départ, les autres peuvent être spécifiés dans le désordre.
- Leur identifiant sera postfixé d'un numéro unique, afin qu'il reste unique au sein de la pipeline.
- Chaque nœud possède un attribut `_loop` qui correspond à l'élément à traiter.
- Les successeurs des nœuds spécifiés dans le champ `next` doivent rester à l'intérieur de la branche d'itération (ils ne doivent pas référencer un autre nœud dans le graphe) ou référencer le noeud de fin de la boucle `loopEnd`.
- La directive `input` de chaque nœud peut utiliser la sortie d'un nœud de la même branche, il suffira de le préfixer avec `loop..`

9.4 Flux de données

Une des notions fondamentales pour que les pipelines puissent fonctionner est le flux de données. En effet, chaque nœud peut consommer et produire des données. La sortie d'un nœud, stockée dans son champ `out`, dépend de son type:

- La sortie d'un nœud générique est la copie de ses paramètres d'entrée
- La sortie d'un nœud `entry` correspond aux données reçues dans la requête qui initie la tâche dans la pipeline, ce qui correspond aux champs spécifiés dans `api:body`.
- La sortie d'un nœud `service` correspond aux résultats de l'exécution du service reçu via le `callback`.
- La sortie d'un nœud `branch` est potentiellement définie par la directive `out` de la branche exécutée.

- La sortie d'un nœud `loop` est potentiellement définie par la directive `collect` lorsque l'itération est terminée.

L'entrée d'un nœud est stockée dans son champ `input`, elle est par défaut l'agrégation des sorties de tous ses prédecesseurs, mais elle peut être modifiée de manière explicite avec une syntaxe de composition: `A.B.C.D`. Cela permet de définir de manière très simple des entrées spécifiques, par exemple:

```
input:
{
    img: pipelineEntry.out.avatar,
    txt: someOtherNode.out.analysis
}
```

Le nœud comportant cette directive recevra en entrée l'image reçue dans la requête de l'utilisateur dans un champ `img`, ainsi que le champ `analysis` de la sortie d'un autre noeud dans le champ `txt`.

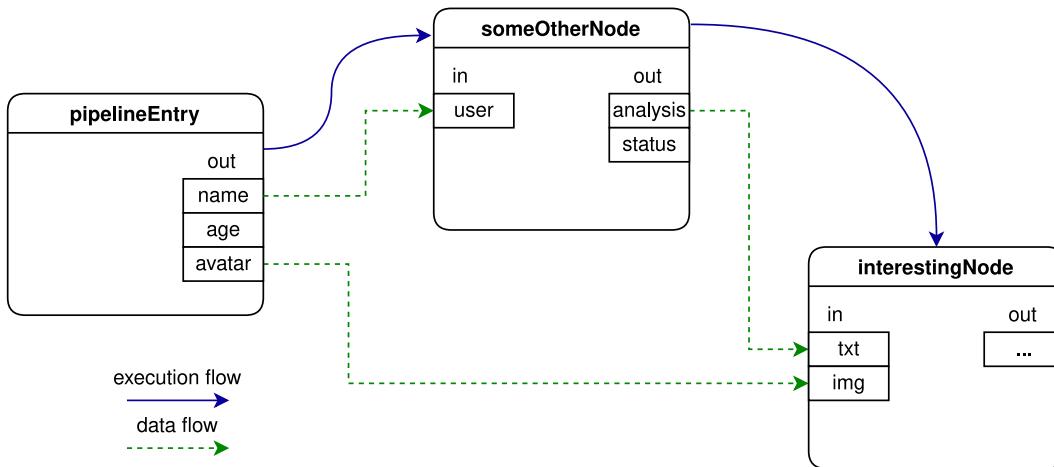


Figure 19: Exemple de flux de données

Ce même mécanisme est utilisé pour les directives `items` et `collect` d'un nœud `loop`.

9.5 Objets binaires

Comme décrit dans le chapitre 8, les objets binaires sont stockés sur un serveur de stockage objet S3. Aucun n'est stocké dans la base de données, ni en RAM lorsqu'ils ne sont pas utilisés. Le mécanisme de flux de données décrit ci-dessus nous permet de "traquer" les ressources binaires et de les transmettre lorsque c'est nécessaire.

En effet, dans la pipeline, un champ permet d'effectuer la correspondance entre les entrées/sorties de nœuds et l'éventuel objet binaire qui doit être transmis. Pour reprendre l'exemple ci-dessus, la ressource binaire a été tracée comme suit:

- La requête arrive sur le point d'accès de la pipeline, le corps contient une *part* nommée "avatar" qui est une image.
- Dès sa réception, cette image est stockée sur le serveur S3 en lui attribuant un identifiant UUID v1.
- La ressource est alors enregistrée dans la table traçant les objets binaires de la pipeline: `pipelineEntry.out.avatar: <UUID>`.
- Plus loin dans l'exécution, le nœud `interestingNode` a besoin de la ressource `pipelineEntry.out.avatar` qui se trouve dans la table de correspondance des objets binaires, on y ajoute donc `interestingNode.input.img: <UUID>`.

