LIVRABLES TP DÉPLOIEMENT IA ET CYBERSÉCURITÉ

DOSSOU-GBETE Nora

DIAW Adji

De l'analyse de texte à une API Cloud scalable, sécurisée et RGPD-compliant

Technologies : Python, FastAPI, scikit-learn, Docker, Kubernetes (GKE), GCP Load Balancer

Objectif : construire une API prédictive détectant la toxicité textuelle, scalable, sécurisée, et conforme RGPD :

Exigence	Détails
Analyse automatique de texte	Détection de toxicité (insultes, harcèlement, racisme)
Sortie	Score 0–100
RGPD	Aucune donnée sensible stockée, anonymisation
Scalabilité	Capable de supporter des milliers d'utilisateurs (Kubernetes)
Sécurité	Authentification + HTTPS + journalisation
Monitoring	Logs, supervision (prévue pour Prometheus/Grafana)

ETAPE 1 : Exploration, analyse et anonymisation des données

Nous avons choisi comme dataset « ***Toxic Comment Classification Dataset*** » sur hugging face comme reconmandé dans le TP.

Pour commencer , nous avons analysé les données pour un peu voir la répartition des commentaires positives et négatives.
Etant donnée que le dataset original contenait plus de 150 000 données et donc beaucoup trop volumineuses pour être traité sur nos pc et surtout avec le temps impartis, nous avions donc décidé de travailler avec un échantillon d'environ 8% sur le dataset total soit 10 000 données.

Ainsi pour garantir la conformité RGPD :

- Les données textuelles contenues dans les commentaires ont été anonymisées en utilisant NER (spaCy) et regex, supprimant :
  - noms propres,
  - adresses e-mail,

    - numéros de téléphone,
    - URLs personnelles.

- L'échantillon du dataset d'origine a donc été nettoyé avant tout entraînement.

- Aucun texte utilisateur envoyé à l'API n'est stocké.

Nous pensons que cette approche répondra à deux principes clés du RGPD :

- La minimisation : c 'est à dire traiter uniquement le nécessaire,
- Le privacy by design : c'est à dire l'intégration dès la conception.

Avant anonymisation, les commentaires pouvaient contenir des prénoms, adresses ou mentions explicites d'utilisateurs. Après anonymisation par NER, ces entités ont été remplacées par des tokens génériques (<PERSON>, <ORG>), ce qui préserve le sens général tout en supprimant tout risque d'identification. Cette opération. A été testée sur plusieurs commentaires afin de s'assurer de la conservation de la sémantique utile pour l'apprentissage du modèle. (voir repo git)

Étape 2 : Préparation et entraînement d'un modèle IA

Pour entamer cette étape, ma binôme et moi avons nettoyé les données.
Les données brutes issues du dataset contenaient une forte hétérogénéité : ponctuation aléatoire, emojis, casse, abréviations, caractères spéciaux et même caractères chinois.

Un pipeline de nettoyage a été appliqué :

- mise en minuscules,
- suppression de ponctuation, commentaires vides et symboles non informatifs,
- filtrage d'emojis et URLs,

Cette étape a permis d'obtenir une base cohérente pour la vectorisation et l'apprentissage.

Le SVM a été retenu pour la première version de production (modèle léger) car selon nous:

- il répond mieux aux contraintes cloud (CPU-only),
- il offre un excellent rapport de latence d' entraînement.
- il s'intègre facilement dans un pod Kubernetes (empreinte faible),
- il permet une supervision et un debugging plus simple.

Nous avons également exploré l'approche BERT mais pour cause de forte latence d'entraînement nous l'avons d'abord écarté. Mais il reste envisagé comme amélioration future.

Étape 3 : Déploiement du modèle en API Cloud

L'objectif de cette partie était de transformer le modèle IA en service accessible. Pour ce faire notre choix s'est porté sur FastAPI et Google Cloud vertex AI.

