 **2024-2025**

**Rapport projet annuel**

**Garbage.ia**

**Sous la direction de : Par**

Mr. Wajnberb & DEHAESE Romain &

Ms. Balasoupramaniane EL HARRAR Ethan

**Table des Matières**

1. Mise en place de l’environnement frontend
2. Intégration de l’interface pédagogique
3. Connexion au modèle de détection YOLOv8
4. Préparation du dataset pour le réentraînement du modèle YOLOv8
5. Création d’une API pour streamer le flux vidéo
6. Intégration du flux dans le frontend
7. Préparation de la suite du projet
8. Clics précis et interactions intelligentes avec YOLOv8
9. Capture d'image et stockage structuré des corrections
10. Recadrage précis des corrections et robustesse des retours utilisateur
11. Réentraînement automatique du modèle à partir des retours utilisateurs
12. Intégration cloud, déploiement et limitations rencontrées

**1. Mise en place de l’environnement frontend**

**Objectif :**

Créer une interface utilisateur avec une grille de catégories de déchets, stylisée et interactive, destinée à un usage pédagogique (enfants).

Étapes :

* Création d’un projet React + Vite avec TypeScript.
* Intégration de TailwindCSS pour la mise en forme rapide et responsive.

**Problèmes rencontrés :**

* Tailwind n’était initialement pas installé ou mal configuré (absence de fichiers tailwind.config.js, postcss.config.js, ou directives manquantes dans index.css).
* Le CSS ne s’appliquait pas car les fichiers n’étaient pas reconnus par Vite ou mal liés.
* Erreurs récurrentes dues à une mauvaise initialisation avec npx tailwindcss init -p causées par un problème de permissions ou de versions incompatibles de node, npm, ou bun.

**Résolution :**

* Repartir d’un projet frontend-ready fonctionnel.
* Déplacer les anciens composants et ajuster l’architecture.
* Corriger la configuration de Tailwind et l’initialisation Vite.
* Vérification que le style était effectivement appliqué.

**2. Intégration de l’interface pédagogique**

**Objectif :**

Proposer une interface simple pour afficher des catégories de déchets cliquables.

**Fonctionnalités ajoutées :**

* Liste de catégories avec émojis.
* Affichage de descriptions sous les catégories.
* Composant CorrectionDialog pour permettre à l’utilisateur de reclassifier un déchet.
* Déclenchement du CorrectionDialog lors du clic sur un élément.
* Système de lecture de paramètres d’URL pour ouverture conditionnelle de la boîte de dialogue.

**3. Connexion au modèle de détection YOLOv8**

**Objectif :**

Exploiter un modèle YOLOv8 (segmentation) pour détecter des objets en temps réel via la caméra.

**Étapes :**

* Fichier detect.py mis en place pour tester la détection en local.
* Utilisation de cv2.VideoCapture() pour accéder à la caméra.
* Visualisation de la sortie via OpenCV.
* Détection réussie avec la caméra de l’iPhone (via Camo), mais il a fallu tester plusieurs index (0, 1, etc.).

**Problèmes :**

* OpenCV n’ouvrait pas la bonne caméra selon les machines (Webcam interne vs Camo).
* Les logs générés par YOLOv8 prenaient énormément de place dans le terminal, rendant le contrôle difficile.

**Résolution :**

* Ajout d’une fonction list\_cameras pour tester les indices de caméra disponibles.
* Ajout d’un système de fermeture du flux par touche q.
* Tentative de suppression des logs via redirection de stdout, mais YOLOv8 écrit directement dans la sortie système.
* Solution partielle : laisser les logs pour le moment et prioriser la stabilité de l’affichage.

**4. Préparation du dataset pour le réentraînement du modèle YOLOv8**

1. **Importer le dataset brut**

Dans une première étape, l’ensemble d’images de départ se trouve dans l’environnement de développement local (PyCharm). Les images ont été rangées dans des sous-dossiers, chaque dossier correspondant à une catégorie de déchets (ex : plastique, métal, verre, organique…).

Cette architecture hiérarchique aide à une lecture initiale de la nature du dataset.

On peut ainsi se faire une première idée sur la composition des classes dans l’ensemble du dataset avant traitements.

1. **Découpage en sous-ensembles : folder\_sort.py**

Pour respecter les bonnes pratiques de l’apprentissage supervisé, un script Python nommé folder\_sort.py a été développé.

Il automatise la répartition du dataset brut en trois sous-ensembles :

* Train (70%)
* Validation (20%)
* Test (10%)

Cette découpe aléatoire mais reproductible garantit que chaque catégorie conserve la même proportion d’images dans chacun des sous-ensembles, évitant ainsi tout biais et assurant une évaluation robuste du modèle.

1. **Reformatage du dataset : flatten\_yolo\_dataset.py**

YOLOv8 attend une arborescence plate où chaque image est associée à un fichier d’annotation .txt portant le même nom. Le script flatten\_yolo\_dataset.py réalise les opérations suivantes :

* Parcours des dossiers train/, val/ et test/ générés précédemment.
* Copie de chaque image directement dans trois nouveaux répertoires racines (train, val, test) sans sous-dossiers.
* Initialisation d’emplacements vides pour les fichiers d’annotations à créer ou importer.

