Fake News Detection Platform
Programmation Distribuée

Wassim CHIKHI & Dounia HOUADRIA

18 mai 2025

Résumé

Ce rapport présente la conception d'une application web interactive dédiée à la détection automatisée de fausses informations. Basée sur des modèles de traitement du langage naturel accessibles via l'API de Hugging Face, la plateforme permet aux utilisateurs de soumettre un texte et d'obtenir une prédiction en temps réel via une interface moderne développée avec React, et une architecture backend fondée sur les microservices.

Mots-clés : microservices, Kubernetes, Docker, Kafka, REST API, BERT, RoBERTa, PostgreSQL, React, fake news.

---

1 Introduction

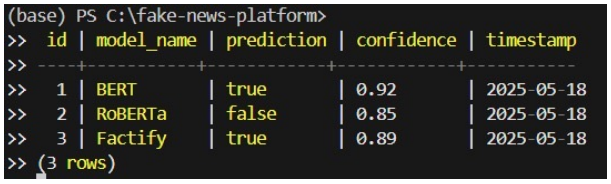
Dans le cadre du cours de Programmation Distribuée, nous avons développé une application web distribuée nommée **Fake News Detection Platform**.

Cette plateforme permet d'analyser la véracité d'un texte à l'aide de modèles NLP, tout en étant architecturée autour de microservices REST. Elle est déployée dans un environnement Kubernetes, offrant scalabilité, résilience et modularité.

2 Développement de l'application

Notre système repose sur une architecture en microservices, dans laquelle chaque composant est dédié à une tâche bien définie. L'API Gateway joue le rôle de point d'entrée unique vers l'application, en exposant les différentes routes REST nécessaires à la communication entre le frontend et les services internes. Le service de prédiction est responsable de l'analyse des textes : il interagit avec l'API de Hugging Face afin de solliciter des modèles de traitement du langage naturel tels que BERT ou RoBERTa. Une fois la prédiction effectuée, les résultats sont pris en charge par le service de stockage, qui les enregistre dans une base de données PostgreSQL pour permettre un accès persistant aux historiques. Enfin, le service de statistiques agrège les données issues des prédictions afin d'alimenter le tableau de bord accessible depuis l'interface utilisateur.

(a) Schéma de la base de données PostgreSQL



(b) Exemple de données dans la base PostgreSQL

FIGURE 1 – Représentations de la base de données PostgreSQL

Pour assurer une communication efficace et découplée entre les microservices, nous avons intégré Apache Kafka comme système de messagerie. Chaque service peut ainsi produire ou consommer des messages en se connectant à des topics dédiés. Par exemple, le service de prédiction envoie les résultats d'analyse dans un topic Kafka, pendant que le service de statistiques les consomme afin de mettre à jour les métriques globales affichées sur le tableau de bord. Cette approche basée sur la publication et la souscription améliore la scalabilité du système et permet une gestion plus souple des flux de données asynchrones.

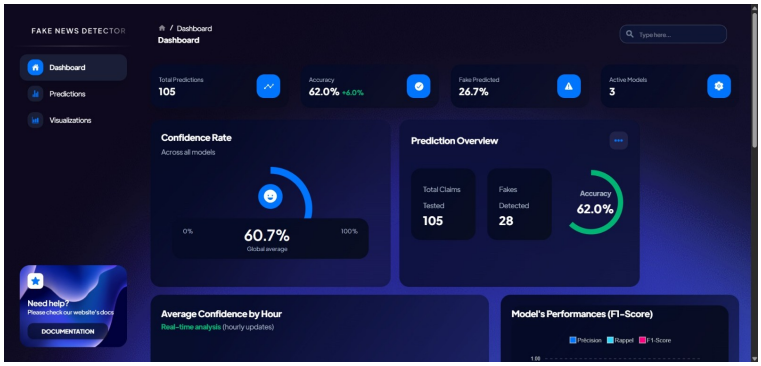


FIGURE 2 – Capture d'écran de l'application.

Chaque service se connecte à des *topics* spécifiques pour publier ou consommer des messages. Par exemple :

— Le service de prédiction publie les résultats d'analyse dans un topic nommé `predictions`.

— Le service de statistiques consomme les messages de ce topic pour mettre à jour les métriques.

Ce mécanisme permet de découpler les services entre eux, favorisant ainsi la scalabilité, la tolérance aux pannes, et la facilité d'ajout de nouveaux services consommateurs sans modifier le cœur du système.