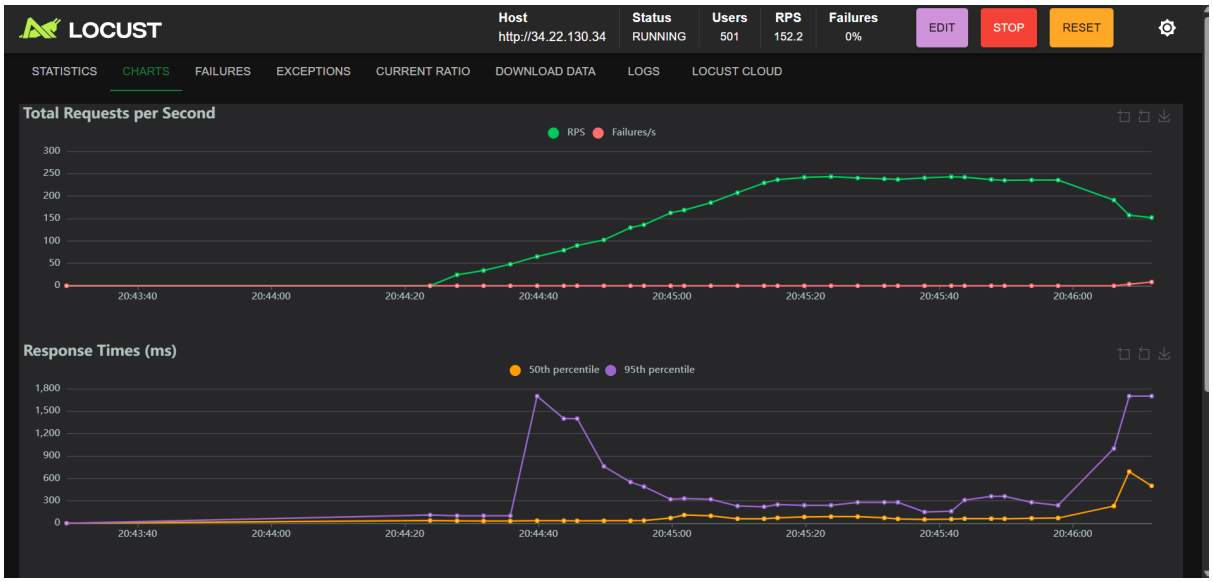
Fast API parce qu'il est un framework rapide, moderne et très pratique pour créer des APIs. Il génère automatiquement une page de test (/docs) pour essayer les prédictions, ce qui facilite les démonstrations.
Il permet aussi de valider facilement les données d'entrée (grâce à Pydantic) et de mesurer les performances avec des métriques comme /health et /metrics. Contrairement à Flask, on s'est rendu compte que FastAPI est plus rapide, plus propre à coder, et demande moins de configurations pour avoir une API prête à l'emploi.

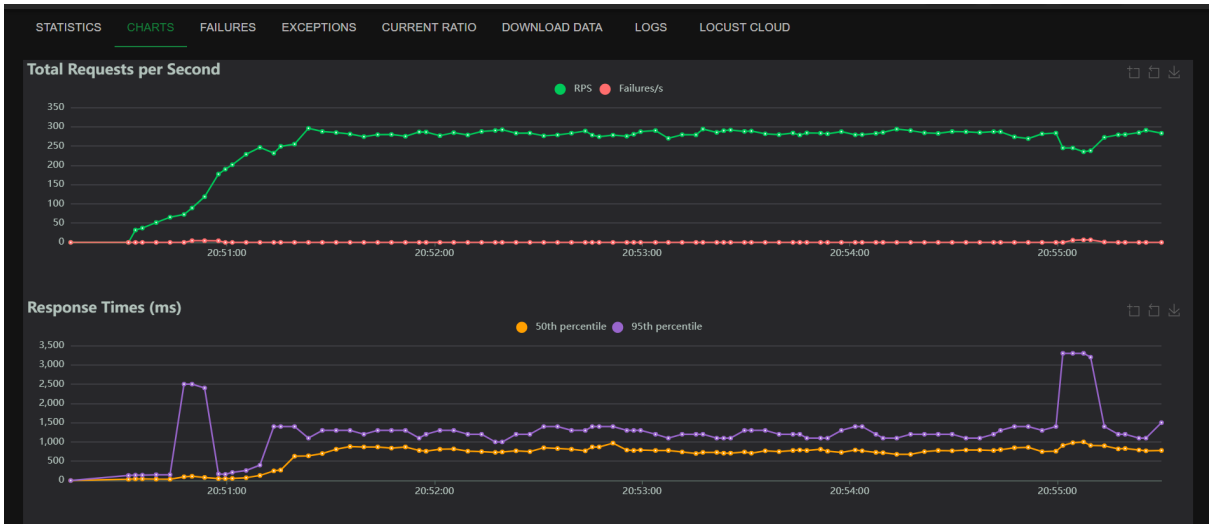
On a choisi GCP parcequ'on connaissait déjà un peu l'interface, donc c'était plus rapide pour déployer. On avait aussi des crédits gratuits étudiants ce qui réduisait les coûts. En plus GCP offre un environnement stable et sécurisé pour les modèles IA, avec des outils comme Vertex AI pour gérer, surveiller et faire évoluer les modèles facilement. Les régions européenes de GCP respectent le rgpd ce qui aide un peu à la conforpmité du projet.