*Limitation rencontré : À ce stade, seul le format d’arborescence est correct, mais les fichiers .txt d’annotations sont manquants. YOLOv8 ne peut pas s’entraîner sans ces annotations.*

1. **Extraction d’un sous-ensemble annotable : extract\_balanced\_subset.py**

Pour réduire la charge d’annotation manuelle et garantir une bonne représentation de chaque classe, nous avons créé le script extract\_balanced\_subset.py :

* Il sélectionne aléatoirement 40 images par catégorie parmi les dossiers train/ et val/.
* Le tirage est réalisé de manière à maintenir un équilibre strict entre les classes, évitant qu’une classe majoritaire monopolise l’effort d’annotation.

Ce sous-ensemble restreint servira de noyau pour l’annotation manuelle initiale, permettant de créer un modèle de base à partir duquel automatiser le reste du dataset.

1. **Annotation manuelle via Roboflow**

Nous avons choisi Roboflow pour sa simplicité et son interface intuitive :

* Création d’un projet dédié dans Roboflow.
* Importation du sous-ensemble de 40× N images.
* Utilisation de l’outil de dessin pour tracer des boîtes englobantes (bounding boxes) autour de chaque déchet, avec attribution de la classe correspondante.

Cette étape assure la qualité des annotations, condition sine qua non pour un bon apprentissage du modèle.

Une image contenant texte, capture d’écran, Site web, logiciel

Le contenu généré par l’IA peut être incorrect.

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Le contenu généré par l’IA peut être incorrect.

1. **Export du dataset annoté dans le projet**

Une fois l’annotation terminée, Roboflow a permi d’exporter le jeu de données au format YOLO :

* Fichiers images et fichiers .txt d’annotations placés dans la même arborescence flat (train/, val/).
* Chaque .txt contient, par ligne, la classe et les coordonnées normalisées de la bounding box.

Ces fichiers sont ensuite mis dans le répertoire du projet pour être directement utilisables par YOLOv8.

**Une image contenant texte, capture d’écran, Police, conception

Le contenu généré par l’IA peut être incorrect.**

1. **Entraînement d’un modèle initial : train\_small\_model.py**

Pour disposer d’un outil d’annotation automatique, un premier modèle léger a été entraîné :

* Le script train\_small\_model.py charge le dataset annoté manuellement.
* Paramétrage d’un réseau YOLOv8 « nano » ou « small » (nombre réduit de couches et de filtres).
* Entraînement sur un nombre limité d’époques pour obtenir un modèle fonctionnel, sans viser la performance maximale.

Ce modèle sert de base pour générer des annotations automatiques sur les images non annotées.

1. **Annotation automatique du dataset complet : auto\_annotate\_dataset.py**

Enfin, le script auto\_annotate\_dataset.py permet d’étendre les annotations :

* Chargement du modèle YOLOv8 préalablement entraîné.
* Parcours de tous les fichiers images restants dans train/ et val/.
* Prédiction des bounding boxes et des classes pour chaque image.
* Génération automatique des fichiers .txt d’annotations correspondants.

Ainsi, on obtient un dataset complet, prêt pour un entraînement final à grande échelle, sans avoir eu à annoter manuellement chaque image.

1. **Entraînement final du modèle et sauvegarde : train.py**

* Chargement du dataset complet annoté (train/ et val/).
* Lancement du script train.py qui :
* Initialise un réseau YOLOv8 (par exemple « small » ou « medium » selon les besoins).
* Charge les hyperparamètres définitifs (learning rate, batch size, nombre d’époques, etc.).
* Entraîne le modèle sur l’ensemble des images annotées.  
  À la fin de l’entraînement, le meilleur modèle (celui qui obtient la plus haute mAP sur l’ensemble de validation) est automatiquement sauvegardé sous le fichier :

final\_model/best.pt

Ce checkpoint best.pt pourra ensuite être déployé pour l’inférence ou servir de point de départ à un fine-tuning futur.

**Voici les métriques récupérées après le réentraînement du modèle :**

**mAP50–95 (mask)**

Avec une valeur finale de 0.62, cette métrique reflète la qualité globale et la précision spatiale fine des masques sur toute la plage d’IoU (0.50–0.95).

**Une image contenant texte, capture d’écran, Police, diagramme

Le contenu généré par l’IA peut être incorrect.**

**mAP50 (mask)**

À 0,73, ça indique que dans 73 % des cas, le masque couvre au moins la moitié de l’objet, ce qui correspond au critère minimal de recouvrement souvent requis pour garantir une détection opérationnelle fiable.

Une image contenant texte, capture d’écran, diagramme, ligne

Le contenu généré par l’IA peut être incorrect.

**Précision (mask)**

Avec 0.80, le modèle génère très peu de “faux positifs” dans la segmentation, garantissant des contours fiables.

Une image contenant texte, capture d’écran, diagramme, Police

Le contenu généré par l’IA peut être incorrect.

**Rappel (mask)**

À 0.65, il témoigne du bon taux de couverture des objets réels, limitant les “faux négatifs” au sein des masques.

Une image contenant texte, capture d’écran, ligne, Police

Le contenu généré par l’IA peut être incorrect.

Une image contenant Turquoise, bleu vert, Bleu sarcelle, mosaïque

Le contenu généré par l’IA peut être incorrect.

Une image contenant rose, intérieur, salle de bain, violet

Le contenu généré par l’IA peut être incorrect.

**5. Création d’une API pour streamer le flux vidéo**

**Objectif :**

Envoyer le flux traité par YOLOv8 au frontend via une API FastAPI.

**Étapes :**

* Création de api.py avec FastAPI et StreamingResponse.
* Configuration d’un endpoint /video\_feed pour renvoyer un flux MJPEG.
* Mise en place d’un serveur via uvicorn.