- Ainsi de suite, chaque nœud ayant besoin d'une ressource identifiée comme objet binaire sera enregistrée dans la table de correspondance.
- Lorsqu'un noeud a besoin d'utiliser la ressource effective, c'est-à-dire le fichier lui-même, il suffit de le récupérer sur le serveur S3 en utilisant l'identifiant.

Cela nous permet de constamment tracer où un objet binaire est utilisé afin d'automatiquement le transmettre ainsi que le supprimer au nettoyage de la pipeline. Évidemment, cela ne fonctionne que si les objets binaires sont transmis par le mécanisme des directives d'entrées/sorties, si ces ressources sont modifiées dans du code dynamique, comme le `after`, c'est sa responsabilité de gérer les éventuels enregistrements supplémentaires dans la table de correspondance.

9.6 Code dynamique

Certaines directives, comme `ready`, `before`, `after` ou d'autres vont exécuter ou évaluer du code python dynamiquement. Le code évalué est supposément court et doit renvoyer un résultat, ce qui est utilisé pour `ready` et `branch:if` par exemple, ou exécuté. Cela correspond aux deux appels python natifs `eval` et `exec`, voir la [documentation officielle](#).

Dans les deux cas, le code est exécuté dans le contexte de la pipeline et du nœud courant, ce qui signifie que l'environnement contient les objets suivants qu'il est possible d'utiliser:

- `node`: Corresponds à l'instance du nœud actuel.
- `pipeline`: Corresponds à l'instance de la pipeline contenant le nœud.
- `input`: Corresponds aux entrées du noeud, utilisables directement.
- `<nodeId>`: Instances de tous les nœuds de la pipeline actuelle, utilisables directement via leur identifiant.

9.7 Pipelines automatiques

Dans l'engine, tout ce qui peut être utilisé sous le point d'accès `/services` est une pipeline. De plus, comme décrit dans le chapitre 7, ceux-ci ne sont pas accessibles depuis l'extérieur, ils doivent donc être rendus disponibles via une pipeline créée automatiquement.

Lorsqu'un service démarre, ainsi que périodiquement, un service va s'annoncer auprès de l'engine en lui fournissant son point d'accès ainsi que son API. Celle-ci est décrite de la même manière que le champ `api` d'un nœud `entry`, ce qui permet de connaître la route à lui associer ainsi que les paramètres attendus. Une pipeline automatique est donc créée contenant un nœud `entry`, un nœud `service` et un nœud `end`.

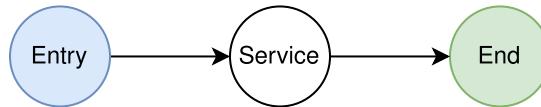


Figure 20: Pipeline automatique créée pour chaque service

10 Webapp

Il a été fixé dans le projet de manière optionnelle, qu'une application web sera développée et livrée. Les fonctionnalités n'ayant pas été strictement défini, il a tout de même été possible de la réaliser avec le minimum de fonctionnalités. L'ensemble de l'application a été développée avec React et ReactFlow.

La webapp qui a été développée permet à n'importe quelle catégorie d'utilisateurs de l'utiliser. La méthode de configuration de pipeline peut être très facilement représentée sous la forme d'un graphe. Il a été décidé que ces pipelines, pour une meilleures expérience utilisateur, seront représentées de manières visuelles sur l'application. Voici à quoi ressemble l'application.