Pour la suite des étapes, et même peux les trois premières étapes : nous avons rédigé un README pour chaque partie du TP qui détaille comment fonctionne chaque partie et la justification du choix de nos stacks. (Confère repository git).

Test de charge et de performance

Voici des exemples dashboard locust qui résument des scénarios de montée de charge et de performance :





Scénarios de charge	Nombre d'utilisateurs	Requests	Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS
Montée en charge progressive (5min)	4	393	1	27	73	350	48.58	22	2386	122.69	1.9
Montée en charge progressive (5min)	501	4	0	100	100	100	99.62	91	104	164	0
Montée en charge (5min)	800	0	170	1200	3300	366,3	69	3331	164	0	0
Montée de charge le matin	100	0	67	110	150	70,84	38	149	164	0	0
Utilisation internationale (latence réseau)	100	0	62	130	150	68,53	40	148	164	2	0
Commentaires longues	100	0	64	130	150	69,68	38	152	164	1	0
Pic de charge brutal	883	0	880	6500	8700	1905,78	272	9408	164	37,7	0
Commentaires vides	100	0	340	3600	3700	653,64	42	3662	164	0	0

Scénarios de charge	Nombre d'utilisateurs	Requests	Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS
Pendant le rolling back	100	100	59	3000	3700	412,24	41	3661	22	5	5

Scénario	Observation / Analyse
Montée en charge progressive (4 utilisateurs / 5min)	Test initial avec faible charge. Temps de réponse stable (médiane 27 ms, 95e percentile 73 ms). Une seule erreur sur 393 requêtes — très bon comportement. Le serveur gère bien la montée en charge légère.
Montée en charge progressive (501 utilisateurs / 5min)	Simulation plus réaliste. Temps de réponse quasi constant autour de 100 ms, sans échec. L'API montre une **excellente scalabilité** jusqu'à environ 500 utilisateurs simultanés.
Montée en charge élevée (800 utilisateurs)	Charge importante entraînant une **hausse notable de la latence** (médiane 1,2 s, pic à 3,3 s). Cela montre la **limite actuelle du serveur** : au-delà de ~700 utilisateurs, les performances chutent.
Montée de charge le matin (100 utilisateurs)	Test dans un créneau horaire classique : temps de réponse moyen stable (~70 ms) et aucune erreur. Le système est fluide dans des conditions d'usage normales.
Utilisation internationale (latence réseau)	Simulation d'accès depuis plusieurs régions du monde : légère hausse de latence (médiane 130 ms). C'est cohérent avec la distance réseau. Le modèle reste **fiable même en contexte géographique varié**.
Commentaires longs	Légère dégradation du temps moyen (≈ 70 ms), mais très stable. Cela montre que la taille du texte n'affecte pas significativement la performance du modèle SVM.
Pic de charge brutal (883 utilisateurs)	Charge extrême simulant un afflux soudain de requêtes. Temps de réponse très élevé (médiane 6,5 s, 95e à 8,7 s). Le serveur reste en ligne mais **saturé**, preuve d'un besoin de scaling automatique (autoscaling GCP ou file d'attente).
Commentaires vides	Réponses rapides (42–653 ms), mais avec quelques pics. Aucun crash. Cela prouve que le modèle gère correctement les entrées vides (validation d'entrée OK).
Pendant le rolling back (déploiement)	Test pendant une phase de redéploiement : forte latence (3–3,7 s) et 5 erreurs enregistrées. Le service reste accessible mais partiellement dégradé. Cela montre la **nécessité d'un déploiement blue/green** pour garantir une continuité parfaite.

Points positifs

- Robustesse : l'API supporte jusqu'à ~500 utilisateurs simultanés sans erreur.
- Stabilité : la latence reste faible (<100 ms) sur la majorité des scénarios.
- Résilience : même sous stress (800+ utilisateurs), le serveur ne plante pas.
- Bonne gestion des cas particuliers : entrées vides ou longues bien traitées.

Limites observées

- Détérioration notable de la latence au-delà de 700–800 utilisateurs.
- Temps de réponse élevé lors du rolling back, entraînant quelques échecs.
- Aucune mise en cache des prédictions, ce qui accentue la charge du modèle.