**Problèmes rencontrés :**

* Mauvais nom de module dans le lancement (utils.api:app au lieu de api:app).
* Problèmes de permissions sur certains fichiers \_\_init\_\_.py.

**Résolution :**

* Réorganisation du backend et bon lancement via uvicorn api:app --reload.

**6. Intégration du flux dans le frontend**

**Objectif :**

Afficher le flux caméra dans l’interface principale avec un titre et la grille de catégories en dessous.

**Étapes :**

* Intégration d’une balise img avec source http://localhost:8000/video\_feed.
* Mise en page avec TailwindCSS pour un rendu soigné.
* Ajout d’un titre explicite : “Déposez votre plateau.”
* Gestion du clic sur les catégories.

**7. Préparation de la suite du projet**

**Objectif final :**

Permettre à l’utilisateur de cliquer sur un déchet détecté sur le flux, ouvrir une boîte de correction, reclassifier le déchet, et stocker l’information pour réentraînement du modèle.

**8. Clics précis et interactions intelligentes avec YOLOv8**

**Objectif :**  
Améliorer la précision de l'interaction utilisateur en permettant de cliquer directement sur les cadres détectés par YOLOv8 dans le flux vidéo.

**Problèmes rencontrés :**

* Le flux vidéo affichait les cadres dessinés par YOLOv8, mais ceux-ci n’étaient pas interactifs (inclus dans l’image, donc non cliquables).
* Un clic sur le flux ne permettait pas de savoir précisément quel déchet avait été mal détecté.
* Les cadres dessinés côté frontend ne correspondaient pas exactement aux dimensions du flux affiché, ce qui rendait les clics imprécis.
* L’affichage de tous les cadres rendait l’interface illisible lorsque plusieurs objets étaient détectés.

**Solutions mises en place :**

* Récupération des coordonnées des bounding boxes via l'endpoint /detections côté backend.
* Superposition d'un canvas HTML transparent sur le flux vidéo.
* Application d’un facteur d’échelle pour corriger l’alignement des cadres avec l’image redimensionnée.
* Détection des clics uniquement à l’intérieur des zones délimitées par les bounding boxes.
* Réduction de la fréquence de rafraîchissement des détections à 300 ms pour améliorer la fluidité.
* Suppression de l'affichage visuel des cadres tout en conservant les zones de clics actives (cadres invisibles).

**9. Capture d'image et stockage structuré des corrections**

**Objectif :**  
Permettre à l’utilisateur de corriger une mauvaise catégorisation et stocker cette correction pour améliorer le modèle.

**Problèmes rencontrés :**

* Il n’était pas clair à quel moment capturer l’image pour l’associer à la correction.
* Des captures étaient parfois déclenchées au mauvais moment (ex. : dès le clic, même si l’utilisateur annulait ensuite).
* Le lien entre la capture et les corrections n'était pas stocké dans une structure exploitable.

**Fonctionnement mis en place :**

* Lorsqu’un utilisateur clique sur un objet, le composant CorrectionDialog s’affiche.
* Étape 1 : sélection de la catégorie erronée détectée par le modèle.
* Étape 2 : sélection de la catégorie correcte par l’utilisateur.
* La capture de l’image de la caméra n’est déclenchée **qu’après validation finale**.
* Envoi des données au backend via une requête POST avec la capture et les catégories.
* Enregistrement dans un fichier image + ligne CSV contenant : timestamp, nom du fichier image, mauvaise catégorie, catégorie corrigée.
* Vérification que l’image est bien écrite et le CSV mis à jour.

Se pose maintenant une question crusiale dans le fait de prendre les feedback des utilisateurs :

**Peut-on faire confiance à tous les retours utilisateurs ?**

Non. Et ce n’est pas une question de mauvaise foi (surtout pas avec des enfants), mais de contexte. En effet, les enfants peuvent se tromper, cliquer vite, mal comprendre la catégorie.

Le retour utilisateur doit être considéré comme une *source bruyante*, pas une vérité absolue.

**Donc peut-on savoir si le modèle faisait réellement une erreur ?**

On peut estimer la confiance du modèle au moment de la détection, mais pas directement son "accuracy".

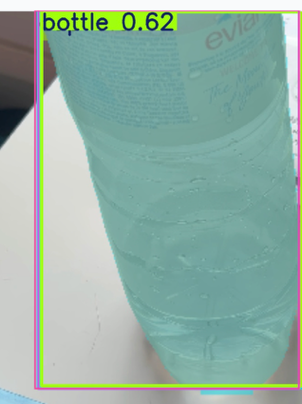
Exemple de log YOLOv8 :

0: 384x640 1 bottle, 53.3ms

Speed: 1.4ms preprocess, 53.3ms inference, 1.0ms postprocess per image at shape (1, 3, 384, 640)

Ce log ne contient pas la confiance (score de probabilité), mais YOLOv8 peut la renvoyer facilement via son API Python.

Exemple :

 Ici on récupèrerait dans l’idée « 0,62 »

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Condition** | **Action** | | conf > 0.75 + correction | Enregistrer mais avec un **flag downweight** | | conf < 0.4 + correction | Donner **plus de poids** à la correction | | conf > 0.9 + correction | Possible **flag d'anomalie / bruit** | |  |

Ainsi, on a une sorte de système de pondération dynamique du feedback utilisateur, qui protège ton modèle contre l’oubli de ce qu’il sait déjà.