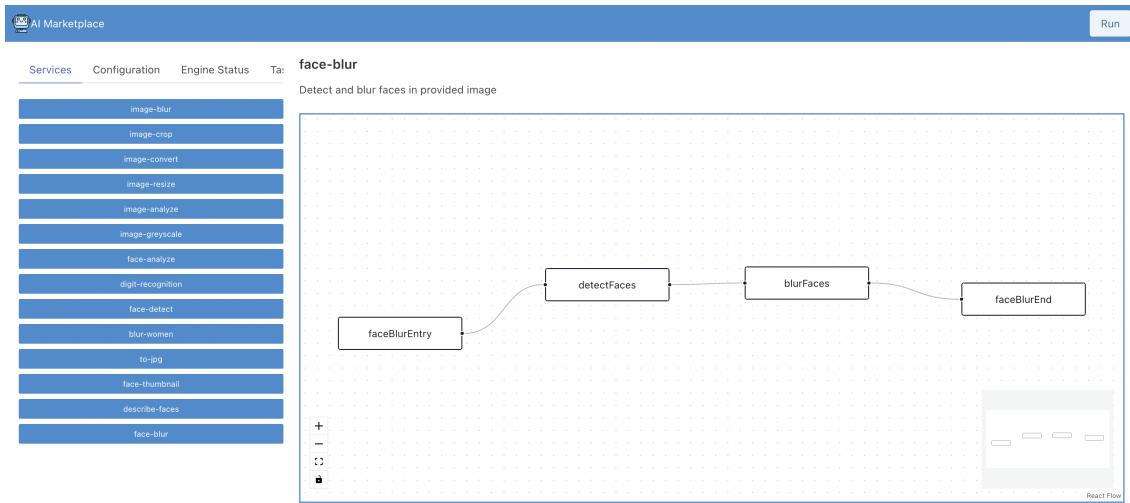
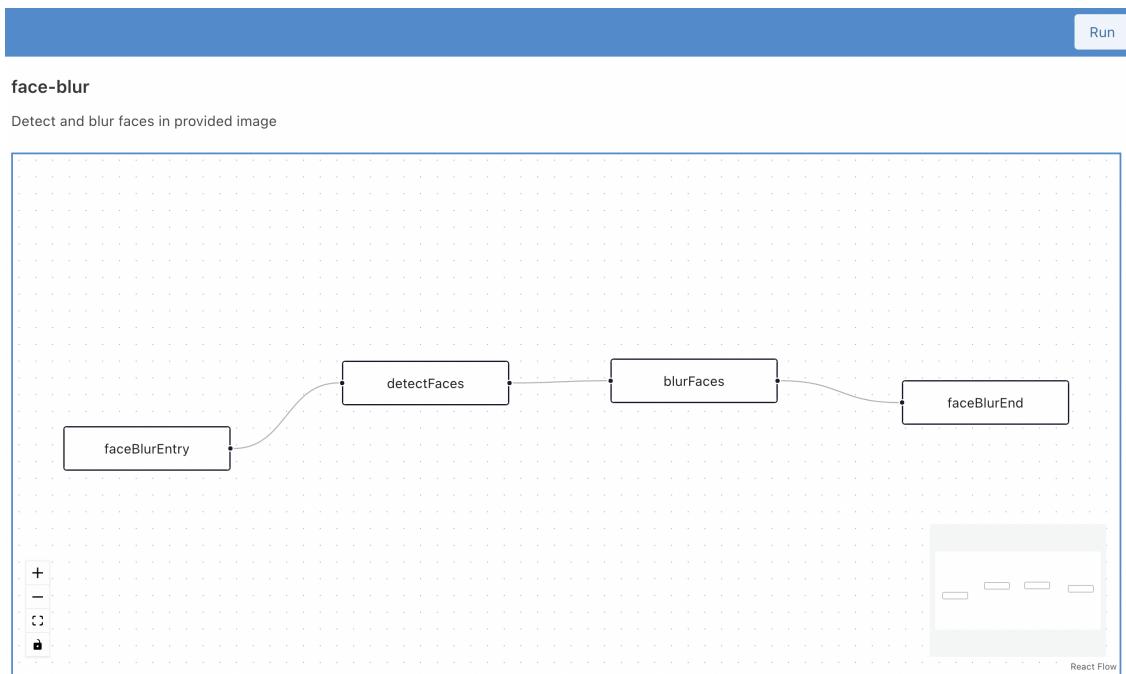


Figure 21: Impression écran de la webapp

L'application comprend en premier lieu une liste de l'ensemble des services disponibles et mis à disposition par l'engine (partie gauche de l'écran). La partie la plus importante est celle du milieu. Cette fonctionnalité permet de visualiser la pipeline sous la forme d'un graphe.

Figure 22: Visualisation de la pipeline *blurFace* sur la Webapp

En plus de la description et du titre, les utilisateurs peuvent très rapidement voir à quoi leur pipeline ressemble et comment chaque service ou noeud sont interconnectés.

Services
image-blur
image-crop
image-convert
image-resize
image-analyze
image-greyscale
face-analyze
digit-recognition
face-detect
blur-women
to-jpg
face-thumbnail
describe-faces
face-blur

Services
Configuration
Engine Status
Ta:

```

"root": {
  "nodes": [
    {
      "id": "faceBlurEntry",
      "type": "entry",
      "url": "http://service-face-detection.project-dev.svc.cluster.local/compute"
    },
    {
      "id": "detectFaces",
      "type": "service",
      "url": "http://service-face-detection.project-dev.svc.cluster.local/compute"
    },
    {
      "id": "blurFaces",
      "type": "service",
      "url": "http://service-face-detection.project-dev.svc.cluster.local/compute"
    }
  ],
  "links": [
    {
      "source": "faceBlurEntry",
      "target": "detectFaces"
    },
    {
      "source": "detectFaces",
      "target": "blurFaces"
    },
    {
      "source": "blurFaces",
      "target": "faceBlurEnd"
    }
  ]
}
  