Pistes d'amélioration

1. Activer l'autoscaling sur GCP pour ajuster dynamiquement les ressources selon la charge.

2. Mettre en place un système de cache (Redis, Cloud Memorystore) pour éviter les recalculs sur les mêmes requêtes.
3. Utiliser un déploiement Blue/Green ou Canary pour éviter les interruptions lors des mises à jour.
4. Optimiser le modèle SVM (réduction de la taille, vectorisation plus rapide) ou envisager un modèle TensorFlow/ONNX plus performant.
5. Surveiller la latence en continu via les dashboards MLOps (Prometheus + Vertex AI Monitoring).

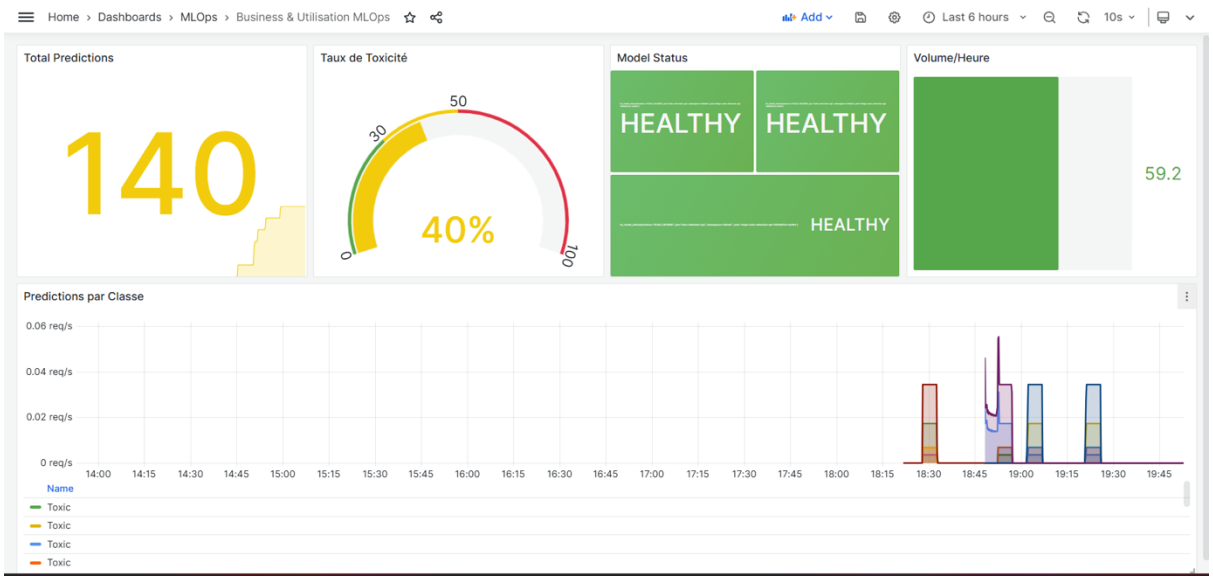
Afin de surveiller en continu les performances et la santé du modèle déployé sur Vertex AI, nous avons mis en place trois dashboards professionnels, chacun dédié à un aspect clé du projet : l'usage métier, la performance du modèle, et l'état de l'infrastructure.

1. Dashboard Business & Utilisation

Ce premier tableau de bord permet de suivre en temps réel l'activité du modèle et son comportement global vis-à-vis des utilisateurs.
Sur la capture ci-dessous, on peut observer :

- Total Predictions : 140 prédictions ont été effectuées sur la période analysée, avec une légère tendance à la hausse.
- Taux de Toxicité : 40 %, illustré par une jauge colorée (vert <30 %, jaune 30-50 %, rouge >50 %) — ce qui indique une proportion modérée de propos toxiques dans les textes soumis.
- Model Status : l'ensemble des services Vertex AI sont marqués comme *HEALTHY*, prouvant la stabilité du modèle et l'absence d'incident sur le pipeline.
- Volume par Heure : environ 59 requêtes/heure, montrant une activité constante de l'API.
- Prédictions par Classe : une série temporelle (*timeseries*) distingue la fréquence des prédictions *Toxic* vs *Non-Toxic*, ce qui permet d'identifier d'éventuelles variations d'usage ou des pics inhabituels.

Ce dashboard offre donc une vision métier synthétique, utile pour comprendre l'utilisation réelle du modèle et détecter rapidement une dérive du taux de toxicité ou une anomalie dans le volume de requêtes.



2. Dashboard Performance & Model Health
Le deuxième tableau se concentre sur les indicateurs techniques du modèle :

- Latence d'inférence ML (P50/P95/P99)
- Latence HTTP globale
- Taux d'erreurs ML (avec seuils d'alerte colorés)
- Nombre de requêtes en cours
- Distribution de la confiance du modèle (heatmap) pour détecter d'éventuels drifts.