**10. Recadrage précis des corrections et robustesse des retours utilisateur**

**Objectif :**  
Améliorer la fiabilité des retours utilisateurs en enregistrant non plus l’image complète de la caméra, mais **seulement le déchet concerné** au moment de la correction, pour permettre un réentraînement plus précis du modèle.

**Problèmes rencontrés :**

* Le système initial capturait **toute l’image caméra**, ce qui était imprécis en cas de détections multiples.
* L’identifiant (detection\_id) utilisé pour relier une correction à une détection devenait **vite invalide** : les détections étaient mises à jour en continu côté backend et l’ID disparaissait avant validation par l’utilisateur.
* Le lien entre la détection sélectionnée dans l’interface et l’image capturée était donc **fragile** et provoquait des erreurs côté backend ("ID de détection introuvable").

**Fonctionnement mis en place :**

* Côté frontend, lorsqu’un utilisateur clique sur un objet détecté, l’objet (Detection) complet est mémorisé dans le composant React.
* Ce Detection contient : id, label, bbox, score, image\_width, image\_height.
* Lors de la soumission de correction via le CorrectionDialog, toute la détection est envoyée au backend, pas seulement l’ID.

**Exemple :**

*{*

*"wrong": "plastic",*

*"corrected": "bio",*

*"detection": {*

*"id": "...",*

*"label": "plastic",*

***"bbox": [x, y, w, h],***

*"score": 0.62,*

*...*

*}*

*}*

* Le backend utilise directement la bbox pour **croper l’image en temps réel**, ne dépendant plus d’un ID volatile.
* Deux images sont enregistrées :
  + l’image complète (capture\_YYYYMMDD\_HHMMSS.jpg),
  + l’image croppée (crop\_YYYYMMDD\_HHMMSS.jpg).
* Une ligne CSV est ajoutée avec : timestamp, nom de fichier, classe prédite, classe corrigée, bbox, confiance.

**Avantages :**

* La précision spatiale de l’apprentissage est grandement améliorée : seul le déchet concerné est utilisé.
* Le système devient robuste à la mise à jour fréquente des détections, car il n’utilise plus de référence volatile.
* Ce pipeline est désormais entièrement automatisé et fiable pour préparer un dataset de réentraînement.

**11. Réentraînement automatique du modèle à partir des retours utilisateurs**

**Question : Quand faut-il déclencher le réentraînement du modèle ?**  
Deux approches principales sont envisageables :  
• Déclenchement par seuil de volume : on peut décider de lancer un fine-tuning dès qu’un certain nombre de corrections validées est atteint (par exemple 100 ou 250). Cette méthode permet un meilleur contrôle sur la qualité des données utilisées à chaque réentraînement, et évite les déclenchements fréquents et inutiles.  
• Déclenchement périodique : on peut opter pour un réentraînement à fréquence fixe (par exemple une fois par semaine via une tâche planifiée), même s’il y a peu de nouvelles données. Cela permet une automatisation plus simple mais comporte le risque d’entraîner le modèle sur des volumes trop faibles.

**Solution retenue :**  
On adopte une stratégie hybride : on réentraîne le modèle une fois par semaine, mais uniquement si un seuil minimal de corrections valides a été atteint (exemple : au moins 100 nouvelles corrections).

**Ce qui a été mis en place :**  
Un workflow GitHub Actions (.github/workflows/clean\_and\_train.yml) a été configuré pour s’exécuter automatiquement chaque semaine (le mercredi à 4h du matin).  
Ce workflow appelle un script check\_and\_run.py situé dans le dossier backend/, qui vérifie que le fichier corrections.csv contient au moins 100 lignes valides. Si le seuil est atteint, il déclenche successivement :

* Le nettoyage et la préparation des données via clean\_correction.py,
* Puis l'entraînement du modèle via train\_from\_corrections.py.

**Question : Doit-on réentraîner uniquement avec les nouvelles images ou inclure l’historique ?**  
Différentes stratégies sont possibles :  
*•*Réentraîner uniquement sur les nouvelles corrections*:* cela permet un apprentissage rapide et réduit le risque de surapprentissage, mais devient risqué si le volume de corrections est faible ou déséquilibré.  
*•*Réentraîner sur un dataset cumulatif (nouvelles + anciennes données)*:* cela permet de conserver la mémoire globale du modèle, mais peut renforcer certains biais si les anciennes données sont surreprésentées.  
*•*Réentraîner sur un dataset mixé et pondéré*:* cela permet de préserver un bon équilibre, mais nécessite une sélection aléatoire et un pipeline un peu plus sophistiqué.

**Solution retenue :**  
On conserve l’ensemble de l’historique des corrections validées, et on constitue un dataset d’entraînement composé à 100 % des nouvelles corrections, complété par un échantillon aléatoire de 20 à 30 % des anciennes corrections.

**Ce qui a été mis en place :**  
Le fichier clean\_correction.py gère cette logique :

* À la première exécution, il crée le fichier historique\_corrections.csv et y copie les corrections valides du fichier corrections.csv,
* Le fichier corrections.csv est ensuite vidé pour accueillir de nouveaux retours utilisateurs,
* Un bloc de code (actuellement commenté) est prêt à intégrer automatiquement 20 à 30 % des anciennes corrections tirées de historique\_corrections.csv, dès que ce fichier existera.