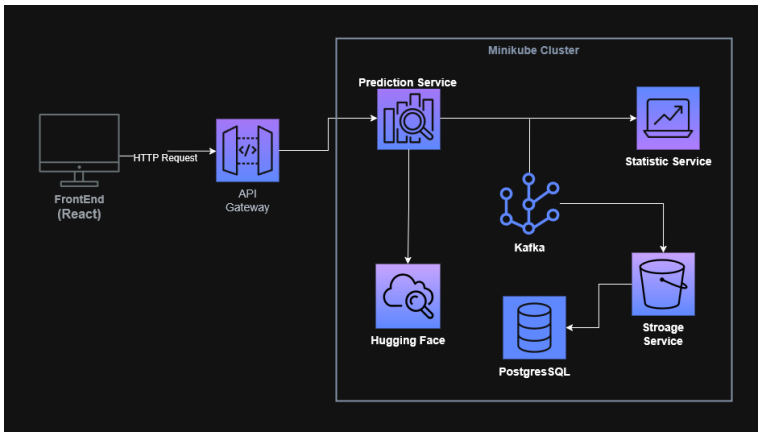


FIGURE 3 – Diagramme de l'architecture.

3 Création des images Docker

Une fois les microservices développés, nous avons créé une image Docker pour chacun d'entre eux à l'aide d'un fichier Dockerfile dédié. Ces images contiennent toutes les dépendances nécessaires au bon fonctionnement des services, garantissant ainsi un environnement d'exécution stable et reproductible. Après génération, elles ont été poussées sur Docker Hub afin de faciliter leur déploiement via Kubernetes. L'ensemble des Dockerfiles ainsi que les scripts associés sont disponibles dans notre dépôt GitHub, référencé dans la section Liens utiles.

```
> docker images | grep "douns"
douns/api-gateway                    latest    c6a57c5d4516   24 minutes ago   241MB
douns/fake-news-platform-frontend    latest    5494c0e6443b   2 hours ago      85.7MB
douns/fusion-analyzer                latest    bce689d76860   3 hours ago      1.54GB
douns/text-analyzer                  latest    2e50dad0ac56   3 hours ago      2.03GB
douns/rent                           latest    8d8b29bcae89   7 weeks ago      727MB
douns/car-rental-service             latest    d2f422c54865   7 weeks ago      727MB
```

FIGURE 4 – Aperçu des images Docker générées localement pour les différents services.

4 Déploiement Kubernetes

Pour simuler un environnement Kubernetes en local, nous avons utilisé Minikube. Chaque microservice de notre application dispose de ses propres fichiers de configuration au format YAML, notamment un fichier de type Deployment et un fichier de type Service. Les fichiers Deployment permettent de définir les pods à lancer, le nombre de réplicas souhaité, ainsi que l'image Docker à utiliser pour chaque service. Les fichiers Service, quant à eux, servent à exposer les pods à l'intérieur du cluster à l'aide du type ClusterIP, facilitant ainsi la communication entre les services. Enfin, un fichier Ingress est utilisé pour exposer l'API Gateway à l'extérieur du cluster via une URL, rendant l'application accessible depuis le navigateur.

```
> kubectl get all
NAME                                  READY   STATUS    RESTARTS   AGE
pod/api-gateway-f6dff5889-hrg8n       1/1     Running   0          36s
pod/debug                             1/1     Running   0          18m
pod/frontend-56c64fdc64-qgb58         1/1     Running   0          152m
pod/fusion-analyzer-575546bf9b-tq7z4  1/1     Running   0          152m
pod/text-analyzer-55c978f8f6-qhcwx    1/1     Running   0          152m

NAME                      TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
service/api-gateway       ClusterIP      10.109.171.104   <none>        8000/TCP       11s
service/frontend          LoadBalancer   10.104.124.150   <pending>     80:32091/TCP   67m
service/fusion-analyzer   ClusterIP      10.101.58.115    <none>        8002/TCP       67m
service/kubernetes        ClusterIP      10.96.0.1        <none>        443/TCP        67d
service/nginx             NodePort       10.100.124.5     <none>        80:32148/TCP   3h20m
service/text-analyzer     ClusterIP      10.102.14.105    <none>        8001/TCP       68m

NAME                              READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/api-gateway       1/1     1            1           36s
deployment.apps/frontend          1/1     1            1           152m
deployment.apps/fusion-analyzer   1/1     1            1           152m
deployment.apps/text-analyzer     1/1     1            1           152m

NAME                                        DESIRED   CURRENT   READY   AGE
replicaset.apps/api-gateway-f6dff5889       1         1         1       36s
replicaset.apps/frontend-56c64fdc64         1         1         1       152m
replicaset.apps/fusion-analyzer-575546bf9b  1         1         1       152m
replicaset.apps/text-analyzer-55c978f8f6    1         1         1       152m
```

FIGURE 5 – Ressources deployées dans Kubernetes.