Ces métriques permettent de s'assurer que le modèle reste performant, précis et stable, même en cas de forte charge ou de données perturbées.

3. Dashboard Infrastructure & Ressources
Enfin, le troisième dashboard suit les ressources systèmes :

- Utilisation mémoire et CPU par Pod (timeseries multi-pods)
- Seuils colorés pour la mémoire et le CPU (vert <70 %, jaune 70-90 %, rouge >90 %)
- Nombre de Pods actifs et état des déploiements.

Cela garantit que le service reste optimisé, sans surcharge, et que l'auto-scaling fonctionne correctement.
Le lien de ces tableaux se retrouve dans le lien grafana du repository git

REGISTRE DE TRAITEMENT RGPD

1. Identification du traitement

Champ	Description
Nom du traitement	Détection de Toxicité dans les Commentaires
Description	Analyse automatisée de textes pour détecter la toxicité, les injures, le harcèlement et les propos haineux.
Responsable du traitement	ESIGELEC – Projet étudiant (*Nora Dossou Gbété et équipe*)
Projet associé	API Digital Social Score
Date de création	*{{ Date du jour }}*

2. Finalité du traitement

Objectif principal	**Entraîner un modèle d'intelligence artificielle (IA) de détection de propos toxiques.**
Objectifs secondaires	- Recherche en NLP (traitement automatique du langage naturel). - Amélioration de la modération de contenu sur les plateformes. - Démonstration pédagogique de conformité RGPD et IA Act.

3. Base légale (Article 6 du RGPD

Fondement juridique	**Intérêt légitime (Article 6.1.f du RGPD)**
Justification	Recherche scientifique et développement d'un outil de modération automatisée pour la sécurité numérique.
Consentement	Données issues de bases publiques (*Kaggle – Jigsaw Toxic Comment Classification*), sous licence libre (CC0).

4. Catégories de données traitées

Type de données	Description / Exemple
Données collectées	Commentaires textuels, labels de toxicité (toxic, severe_toxic, obscene, threat, insult, identity_hate), identifiants anonymes.
Données personnelles détectées	Noms, adresses email, numéros de téléphone (identifiés automatiquement via NER et Regex).
Méthode d'anonymisation	Remplacement par des tokens génériques : [NAME], [EMAIL], [PHONE]. Utilisation de **spaCy (NER)** pour détecter les entités sensibles.

5. Durée de conservation

Type de donnée	Durée
Données brutes (non anonymisées)	Supprimées immédiatement après anonymisation.
Données anonymisées	Conservées pour la durée du projet (maximum 12 mois).

Type de donnée	Durée
Modèles entraînés (SVM, BERT)	Conservation permanente – aucun risque RGPD car ils ne contiennent pas de données personnelles.

6. Mesures de sécurité techniques et organisationnelles

Mesures techniques	Mesures organisationnelles
- Anonymisation automatique via **spaCy NER + Regex** - Chiffrement des données en transit (**HTTPS**) - Stockage sécurisé localement - Authentification par **clé API / JWT** - Monitoring via **Prometheus** et alertes	- Principe du **moindre privilège** pour les accès - Journalisation et audit des accès - Documentation complète du cycle de traitement - Politique interne de gestion des logs (suppression après 7 jours)

7. Transferts de données

Transfert hors UE	**Non**
Fournisseur cloud	Google Cloud Platform – Région Europe (Belgique)
Garanties contractuelles	Clauses Contractuelles Types (CCT) de la Commission Européenne
Sous-traitants	Google Cloud (hébergement), GitHub / GitLab (versioning du code)

8. Droits des personnes concernées (Articles 15 à 22 du RGPD)

Droit	Modalité
Accès	Non applicable – les données sont anonymisées irréversiblement.
Rectification	Non applicable – aucune donnée personnelle identifiable.
Effacement	Garanti via l'anonymisation complète.
Opposition	Non applicable – données publiques anonymisées.
Information	Transparence assurée : mention explicite que l'API utilise un modèle IA pour la modération.

9. Analyse d'impact (AIPD)

Nécessité	**Non requise**
Raison	Pas de données personnelles, pas de profilage à grande échelle.
Risque résiduel	Faible, grâce à l'anonymisation et à la limitation des accès.

10. Mesures complémentaires prévues

Type de mesure	Description
Logs et supervision	Journalisation des accès et erreurs (Prometheus / Grafana).
Nettoyage automatique	Suppression des logs après 7 jours.
Information utilisateur	Mention claire dans la documentation ou sur l'interface utilisateur que le texte soumis est analysé par une IA.