**Question : Comment s’assurer que les données sont exploitables avant de relancer l’entraînement ?**  
Un script de vérification est nécessaire pour garantir la qualité des données utilisées. Il doit comprendre trois étapes :

**a. Validation de la structure**  
• Vérifier que tous les champs requis sont présents dans le fichier CSV.  
• Vérifier que les images associées (capture\_\*.jpg et crop\_\*.jpg) existent bien sur le disque.  
• Vérifier que les bounding boxes sont exploitables (taille minimale raisonnable, etc.).

**b. Analyse de qualité**  
• Vérifier que le score de confiance du modèle est disponible et cohérent.  
• Vérifier que la correction implique un vrai changement (la classe corrigée doit être différente de la classe prédite).

**c. Filtrage automatique**  
• Supprimer les images croppées trop petites, totalement noires ou redondantes.  
• Rassembler les corrections valides dans un répertoire corrections\_to\_train/ prêt à être utilisé pour le réentraînement.

**Ce qui a été mis en place :**  
Le script clean\_correction.py gère l’ensemble de ces vérifications :

* Il lit et filtre les lignes de corrections.csv contenant des champs manquants, des erreurs de format, ou des valeurs aberrantes (taille de crop, confiance, etc.).
* Les images valides sont copiées dans un dossier structuré (corrections\_to\_train/images/train et labels/train) au format YOLOv8.
* Le fichier historique\_corrections.csv est automatiquement alimenté avec les corrections traitées, afin d’être utilisé ultérieurement comme source d’apprentissage cumulée.

**TEST**

Nous avons pu valider le bon fonctionnement de toute la chaîne de réentraînement automatique grâce à un test en conditions réelles. Lorsqu’un utilisateur envoie une correction via l’interface, celle-ci est bien réceptionnée par le backend, enregistrée dans la base de données PostgreSQL distante, et prise en compte dans le processus d’entraînement.

Un exemple de log, généré lors d’une correction réelle, confirme la réussite de cette opération :

📥 Reçu : {

'timestamp': '2025-07-10T21:17:44.782068',

'image\_filename': 'crop\_73c103.jpg',

'wrong\_category': 'plastic',

'corrected\_category': 'plastic',

'confidence': 0.97

}

INFO: 34.169.125.42:0 - "POST /correction HTTP/1.1" 200 OK

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Le contenu généré par l’IA peut être incorrect.

Ce log atteste que la correction envoyée depuis l’interface utilisateur a bien été traitée, avec toutes les informations nécessaires : date, image croppée, catégorie initiale et catégorie corrigée, ainsi que la confiance du modèle.

Cette correction a été automatiquement ajoutée à la base PostgreSQL hébergée sur Render, et sera prise en compte lors du prochain cycle de réentraînement déclenché par GitHub Actions.

Ce processus, bien qu'automatisé, reste entièrement traçable : chaque correction reçue laisse une ligne dans les logs du backend, une entrée dans la base, et une image enregistrée dans les dossiers captures/ et crops/. Cela permet une revue manuelle en cas de doute ou de vérification de qualité.

La réussite de ce test constitue une preuve que l’intégration cloud fonctionne de bout en bout, et que le système est prêt à évoluer avec de vraies données utilisateurs.

**Une image contenant texte, logiciel, Icône d’ordinateur, Page web

Le contenu généré par l’IA peut être incorrect.**

**12. Intégration cloud, déploiement et limitations rencontrées**

**Différentes architectures ont été envisagées :**

• **Architecture full local** : backend FastAPI + frontend React lancés simultanément sur la machine locale. Cette approche garantit une fluidité optimale du flux vidéo, sans latence.

• **Architecture cloud hybride** : frontend React déployé sur Vercel, backend FastAPI déployé sur Render (Docker). Les deux communiquent via HTTP en utilisant une URL publique transmise au frontend via une variable d’environnement (NEXT\_PUBLIC\_API\_URL).

• **Architecture cloud pure** : envisager un déploiement 100% cloud avec ingestion vidéo distante (via edge device, caméra IP ou système embarqué). Cela permet de centraliser l’ensemble du traitement sur le cloud, mais nécessite des infrastructures supplémentaires.

**Solution retenue :**

Le déploiement cloud a été découpé en plusieurs modules distincts :

• Le frontend React a été déployé sur Vercel. Cette solution permet d’héberger facilement une interface web statique et responsive, accessible depuis n’importe quelle tablette ou navigateur. Cependant, comme Vercel n’intègre pas nativement de serveur backend, l’interface React fonctionne de manière isolée, sans accès direct au backend ni au flux vidéo temps réel.

• Le backend FastAPI a été dockerisé et déployé sur Render. Cette plateforme permet d’héberger des containers Docker exécutant le modèle YOLOv8 et l’ensemble des APIs REST associées. Les endpoints d’API de détection et de corrections (/detections, /correction) sont bien accessibles via des appels HTTP standards.

Cependant, plusieurs limitations majeures ont été rencontrées lors de la tentative de connexion directe entre le frontend cloud et le backend cloud :

Le flux vidéo temps réel, capturé avec OpenCV en StreamingResponse HTTP, ne s’adapte pas bien à une diffusion sur un backend cloud Render. Des délais de 14 secondes de latence ont été constatés lors du streaming, avec une réduction significative de la qualité vidéo. Cette latence est due à l'accumulation des frames sur le tunnel réseau et le fait que HTTP multipart n'est pas conçu pour du streaming vidéo temps réel distribué sur le cloud. La connexion entre les deux services frontend et backend reste donc partielle : les appels API passent, mais le flux vidéo ne peut pas être affiché correctement dans l’interface web cloudisée.