```

(a) Liste des services disponibles dans l'engine
(b) Visualisation d'une configuration d'une pipeline

Figure 23: Fonctionnalités de la webapp pour les pipelines

Il est possible de sélectionner et visualiser une pipeline. Ces pipelines sont mises à disposition par l'engine. Tant d'un point de vue visuel que d'un point de vue plus technique, les utilisateurs ont la possibilité de voir le fichier de configuration au format JSON

Services Configuration Engine Status Ta:

Statistics of the engine
 Total : 0
 Running : 0
 Finished : 0
 Failed : 0

Statistics per services
 Image-blur : 0
 Image-crop : 0
 Image-convert : 0
 Image-resize : 0
 Image-analyze : 0
 Image-greyscale : 0
 Face-analyze : 0
 Digit-recognition : 0
 Face-detect : 0
 Blur-women : 0
 To-jpg : 0
 Face-thumbnail : 0
 Describe-faces : 0
 Face-blur : 0

Figure 24: Status de l'engine

Il est aussi possible de visualiser le status actuel de l'engine. L'ensemble des statistiques de chacun des services ainsi que de l'engine sont visionables dans cet onglet.

describe-faces

Describe the people in an image.

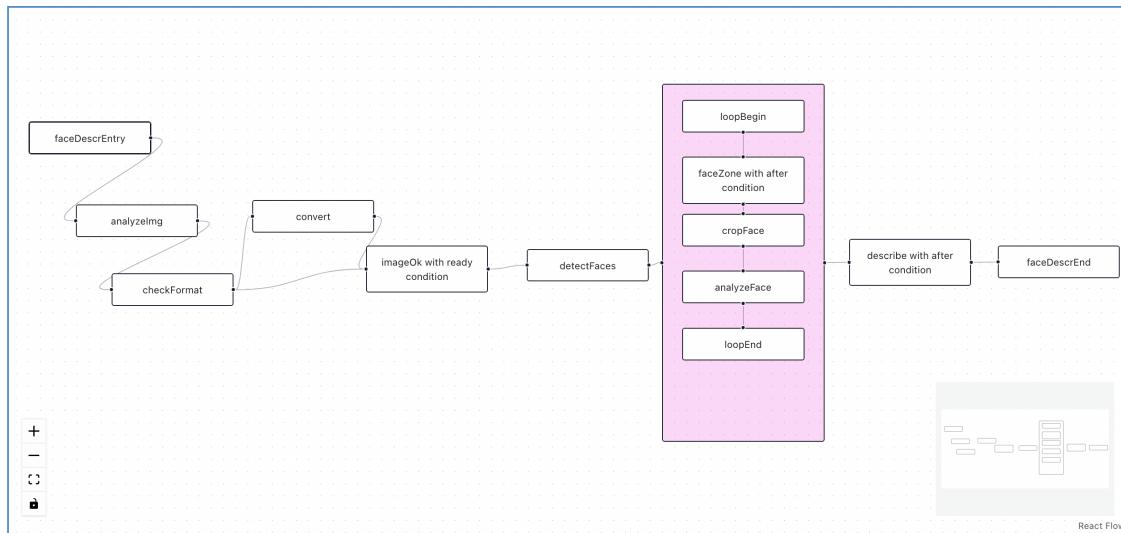


Figure 25: Visualisation d'une pipeline complexe

La webapp reconnaît et permet d'afficher les noeuds de type *loop* et *branch*. On peut y voir en rose un noeud *loop* qui contient un ensemble de des sous-noeuds qui combinés les un aux autres form un sous-graphe.

Cette application n'est malheureusement pas terminée, et la fonctionnalité permettant d'exécuter une pipeline n'est pas présente. Il est cependant assez facile de la rajouter grâce à la modularité des composants React.

11 Organisation

Il était souhaité par tous les membres de l'équipe que l'aspect organisationnel du projet soit le plus simple possible. Pour cette raison, la méthodologie Kanban a été adoptée. Kanban, contrairement à SCRUM, contient bien moins d'artefact et de cérémonies. Les aspects de modélisations et de développements ont pu être mis en avant de par la facilité organisationnelle qu'apporte Kanban. Cela a notamment permis de respecter les délais de livraisons du logiciel. Un jalon de livraison du *Minimum Viable Product* (MVP) (ou Produit minimum viable en français) a été placé au milieu du temps accordé pour la réalisation. Le jalon pour le MVP a été fixé à la semaine 8.