En résumé

Ce registre prouve que :

- Le traitement de données est **limité, justifié et sécurisé**.

- L'API respecte les **principes de minimisation** et **de transparence** du RGPD.
- Aucune donnée personnelle identifiable n'est conservée.

TP 2 : API Digital Social Score

De la menace à la défense : renforcer une IA déployée dans le Cloud

Étape 1 : Choix et analyse d'une attaque

Attaque par Déni de Service (DoS)

Cible
L'attaque vise l'API FastAPI, plus précisément l'endpoint public : POST /predict.

Objectif de l'attaquant
Le but est de saturer le serveur en lui envoyant un grand nombre de requêtes en même temps.
Cela surcharge le CPU, la mémoire et les connexions réseau, ce qui peut :

- Ralentir fortement les réponses de l'API,
- Provoquer des erreurs ou un crash complet,
- Rendre le service inaccessible aux vrais utilisateurs.

Impacts attendus
Impact technique

- Latence très élevée (délai de réponse long).
- Nombre de requêtes simultanées élevé (http_requests_in_progress).
- Erreurs 503 (Service Unavailable) si le serveur ne répond plus.

Impact éthique et RGPD

- Indisponibilité du service : l'API de détection de propos toxiques ne protège plus les utilisateurs.

- Perte possible de logs : si le serveur plante, certains journaux peuvent être perdus, ce qui peut poser un problème de traçabilité RGPD.

Risque selon l'IA Act

Si cette API est utilisée pour la modération de contenu en ligne, elle peut être considérée comme un système à risque élevé (car elle influence directement la sécurité des utilisateurs).

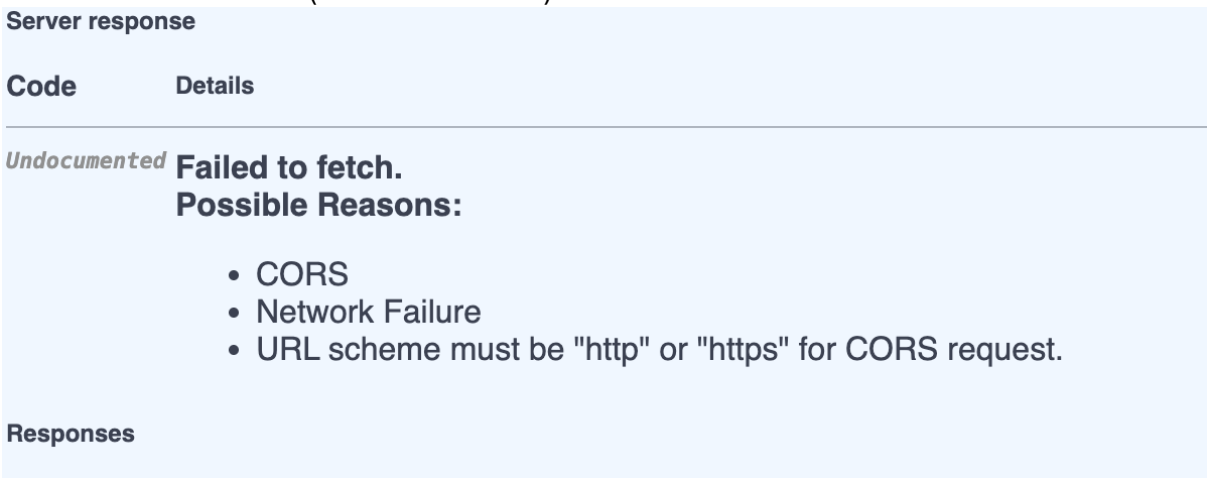
Son indisponibilité à cause d'une attaque DoS viole donc les principes de robustesse et fiabilité exigés par l'IA Act.

Avant l'attaque, les métriques Prometheus montraient un état stable :