Le flux vidéo temps réel, capturé via OpenCV (cv2.VideoCapture), ne peut actuellement pas être transmis efficacement en cloud pur via HTTP multipart (StreamingResponse), en raison des limitations de latence et de bufferisation induites par ngrok et les tunnels cloud.

**Ce qui a été mis en place :**

Hébergement du frontend React sur Render :

Le **frontend**, développé en **React avec TypeScript**, constitue l’interface graphique de l’application. Cette interface permet à l’utilisateur de visualiser le flux vidéo annoté par le modèle YOLOv8, de cliquer sur des objets détectés, et d’éventuellement proposer des corrections.

Le frontend a été initialement déployé sur Vercel, qui offrait une solution d’hébergement gratuite adaptée à des sites statiques. Cependant, Vercel ne permet pas de configuration avancée côté serveur, ce qui posait problème pour la lecture du flux vidéo /video\_feed.

C’est pourquoi un déploiement final a été réalisé sur Render, qui permet également d’héberger gratuitement des applications statiques, mais avec un meilleur contrôle de la configuration et des redirections.

Déploiement du backend FastAPI sur Google Colab + tunnel Ngrok :

Le **backend** est une API FastAPI contenant toute la logique métier :

* Réception des frames envoyées par l’uploader
* Passage au modèle YOLOv8 (chargé avec Ultralytics)
* Filtrage et transformation des prédictions
* Réponse JSON pour affichage

Le fichier api.py contient l’ensemble de cette logique, exécutée dans un notebook Google Colab, grâce au GPU gratuit disponible.

**Modules utilisés** :

*from fastapi import FastAPI, UploadFile*

*from fastapi.responses import StreamingResponse, JSONResponse*

*from pyngrok import ngrok*

*import nest\_asyncio*

*import uvicorn*

*from ultralytics import YOLO*

*import threading, queue, cv2, uuid, time, numpy as np*

**Initialisation de l’API et du tunnel sécurisé** :

*app = FastAPI()*

*ngrok.set\_auth\_token("TON\_TOKEN\_NGROK")*

*public\_url = ngrok.connect(8000)*

*print("🔗 API disponible :", public\_url)*

**Exemple d’endpoint d’upload de frame** :

@app.post("/upload\_frame")

async def upload\_frame(file: UploadFile):

contents = await file.read()

frame = cv2.imdecode(np.frombuffer(contents, np.uint8), cv2.IMREAD\_COLOR)

frame\_queue.put(frame)

return {"status": "ok"}

**Exemple d’endpoint de récupération des prédictions** :

*@app.get("/detections")*

*async def get\_detections():*

*return JSONResponse(content=latest\_detections)*

**Communication entre frontend et backend : une architecture RESTful :**

Le dialogue entre les deux composants se fait exclusivement via des appels HTTP REST, permettant de rester compatible avec n’importe quel frontend.

Les endpoints clés sont :

* /upload\_frame : reçoit les images depuis la webcam (via uploader.py)
* /detections : retourne les dernières prédictions YOLO
* /video\_feed : renvoie un flux MJPEG généré en direct avec StreamingResponse pour visualiser les détections

Dans le fichier **uploader\_local.py**, le fichier qu’on utilise en local pour lancer l’activation de la caméra entre autre, on utilise des requêtes POST toutes les 0.5 secondes pour transmettre la dernière image capturée :

*requests.post(*

*f"{BACKEND\_URL}/upload\_frame",*

*files={"file": ("frame.jpg", img\_encoded.tobytes(), "image/jpeg")},*

*timeout=20*

*)*

Et côté interface React, on récupère les détections via fetch :

*tsx*

*CopierModifier*

*const fetchDetections = async () => {*

*const res = await fetch(`${apiUrl}/detections`);*

*const data = await res.json();*

*setDetections(data);*

*};*

Puis les détections sont affichées dans un <canvas> interactif superposé à l’image vidéo :

*tsx*

*CopierModifier*

*ctx.strokeStyle = categoryColor; // Couleur dynamique par catégorie*

*ctx.strokeRect(x, y, w, h);*

*ctx.fillText(`${box.label} (${box.score})`, x, y - 5);*

Une fois le backend démarré dans le notebook, il est exposé à Internet grâce à Ngrok via une URL publique sécurisée et fixe (grâce à un abonnement à **8 $/mois**) :  
***https://turkey-adjusted-namely.ngrok-free.app***

Cette URL permet au frontend React de communiquer avec le backend sans se soucier du routage réseau ou du NAT.

Au niveau du visuel de l’interface, plusieurs problemes liés à l’affichage visuel des détections ont été identifiés et résolus. Un des problèmes rencontrés était un souci de persistance des cadres. Concrètement, une détection précédente restait affichée même après la disparition de l’objet sous la caméra. Le backend continuait à envoyer les anciennes prédictions sans les mettre à jour, ce qui rendait l’affichage confus :

Une image contenant texte, logiciel, Logiciel multimédia, Page web

Le contenu généré par l’IA peut être incorrect.