Figure 26: Planning prévisionnel

L'intégralité des fonctionnalités prévues ont pu être réalisées à l'exception de la partie MLOps. L'engine ainsi que trois pipelines fonctionnelle ont été livrée à temps dans un état fonctionnel. Le développement de la CI/CD ainsi que la mise en place de modèles de références pour le développement de services a permis de respecter les délais imposés pour la livraison du MVP. La partie MLOps a été retardée de par la complexité de déploiement des outils.

Les fonctionnalités de MLOps ont été repoussées pour la livraison finale du produit, ce qui est chose faites. Un simple modèle de reconnaissance de chiffres réentrantable est aujourd'hui disponible dans notre application. Le réentrainement est fait de manière automatique lorsque un commit ainsi qu'une demande de fusion sont faites. Le nouveau modèle est ensuite automatiquement déployé dans le service auquel il appartient.

11.1 Séparation des tâches

La séparation des tâches entre les membres de l'équipe a permis de réaliser et livré le projet dans les temps impartis. Les tâches ont été séparées de la sorte :

Fanetti Simon :

- Responsable de la mise en place de la partie MLOps.
- Réalisation de quelques services de machine learning.

Paschoud Nicolas

- Responsable, en collaboration avec Wolleb Benoist, de la modélisation du système.
- Responsable de l'organisation du groupe
- Modélisation et création de la webapp

Srdjenovic Luca

- Responsable de la mise en place d'une pipeline de CI/CD
- Responsable de la mise en place du déploiement de l'application sur Kubernetes

- Réalisation de tests des services

Wolleb Benoist

- Responsable, en collaboration avec Paschoud Nicolas, de la modélisation du système.
- Responsable de la modélisation et réalisation de l'engine

De plus, la seconde moitié du projet a permis à certains membres de travailler sur d'autres parties que celles attribuées. Notre approche Kanban nous a permis l'agilité des membres et l'utilisation des issues gitlab a permis à chacun de contribuer au projet.

L'intégralité de la réalisation de ce projet a été réalisée en bonne partie par binôme. Les intérêts et besoin de chaque membre sur leur partie respective a fortement poussé à la collaboration. La collaboration, l'entre-aide et le respect entre tous les membres a contribué à instaurer une ambiance de travail chaleureuse et respectable ce qui a grandement contribué à la réussite de ce projet.

12 Conclusion

Le dépôt contenant les sources principales de notre projet se trouve à l'adresse:

https://gitlab.forge.hefr.ch/mse_pi/services

Le dépôt contenant les ressources liées au MLOps se trouve à l'adresse:

https://gitlab.forge.hefr.ch/mse_pi/mlops

Le système déployé sur le *cluster* de la HEIA est accessible via l'interface de l'engine et de l'application web:

<https://pi-engine.kube.isc.heia-fr.ch/docs> <https://pi-webapp.kube.isc.heia-fr.ch>

12.1 État actuel

Tel que rendu à l'issue de ce projet, le système est capable de:

- Faire fonctionner un écosystème cohérent et facilement extensible de microservices unitaires.
- Exécuter des opérations simples en utilisant les fonctionnalités offertes par chaque microservice, lesquels utilisent des technologies de *machine learning* ou non.
- Construire, tester et déployer les services, y compris l'engine, l'interface web et la base de données, de manière automatique.
- Permettre la définition de pipelines capables d'utiliser plusieurs services du système afin de créer des fonctionnalités riches.
- Permettre l'utilisation des pipelines définies, la récupération des résultats et leur nettoyage.
- Gérer et faire transiter des données simples (typées, serialisées en json) et plus complexes (images ou autres).
- Ré-entraîner de manière automatique un modèle de machine learning qui sera utilisable par un des microservices du projet.
- Offrir une interface web rendant la visualisation et l'utilisation globale du système plus agréable pour un utilisateur *lambda*.

Ces points correspondent à ce que nous avions prévu dans le cahier des charges, dès lors nous considérons que le projet a été réalisé avec succès.

12.2 Critique

Le système que nous proposons n'est évidemment pas parfait et plusieurs problèmes et limitations sont connus:

- Comme évoqué à la section 8.4 il n'y a pas de notion de compte utilisateur, n'importe qui peut utiliser, créer ou supprimer une pipeline. Cela ne pose *a priori* pas de problème concernant la confidentialité des données, mais rend tout de même le système vulnérable aux utilisateurs mal intentionnés.
- Comme décrit dans le fonctionnement des pipelines, il est possible d'exécuter du code python dynamiquement lors de l'exécution d'un nœud. Cela permet une grande flexibilité mais demeure un problème important. En effet, le code exécuté n'est pas vérifié ni encapsulé, ce qui pourrait poser de sérieux problèmes de performances et sécurité, combiné qui plus est à l'absence de notion de compte utilisateur, n'importe qui peut injecter du code python qui sera exécuté à la volée.
- Même si une certaine portion du code est couverte, l'engine manque encore d'une gestion plus fine des erreurs qui peuvent survenir durant l'exécution d'une pipeline. Il est notamment très difficile de déboguer une pipeline lorsque l'on essaie d'en créer une nouvelle.
- Comme expliqué à la section 8.7, l'engine ne peut, en l'état, pas être mis en redondance, empêchant ainsi de faire de la répartition de charge. Étant le seul point d'accès au système depuis l'extérieur, cela en fait un point de défaillance unique (*single point of failure*) du système. On peut en effet se douter que lors de pics d'utilisation, l'engine puisse devenir inaccessible.