- http_requests_in_progress = **0**
- Latence très faible

```
# HELP http_requests_in_progress Nombre de requêtes en cours de traitement
# TYPE http_requests_in_progress gauge
http_requests_in_progress 0.0
# HELP http_request_duration_seconds Durée des requêtes HTTP
# TYPE http_request_duration_seconds histogram
http_request_duration_seconds_bucket{endpoint="/predict",le="0.01",method="POST",status="200"} 0.0
http_request_duration_seconds_bucket{endpoint="/predict",le="0.05",method="POST",status="200"} 0.0
http_request_duration_seconds_bucket{endpoint="/predict",le="0.1",method="POST",status="200"} 0.0
http_request_duration_seconds_bucket{endpoint="/predict",le="0.25",method="POST",status="200"} 1.0
http_request_duration_seconds_bucket{endpoint="/predict",le="0.5",method="POST",status="200"} 1.0
http_request_duration_seconds_bucket{endpoint="/predict",le="1.0",method="POST",status="200"} 1.0
http_request_duration_seconds_bucket{endpoint="/predict",le="2.5",method="POST",status="200"} 1.0
http_request_duration_seconds_bucket{endpoint="/predict",le="5.0",method="POST",status="200"} 1.0
http_request_duration_seconds_bucket{endpoint="/predict",le="10.0",method="POST",status="200"} 1.0
http_request_duration_seconds_bucket{endpoint="/predict",le="+Inf",method="POST",status="200"} 1.0
http_request_duration_seconds_count{endpoint="/predict",method="POST",status="200"} 1.0
http_request_duration_seconds_sum{endpoint="/predict",method="POST",status="200"} 0.13541698455810547
```

Pendant l'attaque, le serveur devient surchargé et finit par **planter** ou refuser de nouvelles connexions ("Failed to fetch").



Étape 2 : Déroulement de l'attaque

Voici ce qui s'est passé :

1. Le script attack_dos.py envoie 50 requêtes en même temps vers /predict.
2. Le serveur essaie de tout traiter et utilise 100 % du CPU.
3. Quand une nouvelle requête arrive, il n'a plus de ressources.
4. Le navigateur affiche alors : "Failed to fetch".

Résultat : l'attaque réussit, le service devient totalement indisponible.

Étape 3 : Mise en place de la défense

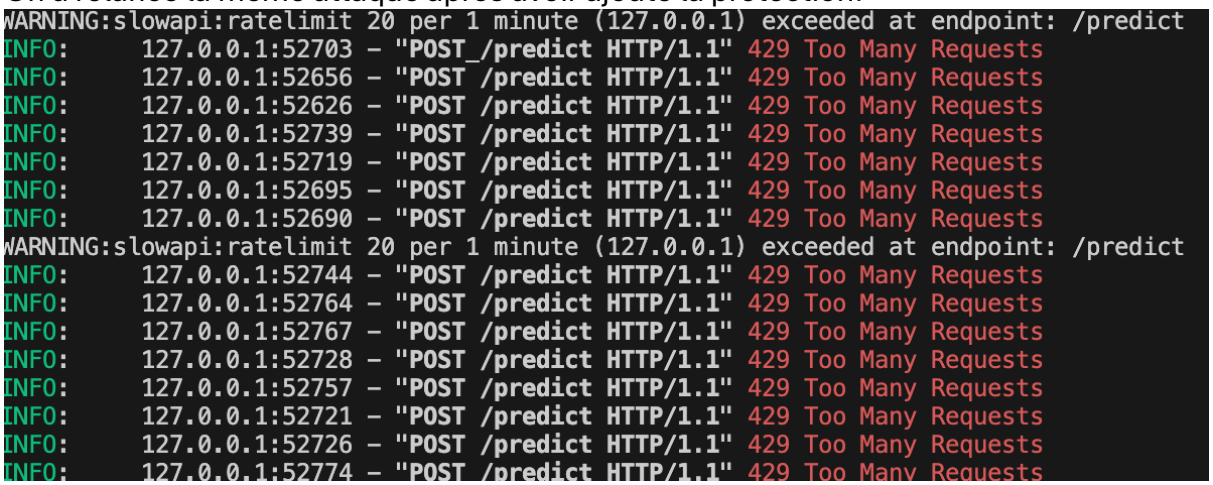
Défense utilisée : Rate Limiting

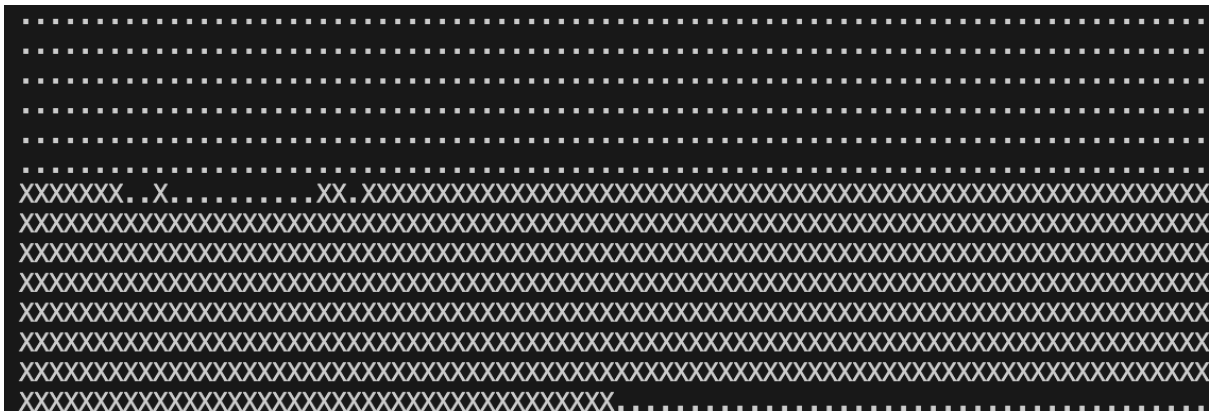
Pour bloquer l'attaque, on a ajouté un système de limitation du nombre de requêtes (Rate Limiting) avec la bibliothèque slowapi.