Solution : un premier filtre a été ajouté au code du backend sur Google Colab pour effacer les anciennes prédictions si la nouvelle image ne contient aucun objet en se basant sur les logs des nouvelles frames :

*INFO: 90.7.240.132:0 - "POST /upload\_frame HTTP/1.1" 200 OK*

*0: 480x640 1 biological, 2 metals, 13.1ms*

*Speed: 2.5ms preprocess, 13.1ms inference, 2.4ms postprocess per image at shape (1, 3, 480, 640)*

*📦 3 détection(s)*

*✅ Frame reçue ((480, 640, 3))*

De plus, comme on peut le voir sur la précédente capture d’écran il y a aussi plusieurs fois la même détection qui s’affiche car la position change légèrement mais du coup le cadre de la frame se multiplie lorsque plusieurs détections sont très proches ou similaires, YOLOv8 génère plusieurs boites presque identiques. Cela rend le visuel chargé et peu lisible :

Une image contenant texte, capture d’écran, logiciel, ordinateur

Le contenu généré par l’IA peut être incorrect.

La solution a été d’ajouter un filtre de suppression des doublons avec seuil de recouvrement pour ne garder qu’un seul cadre pertinent.

Il y a aussi eu un conflit entre cadres natifs YOLOv8 (rouges) et cadres interactifs React (colores). Les premiers servaient uniquement à valider les performances du modèle, mais leur persistance gênait l’expérience utilisateur contrairement à ceux qui étaient rajoutés sur l’interface du frontend et qui étaient cliquable pour l’utilisateur.

Une image contenant texte, capture d’écran, ordinateur, logiciel

Le contenu généré par l’IA peut être incorrect.

Nous avons donc décidé de supprimer les cadres de YOLOv8 dans le flux vidéo, seuls les cadres interactifs sont maintenant affichés. Nous avons aussi fait en sorte de s’assurer que les cadres étaient bien de couleurs différentes pour chaque catégorie dans le frontend :

Une image contenant texte, capture d’écran, logiciel, ordinateur

Le contenu généré par l’IA peut être incorrect.

*INFO: 2a01:cb09:e047:3486:a041:11ab:97a5:f5f3:0 - "POST /upload\_frame HTTP/1.1" 200 OK*

*0: 480x640 1 paper, 1 plastic, 13.0ms*

*Speed: 4.8ms preprocess, 13.0ms inference, 2.8ms postprocess per image at shape (1, 3, 480, 640)*

*2 détection(s)*

*INFO: 2a01:cb09:e047:3486:a041:11ab:97a5:f5f3:0 - "OPTIONS /correction HTTP/1.1" 200 OK*

*INFO: 2a01:cb09:e047:3486:a041:11ab:97a5:f5f3:0 - "GET /detections HTTP/1.1" 200 OK*

*Correction enregistrée : cardboard → paper*

*INFO: 2a01:cb09:e047:3486:a041:11ab:97a5:f5f3:0 - "POST /correction HTTP/1.1" 200 OK*

*Frame reçue ((480, 640, 3))*

Limitations actuelles

• Le backend et le frontend sont techniquement découplés dans l’architecture microservices actuelle (Vercel et Render séparés), et nécessitent une gestion soignée des variables d’environnement pour les synchroniser correctement.

**Améliorations possibles**

• Mise en place d’un edge device pour capturer le flux vidéo et envoyer les frames au cloud via API REST, solution industrielle standard pour l’architecture edge-cloud.

• Passage futur à un protocole de streaming vidéo temps réel optimisé (WebRTC, RTSP, S3 streaming ingestion, ou pipeline Kafka cloud) pour la gestion des flux vidéos distants.

**Résumé des acquis**

* **Interface frontend stylisée, responsive et fonctionnelle**  
  L’interface React permet une utilisation fluide sur desktop ou tablette, avec une mise en page claire, des bordures arrondies et une hiérarchie visuelle cohérente. La détection des objets est rendue lisible grâce à une gestion dynamique des cadres et des couleurs.
* **Flux vidéo YOLOv8 intégré de façon fluide dans le frontend**  
  Le flux vidéo provient d’un serveur FastAPI hébergé dans Google Colab, accessible via un tunnel Ngrok. Il est intégré en direct dans l’interface grâce à un flux StreamingResponse, sans latence excessive. Les cadres de détection sont superposés de façon synchrone.
* **Interaction utilisateur précise via clics sur objets détectés**  
  Les utilisateurs peuvent cliquer sur chaque cadre pour accéder à une boîte de dialogue de correction. Ces cadres sont dynamiquement mis à jour à chaque frame, avec une correspondance spatiale exacte avec l’image diffusée.
* **Affichage personnalisé des cadres de détection**  
  Chaque catégorie de déchet a une couleur spécifique définie dans un dictionnaire CATEGORY\_COLORS. Seuls les cadres sur lesquels on peut interagir sont affichés (ceux générés dans le frontend), tandis que les cadres bruts de YOLOv8 ne sont plus redessinés, pour éviter la confusion.
* **Nettoyage automatique des détections**  
  À chaque nouvelle frame reçue, les anciennes prédictions sont automatiquement effacées du backend. Cela garantit que si aucun objet n’est détecté sur l’image suivante, l’interface ne conserve pas d’anciennes boîtes obsolètes. Ce mécanisme est implémenté via une simple affectation :  
  *latest\_detections = detections*.
* **Filtrage implicite des doublons via NMS**  
  YOLOv8 applique par défaut un filtrage des détections qui se superposent trop (Non-Max Suppression). Cela évite les doublons et assure une seule boîte par objet. En plus, une fonction personnalisée peut être ajoutée pour filtrer manuellement les cadres trop proches.
* **Adaptation de l’interface aux dimensions du flux**  
  Le flux vidéo est redimensionné en CSS pour respecter les proportions et rester dans les bordures prévues par l’interface. Le canvas de superposition s’adapte automatiquement à la taille réelle du flux vidéo
* **Intégration complète du pipeline : détection → clic → correction → capture → stockage**  
  Tout le circuit est opérationnel : un objet est détecté, l’utilisateur peut cliquer dessus pour corriger sa catégorie, et la correction est envoyée au backend. Cela prépare le terrain pour un enregistrement des données annotées et un futur entraînement supervisé.
* **API FastAPI fonctionnelle et stable**  
  Le backend est robuste et divisé en endpoints dédiés : /upload\_frame pour recevoir les frames, /detections pour renvoyer les prédictions, /video\_feed pour diffuser le flux, etc. Le serveur supporte une charge modérée en traitement temps réel grâce à la file queue.Queue().
* **Déploiement cloud opérationnel et économique**  
  Le frontend est déployé sur Render via React. Le backend est exécuté sur Google Colab (grâce à un GPU gratuit), exposé avec un tunnel Ngrok payant (abonnement ~9$/mois pour utiliser un domaine fixe). Cette configuration hybride a permis de contourner les limitations techniques de Render (latence, CPU-only).
* **Tunnel Ngrok personnalisé pour une URL fixe**  
  Grâce au plan payant de Ngrok, un domaine fixe (https://turkey-adjusted-namely.ngrok-free.app) a été attribué au backend Colab. Cela permet de fixer les variables d’environnement côté frontend et uploader sans devoir modifier l’URL à chaque lancement.
* **Enregistrement des données structuré, prêt pour un futur réentraînement du modèle**  
  L’architecture actuelle permet de récupérer les images, les détections et les corrections pour les stocker et constituer un dataset personnalisé de déchets triés. Ce jeu de données peut être ensuite utilisé pour fine-tuner YOLOv8 avec supervision.