- L'utilisation de nœuds de type **service** est découplée du contrôle de la disponibilité de ceux-ci. En effet, comme décrit à la section 7.5, l'engine va automatiquement supprimer l'accès aux services qui ne s'annoncent plus. Ceci est fait pour les pipelines automatiques construites pour ce service, mais pas pour les nœuds de type **service** des autres pipelines.
- Comme décrit dans le chapitre 7, si l'utilisation de HTTP comme protocole rend notre système facilement interopérable, il demeure néanmoins un problème pour définir des RPC et nous avons tout de même l'impression de tordre de protocole pour une utilisation qui n'est pas optimale.
- Le mécanisme de relance présent dans le nœud de service (section 9.3.4) n'est pour l'instant utilisé que si la requête échoue immédiatement, à cause d'une connexion échouée ou d'un code d'erreur HTTP. Or, rien n'est implémenté si la requête est bien effectuée, mais que le service tombe pendant que la tâche lui est assignée. C'est un ajout nécessaire afin de gérer plus de cas d'interruption de service.
- Certains types de nœuds responsables d'opérations complexes, comme le nœud conditionnel ou le nœud d'itération, ne sont pas testés de manière approfondie et il est probable qu'il subsiste des bugs qui pourraient survenir lors de l'exécution (par exemple l'imbrication).
- Il subsiste quelques instabilités lorsque l'engine reste en fonctionnement durant une longue période, particulièrement en ce qui concerne la connexion au service de stockage objet qui semble tomber d'elle-même au bout d'un certain temps. Un mécanisme de reconnexion, voire d'auto-diagnostic de l'engine devrait être mis en place pour assurer une plus grande fiabilité.

13 Annexes

13.1 Portfolio

13.1.1 Luca Srdjenovic

Pour ce projet, il fallait se répartir les tâches entre les 4 membres de l'équipe dont je fais partie. Pour ma part, j'ai décidé de m'orienter dans la partie *DevOps CI/CD* du projet, car depuis le Bachelor, c'est un groupe de compétence qui m'a toujours intéressé à maîtriser, mais que j'ai eu peu d'occasions à entraîner. J'ai décidé de mettre de côté au final la partie *Dev* du projet afin de m'orienter plus sur la partie *Ops*. En effet, travaillant à côté du Master, je fais déjà principalement du *Dev*, mais beaucoup moins d'*Ops* voir pas du tout. C'est une bonne occasion de gagner en expérience ici, car dans un projet à plusieurs personnes (même à une seule personne), selon moi, il est crucial de mettre en place dès le début d'un projet, une infrastructure qui va nous permettre de gagner le plus possible du temps pour éviter d'en perdre inutilement plus tard (expérience déjà vécue dans d'autres projets).

Comme toute première tâche, je devais prendre en main l'outil Kubernetes avec Docker qui allait nous permettre de faire la partie **Continous Deployment (CD)** du *DevOps*. C'était quelque chose de tout nouveau pour moi le *CD* et l'utilisation de l'outil Kubernetes. C'est toujours délicat de prendre en main de nouvelles technologies, car on ignore le potentiel de ces dernières. Par conséquent, je me suis entraîné en créant un projet de test `example-fastapi` sur *Gitlab* pour apprendre à utiliser ces nouveaux outils. De là, j'ai même appris que *Gitlab* offrait beaucoup plus de potentiel que ce que j'utilisais, qui était principalement que du **Continuous Integration (CI)**.

En effet via *Gitlab*, j'ai appris que nous pouvions créer des images Docker et les stocker dans son *Registry Container* (une sorte de dockerhub, mais pour notre projet *gitlab*). J'ai même appris que nous pouvions utiliser différents jetons pour nous authentifier à notre projet (e.g. pour les secrets sur Kubernetes) offrant une sécurité supplémentaire ainsi que la création de nos propres variables pour les *pipelines CI/CD*. Une fois la prise en main de kubernetes accomplie, j'ai pu bien comprendre comment il fonctionnait en tout cas pour nos besoins en terme de déploiement de Pods, de services et d'*ingresses* (routes).