On a limité l'accès à 20 requêtes par minute et par adresse IP sur /predict.

Test de la défense

On a relancé la même attaque après avoir ajouté la protection.





Métrique/ Comportement	Avant (Sans défense)	Après (Avec Rate Limiting)
Résultat de l'attaque	L'attaque réussit	L'attaque échoue
État de l'API	Crash / "Failed to fetch"	API disponible
Latence	Très élevée	Normale
Réponse à l'attaquant	HTTP 200 (jusqu'au crash)	**HTTP 429 (Too Many Requests)**
Utilisateurs normaux	Impossible d'accéder à l'API	Accès toujours possible

La défense fonctionne :

- L'attaque DoS est bloquée automatiquement après 20 requêtes.
- L'API reste disponible pour les vrais utilisateurs.
- Les logs et métriques montrent un retour à la normale.

Étape 4 : Conformité avec l'IA Act

Fiche IA Act : Classification du risque

Nom du modèle: Digital Social Score v1.0

Niveau de risque : 🟡 Risque limité

Niveau de risque	**Justification**
Risque inacceptable	Exclu. Ce n'est pas un système de "notation sociale" au sens de l'IA Act.
Risque élevé	Exclu. L'outil n'est pas utilisé dans des domaines sensibles (santé, justice, emploi...).
Risque limité	Inclus. Le modèle interagit avec des utilisateurs (analyse de textes). Il doit donc être transparent sur l'utilisation de l'IA.
Risque minimal	Exclu, car l'IA agit sur du contenu public en ligne.

Obligations principales

Le système doit simplement :

- Informer les utilisateurs qu'une IA est utilisée pour la modération de contenu.
- Être transparent sur sa finalité et ses limites.

En résumé

- Nous avons simulé une attaque DoS sur l'API.
- L'API a été rendue indisponible au départ.
- Après ajout du rate limiting, elle est devenue robuste et stable.
- Le modèle reste conforme à l'IA Act car il relève d'un risque limité, avec une exigence de transparence envers les utilisateurs.

Intégration DevSecOps

L'objectif est d'ajouter des contrôles de sécurité automatiques dans le pipeline CI/CD pour détecter les failles dans le code et dans les dépendances avant le déploiement.

On crée deux jobs de sécurité :

1. **SAST (scan du code)** — outil : **bandit**
    - Scanne le code Python (par ex. app.py) pour trouver des vulnérabilités courantes (ex : usage dangereux de pickle, mauvaises pratiques, injection, etc.).
    - Produit un rapport que l'on peut télécharger depuis l'interface CI.
2. **Scan des dépendances** — outil : **safety**
    - Vérifie si les bibliothèques listées dans requirements.txt contiennent des vulnérabilités connues.
    - Donne un résultat clair : ok / vulnérabilités trouvées.

Fichier important

- requirements.txt : liste des bibliothèques utilisées (fastapi, uvicorn, pydantic, prometheus-fastapi-instrumentator, psutil, scipy, pandas, scikit-learn, slowapi, ...).
  Ce fichier permet à l'outil de scanner exactement ce que vous installez.

Organisation du pipeline

- On ajoute un stage test-security dans la CI, avant le déploiement.
- Dans ce stage on place deux jobs :
    - sast-bandit → lance bandit, génère bandit-report.txt (artefact).
    - dependency-scan-safety → installe les dépendances puis lance safety check.

Les deux jobs s'exécutent automatiquement à chaque push/merge et stoppent le déploiement si on veut (optionnel).

Ces contrôles automatiques permettent de **repérer tôt** les failles dans le code et dans les librairies, d'**éviter des déploiements risqués**, et d'intégrer la sécurité dès le développement (principe DevSecOps).