5 Sécurisation via RBAC Kubernetes

Afin d'ajouter une couche de sécurité au sein de notre cluster Kubernetes, nous avons mis en œuvre un contrôle d'accès basé sur les rôles (RBAC) pour restreindre les permissions du microservice `api-gateway`.

Compte de service dédié

Nous avons créé un compte de service nommé `api-gateway-sa`, que le déploiement `api-gateway` utilise explicitement via le champ `serviceAccountName`. Ce compte permet d'isoler les permissions de ce service du reste des composants du cluster.

5.1 Rôle personnalisé

Un rôle (`Role`) nommé `api-gateway-role` a été défini dans le namespace `default`. Ce rôle accorde uniquement les permissions nécessaires à l'API Gateway, comme la lecture des pods (actions `get`, `list`, `watch`). Ces autorisations sont utiles pour interagir avec certains services internes ou pour effectuer des vérifications d'état lors du routage.

5.2 RoleBinding

Le rôle est ensuite associé au compte de service `api-gateway-sa` grâce à un objet `RoleBinding` (`api-gateway-binding`). Ainsi, seul ce service possède les droits associés, en suivant le principe du moindre privilège.

5.3 Vérification des permissions

Nous avons validé les permissions du service avec la commande suivante :

```
kubectl auth can-i list pods --as=system:serviceaccount:default:api-gateway-sa
```

Celle-ci confirme que le service possède uniquement les droits nécessaires à sa tâche.

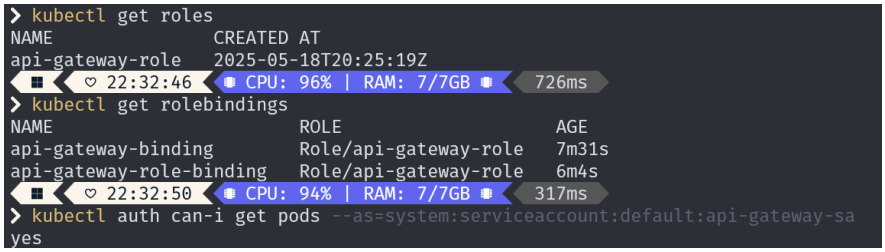


FIGURE 6 – RBAC Kubernetes Roles et RolesBinding.

5.4 ClusterRole et ClusterRoleBinding

Dans le cas où l'API Gateway aurait besoin d'accéder à des ressources en dehors du namespace `default`, nous avons également préparé une définition de `ClusterRole`.

`ClusterRole` fonctionne comme un `Role`, mais ses permissions s'étendent à l'ensemble du cluster, ce qui est utile pour des ressources globales ou pour interagir entre namespaces.

`ClusterRoleBinding` permet ici d'étendre les droits du compte de service `api-gateway-sa` à des ressources partagées au sein du cluster, en maintenant une sécurité contrôlée.

Cette séparation entre `Role` et `ClusterRole` permet une gestion fine des permissions, tout en respectant les bonnes pratiques de sécurité en environnement Kubernetes.

```
❯ kubectl get clusterroles api-gateway-clusterrole
NAME                     CREATED AT
api-gateway-clusterrole   2025-05-18T21:20:17Z
❯ kubectl get clusterrolebindings api-gateway-clusterrolebinding
NAME                             ROLE                                AGE
api-gateway-clusterrolebinding   ClusterRole/api-gateway-clusterrole   5m42s
❯ kubectl auth can-i get pods --as=system:serviceaccount:default:api-gateway-sa
yes
```

FIGURE 7 – RBAC Kubernetes ClusterRoles et ClusterRolesBinding.

6 Conclusion

Ce projet nous a offert une immersion concrète dans les concepts de la programmation distribuée, en mettant en pratique des technologies modernes comme les microservices, Docker, Kubernetes et Kafka. À travers la conception et le déploiement d'une plateforme complète de détection de fake news, nous avons pu expérimenter une architecture scalable, modulaire et orientée production.

L'intégration de mécanismes comme le RBAC, la messagerie Kafka ou encore le déploiement via Minikube nous a permis de mieux comprendre les enjeux liés à la sécurité, à la communication inter-services et à la résilience des systèmes distribués.

Perspectives d'évolution :

- Mise en place d'une solution de centralisation des logs avec la stack ELK.
- Ajout de monitoring et d'alerting via Prometheus et Grafana.
- Déploiement de la plateforme sur un cloud provider (GKE, AWS EKS).
- Intégration d'un service de notifications pour améliorer l'interaction utilisateur.