A partir de là, j'ai pu passer cette fois à notre projet de services, et j'ai conceptualisé les pipelines *CI/CD* sur *Gitlab*. C'était un peu délicat à maîtriser, car certaines fois, un job qui devait fonctionner ne fonctionnait pas (erreur de téléchargement d'un paquet par exemple). J'ai perdu beaucoup de temps au début à chercher à comprendre et gérer ces erreurs inattendues en retestant mes pipelines. C'était long surtout quand nous devions reconstruire des images qui prenaient 4 GiB de taille mémoire et qu'elles s'interrompaient à 95 % du job.

Une fois la *pipeline* finie dans les temps en se basant sur notre calendrier, j'ai pu me réorienter pour aider mes camarades dans leurs tâches respectives. En effet, m'occupant du *CI/CD*, j'ai entrepris les tests, donc j'ai implémenté des services et leurs tests unitaires comme le traitement d'image, de détection/analyse de visages. Après cela, j'ai pu les ajouter à notre pipeline *CI/CD* pour les tester de manière automatique. Cette phase-là était importante, car cela nous permet d'éviter de faire des erreurs humaines en testant les services à la main. Au final, j'ai aussi touché à du *Dev* ce qui était gratifiant, car au début, je ne savais pas jusqu'où est-ce que j'allais aller. J'ai pu me rendre assez polyvalent dans le projet, en faisant du traitement d'image en python, en touchant aussi à de l'IA.

Pour résumer, la répartition des tâches a été bien acceptée par tous les membres de l'équipe. Nous avons pu tous nous en sortir dans nos tâches. La fusion finale de nos travaux a réussi à produire un résultat plutôt convaincant qui a respecté notre cahier des charges. Je suis très satisfait du résultat pour avoir travaillé avec des gens dont je ne connaissais pas vraiment leur compétence. Chaque membre avait sa propre partie qui différait avec celle des autres membres. Nous avons pu tous nous synchroniser grâce à mon travail. Ce fut intéressant et réjouissant pour l'avenir.

Comme derniers mots, je souhaiterais dire que j'ai appris durant ce projet de nouvelles compétences que je n'avais pas avant. En effet, j'ai découvert l'architecture micro-service en plus de celle monolithique. Cela me sera d'une grande aide dans mon travail, car je serai amené à devoir en implémenter. Je serai toujours en train d'apprendre dans le futur, mais à partir de maintenant, quand j'entreprendrai un nouveau projet, je pourrai par cette expérience mieux appréhender mon développement logiciel.

Luca Srdjenovic

13.1.2 Benoist Wolleb

Ma première et principale tâche était d'implémenter l'engine et de proposer une solution permettant de créer des pipelines qui utilisent les microservices de notre système. Je me suis beaucoup documenté sur le style architectural *Pipe and filter* et le paradigme *Flow Based Programming* qui me semblait être la bonne voie, notamment au travers de technologies comme [Node-RED](#) qui ont directement inspiré le design de la solution actuelle. Je pense au final avoir réussi à proposer une solution convaincante.

La gestion des tâches dans l'engine a été un très bon exercice, notamment avec les choix architecturaux que nous avons faits: fonctionnement de l'engine sans état, fin de tâches basées sur des *callbacks* et programmation asynchrone.

Je ne connaissais pas du tout le module FastAPI et j'ai été très agréablement surpris de sa simplicité d'utilisation et de sa flexibilité. J'avais déjà conçu quelques interfaces en python avec Django et Flask, mais si c'était à refaire, je choisirai FastAPI. J'ai aussi été conquis par la documentation OpenAPI automatique que FastAPI génère et l'interface web Swagger qui l'accompagne, y compris pour les points d'accès dynamiques. Concernant ceux-ci, je suis allé assez loin dans les rouages de FastAPI pour que cela fonctionne comme nous le voulions.

Conséquence de l'utilisation de FastAPI, notre implémentation, autant dans les services que dans l'engine, est très orientée asynchrone. Nous avons en effet été naturellement poussés à utiliser beaucoup de coroutines et à pleinement expérimenter le moteur asynchrone `asyncio` de python. Aspect assez inattendu du projet, c'était un très bon exercice, les coroutines étant de plus en plus "à la mode" avec le succès de Node.js et même leur spécification dans C++²⁰.

J'ai la chance d'avoir plusieurs années d'expérience dans un environnement de travail dynamique où des technologies similaires sont utilisées. Je suis par conséquent très familier avec les concepts d'intégration continue, déploiement continu, microservices et interfaces REST, c'est aussi la raison qui m'a poussé à laisser ces aspects à mes collègues.