**Conclusion**

Ce projet annuel, mené à deux nous a permis de construire un pipeline complet de détection, d’interaction et d’amélioration automatique autour du tri des déchets. L’objectif de départ était simple sur le papier : détecter des objets jetés sur un plateau, via une caméra, et les classer automatiquement dans une des catégories prédéfinies (plastique, carton, nourriture, verre, métal). Mais au fil de l’année, ce projet est devenu bien plus qu’une simple démonstration de deep learning.

Nous avons commencé par concevoir une interface utilisateur claire et pédagogique, pensée pour un usage par des enfants. Cela a exigé un vrai travail d’UX : choix des couleurs, réactivité, affichage des catégories sous forme d’émojis, et surtout, intégration fluide du flux vidéo. Cette partie frontend, bien que visuellement simple, a été essentielle pour garantir une expérience utilisateur intuitive et interactive.

Côté technique, la connexion avec YOLOv8, la mise en place du modèle, le traitement des flux vidéo en temps réel et la structuration du dataset nous ont confrontés à des défis concrets. La configuration de l’environnement, la gestion des caméras, l’intégration dans FastAPI ou encore la segmentation des images avec précision sont autant d’étapes qui nous ont permis de solidifier nos compétences pratiques en computer vision.

L’aspect le plus intéressant a été la mise en place d’un système de correction utilisateur : permettre à l’utilisateur de cliquer sur un objet mal détecté, corriger la classe, et voir cette information utilisée plus tard pour réentraîner le modèle. Cela implique toute une chaîne technique que nous avons progressivement mise en place : récupération de l’image croppée, enregistrement dans un CSV, structuration du dataset, puis réentraînement automatique à intervalle régulier.

Ce système est aujourd’hui capable de s’auto-améliorer avec le temps, en prenant en compte les retours utilisateurs, tout en protégeant le modèle contre les erreurs ou les clics trop rapides. Grâce à des règles de pondération basées sur la confiance du modèle, les feedbacks sont filtrés pour ne pas dégrader l’apprentissage.

Sur la partie cloud et déploiement, nous avons testé plusieurs architectures, dont une architecture cloud hybride (frontend sur Vercel ou Render, backend sur Google Colab avec tunnel Ngrok). Cela nous a permis d’assurer une certaine portabilité, tout en gardant la capacité d’accéder à un GPU gratuitement pour l’inférence et l’entraînement. Même si le streaming en cloud pur reste limité par la latence, le reste de l’application fonctionne de façon fiable, y compris à distance.

Nous avons aussi appris à organiser notre code, automatiser les tâches (avec GitHub Actions), créer des scripts robustes, valider les données avant entraînement, et surtout, à penser un projet dans son ensemble : frontend, backend, modèle, base de données, interaction utilisateur, et automatisation. Chaque brique a été pensée pour s’intégrer dans un système cohérent, utile et durable.

Travailler en binôme a été à la fois un défi et une vraie force. La charge de travail était plus dense, mais nous avons su nous organiser efficacement : répartir les tâches intelligemment, confronter nos idées, tester, échouer, recommencer. Être deux nous a aussi permis d’apprendre davantage sur l’ensemble du projet, de mieux comprendre chaque étape, et d’améliorer notre communication. À deux, il est plus simple de se synchroniser, d’échanger régulièrement et d’avancer sans dépendre des disponibilités d’un plus grand groupe.

En résumé, ce projet a été pour nous bien plus qu’un exercice technique. Il nous a appris à penser un produit complet, à anticiper l’usage réel, à traiter des données dans un contexte vivant et imparfait (les retours utilisateurs), et à construire un système qui apprend, évolue et s’adapte. Le système est aujourd’hui opérationnel, modulable, et prêt à évoluer. Il pourra servir de base à d’autres expérimentations ou à un déploiement plus large dans un cadre éducatif ou environnemental.