En résumé, ce projet a constitué une excellente opportunité pour consolider nos compétences DevOps et cloud-native tout en réalisant une application à fort impact.

Liens Utiles

Voici les principaux liens liés au projet :

— **Dépôt GitHub** : `https://github.com/vvazzim/fake-news-platform`

— **Docker Hub Images** :
  — api-gateway : `https://hub.docker.com/r/douns/api-gateway`
  — text-analyzer : `https://hub.docker.com/r/douns/text-analyzer`
  — fusion-analyzer : `https://hub.docker.com/r/douns/fake-news-platform-frontend`
  — frontend : `https://hub.docker.com/r/douns/fusion-analyzer`

Annexes

Google Labs

— Wassim CHIKHI :

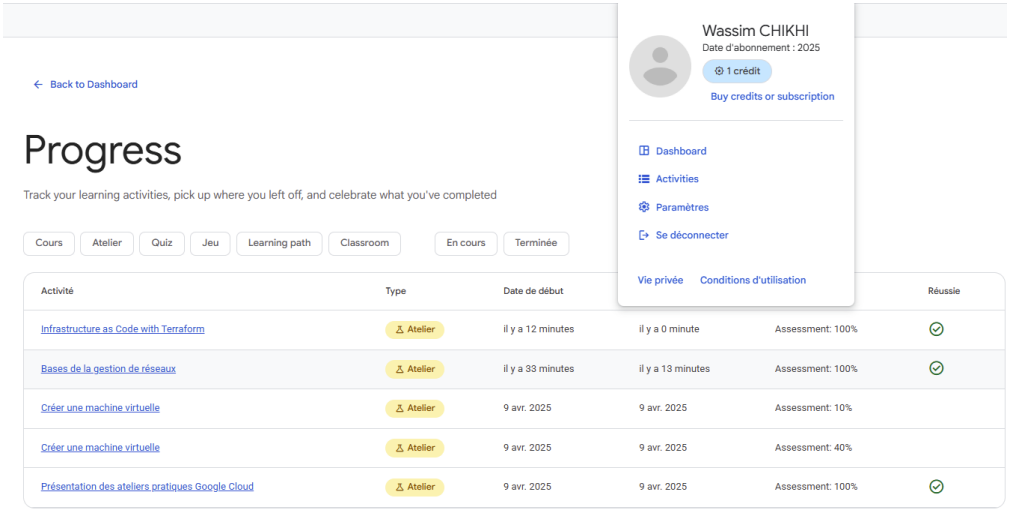


FIGURE 8 – Activité Google lab de Wassim.

— Dounia HOUADRIA :

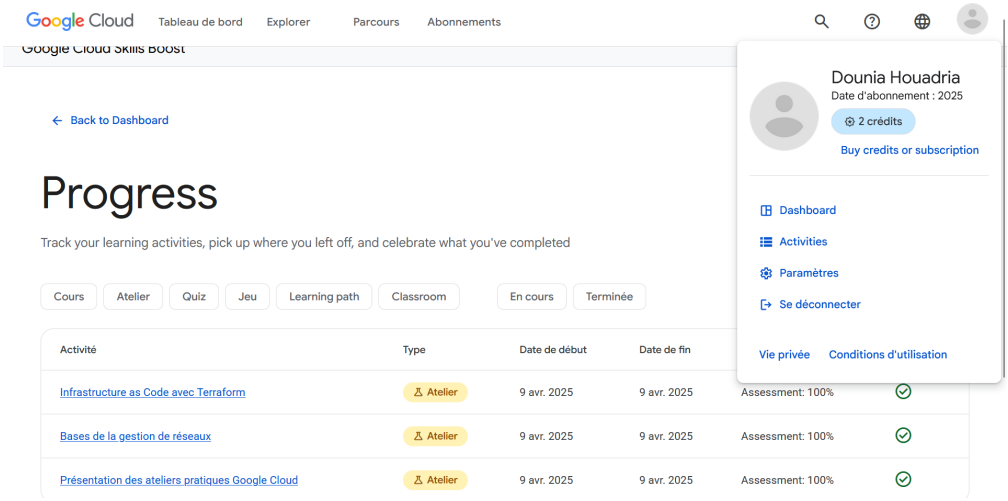


FIGURE 9 – Activité Google lab de Dounia.

Références

— **Vision UI Dashboard Pro React**
Product Page : Vision UI Dashboard Pro React
Description : Le projet Fake News Detection Platform utilise l'interface utilisateur basée sur le template Vision UI Dashboard Pro React, un tableau de bord moderne et dynamique conçu par Creative Tim. Ce template offre une structure réactive et élégante adaptée aux projets d'analyse de données.