Enfin, concernant la gestion de mon travail dans un projet de groupe, le choix de travailler sur l'engine allait de manière prévisible me lancer sur la partie du projet nécessitant le plus d'implémentation, et je pense avoir clairement sous-estimé le travail que ça m'a finalement demandé. Avec une très bonne maîtrise de `git`, j'ai aussi, je pense, été le garant de la bonne gestion des sources dans notre dépôt, et j'ai pu assister mes collègues lors de gros changements.

De manière générale, je pense nous avons réussi à faire une répartition des tâches efficace au sein du projet, car chacun a su trouver un domaine peu maîtrisé où il a pu approfondir ses connaissances, et nous avons réussi à fournir une solution fonctionnelle et intéressante.

Benoist Wolleb

13.1.3 Nicolas Paschoud

Mon premier objectif était, en collaboration avec Benoist, de modéliser l'architecture de notre système. Nous avons pu après plusieurs longues discussions trouver une solution satisfaisante. Nos connaissances respectives ont permis la modélisation du système en imaginant un grand nombre d'architectures au style très varié. De part mon fort intérêt pour les *event-driven systems*, j'ai pu guider la modélisation du système lorsque nous avons eu l'idée d'utiliser des *message queue*. Une des variantes du système avait pour objectif de produire un ensemble de messages au sein d'un cluster afin que les services concernés puissent faire le traitement. Cette version a malheureusement vite été abandonnée de la part complexité d'un tel système. Nous avons donc décidé après plusieurs discussions de revenir sur une version asynchrone basée sur les microservices. Ces longues discussions m'ont permis de mettre à pleine contribution toutes mes connaissances des systèmes et différentes applications que j'ai pu voir tout au long de mes études.

Mes connaissances de FastAPI m'ont permis d'assister mes collègues dans leur phase d'apprentissage. Bien qu'ayant peu d'expérience professionnelle, cela m'a tout de même permis de diriger certaines décisions dans la bonne direction. Notamment sur le sujet du DevOps et des bonnes pratiques de déploiement. J'ai très rapidement permis à l'équipe de se mettre au travail en mettant à disposition mes templates *Hello Worlds* de FastAPI. Ceux-ci contenaient de simples exemples et était *production ready*. Ce template a été légèrement modifié pour correspondre à nos besoins. Celui-ci se trouve dans tous nos services ce qui facilite fortement la maintenance et l'interopérabilité des membres. Le template étant connu, travailler sur le service que quelqu'un d'autre a développé est très facile.

J'ai au cours des dernières années développé un certain dégoût des très populaires frameworks de développement web d'aujourd'hui. Ce projet m'a tout de même permis de passer outre ces idées, et de prendre à bras ouvert le framework React pour la réalisation de notre webapp. N'étant pas un fan du tout de Node.js ainsi que leur "package manager", il m'a fallu m'y faire. Cela est une très bonne expérience car les contraintes techniques tel que celle-ci font partie du monde professionnel.

Je retiens une très bonne expérience de ce projet, autant dans le sujet lui-même que la collaboration avec mes collègues. La collaboration et la bonne volonté que chaque membre a émises durant ces 14 semaines ont fortement contribué à notre réussite. Je suis personnellement satisfait du travail accompli autant techniquement qu'humainement. Ce fut un plaisir de travailler avec ces messieurs !

Nicolas Paschoud

13.1.4 Simon Fanetti

Dans ce projet, j'ai décidé de m'occuper de la partie MLOps car je n'avais pas vraiment d'expérience dans le domaine DevOps. Ayant suivi le cours de machine learning, cette nouvelle dimension me permettait de mieux consolider mes connaissances de ML avec tous les aspects de traque de performances et de versions dans un cadre concret comme celui-ci.

Au sein du groupe, j'ai donc décidé de m'occuper de cette partie, ainsi que des services ML et de traitement d'images que l'engine devra orchestrer. Pour implémenter les services, j'ai pu me familiariser avec les objets multi-parts au travers de FastAPI, ainsi que de m'assurer des dépendances des services dans leur environnement Docker pour la partie CI/CD. Malgré, le peu d'interaction que j'ai pu avoir avec le CI/CD lors du développement des services, j'ai pu y goûter une fois le service de MLOps entamé grâce au pipeline CI permettant d'appliquer les principes MLOps avec des outils tels que DVC, ainsi qu'interagir avec le serveur Minio S3. Cette notion ainsi que les pipelines CI m'ont permis de réaliser concrètement les notions DevOps vues comme dans les "The Twelve-Factor App" tels que les artefacts ainsi que l'aspect sécurité lié aux credentials dans un repo.

Au final, je pense avoir intégré les concepts MLOps, même si la notion de CD reste flou. Je retiens de cette expérience de groupe un très bon travail d'équipe, avec des camarades souvent disponibles pour discuter de l'évolution du projet et s'assurer d'avancer dans la même direction.

Simon Fanetti