

**INF4100 Systèmes repartis et infonuagiques**

**Implémentation d'un Système de Reconnaissance d'Images Distribué**

JOSEPH Samuel Jonathan

11/30/2024

**PLAN**

* **Introduction**
* **Architecture du Système**
* **Implémentation Frontend**
* **Implémentation Backend**
* **Containerisation et Déploiement**
* **Tests et Performance**
* **Conclusion**

**INTRODUCTION**

Dans le cadre du cours de Systèmes répartis et infonuagique (INF4100), j’ai développé un système de reconnaissance d'objets en temps réel qui illustre les concepts fondamentaux des architectures distribuées. Ce projet s’appuie sur les notions explorées dans le TP2, notamment les appels de méthodes à distance et la répartition des tâches dans un environnement distribué.

**Contexte**  
Le TP2 m’a initié aux concepts de gRPC et à la gestion d'une baie de serveurs pour le traitement distribué. J’ai transposé ces notions dans un cas concret : un système de reconnaissance d'objets utilisant des services distants pour l’analyse d’images. Ce projet met en pratique la communication inter-processus et la gestion des ressources distribuées.

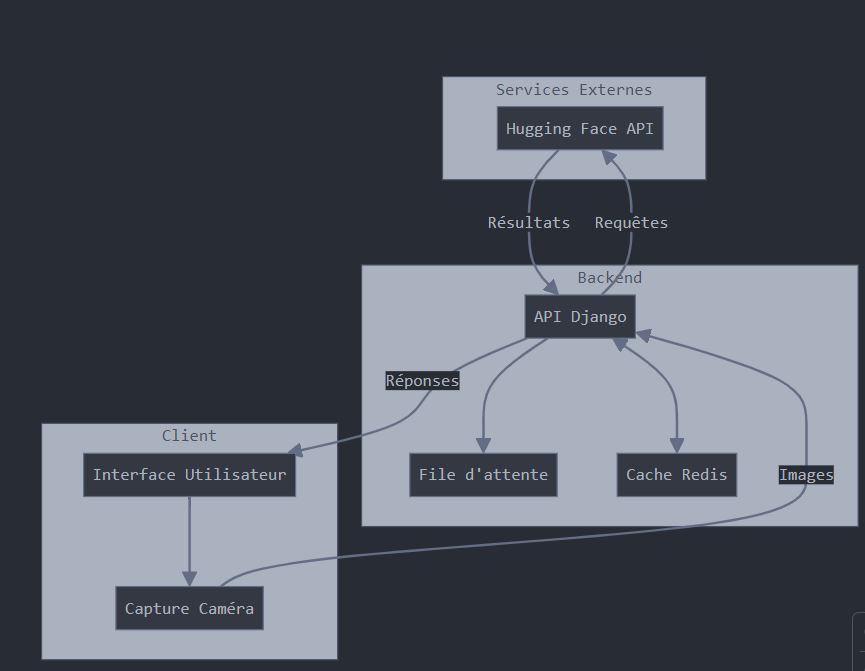
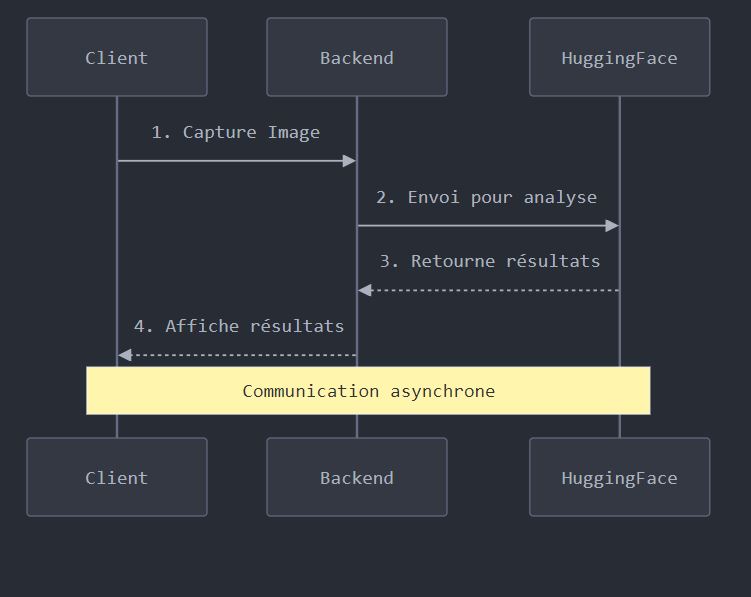
**Problématique**  
La reconnaissance d’objets en temps réel pose des défis caractéristiques des systèmes distribués :

* Gestion efficace des appels distants entre le client et les services d'analyse
* Synchronisation des opérations dans un contexte distribué
* Optimisation des performances et gestion des ressources
* Tolérance aux pannes et gestion des erreurs

**Objectifs**  
Le projet vise à :

* Développer une architecture distribuée efficace pour la reconnaissance d'objets
* Mettre en œuvre les concepts de communication inter-services
* Optimiser la gestion des ressources et des performances
* Garantir une expérience utilisateur fluide malgré la nature distribuée du système

**Structure du rapport**  
Ce rapport détaillera l'implémentation, l'architecture du système, les choix techniques, la containerisation et les performances obtenues, en mettant en évidence les aspects distribués de la solution.

**ARCHITECTURE DU SYSTEME**

Le système développé s'appuie sur une architecture distribuée intégrant les concepts du TP2 de Systèmes répartis. Il comporte trois composants principaux collaborant de manière asynchrone pour un traitement efficace des données. Le client, en tant que couche de présentation, capture les images en temps réel via la webcam, effectue un prétraitement des données et offre une interface utilisateur intuitive, tout en gérant une communication asynchrone avec le backend pour une expérience fluide. Le backend, développé avec Django, agit comme gestionnaire central en organisant les requêtes dans une file d'attente FIFO, en assurant leur validation et en orchestrant la communication avec le service d'analyse, reflétant ainsi les principes de gestion des tâches du TP2. Le service d'analyse, basé sur l'API Hugging Face, représente la couche de traitement spécialisé, prenant en charge l'analyse des images et retournant les résultats de reconnaissance, similaire aux serveurs de calcul présentés dans le TP2. La communication entre ces composants suit un flux asynchrone inspiré du modèle RPC : capture et envoi des images du client vers le backend, traitement et mise en file d'attente par le backend, analyse par le service Hugging Face, puis retour des résultats au client via le backend. L'architecture intègre des mécanismes d'optimisation et de sécurité tels que la limitation intelligente des requêtes simultanées, un système de cache pour les résultats fréquents, une authentification sécurisée via des tokens API, une validation systématique des données à chaque étape et une gestion robuste des erreurs distribuées. Cette conception architecturale démontre l'application pratique des concepts théoriques des systèmes distribués tout en assurant performance, scalabilité et fiabilité. Elle illustre concrètement les principes de distribution des tâches et de communication inter-services, établissant un parallèle direct avec les objectifs pédagogiques du TP2.

**IMPLEMENTATION FRONTEND**

Le frontend du système a été développé en utilisant les technologies web standards : **HTML**, **CSS** et **JavaScript**. Il constitue la couche de présentation et assure l'interaction directe avec l'utilisateur. Les principales fonctionnalités implémentées incluent la capture d'images en temps réel via la webcam, le prétraitement des données pour optimiser leur envoi au backend, et la gestion asynchrone des communications pour une expérience utilisateur fluide.

**Capture d'images et prétraitement :**

La capture d'images est réalisée à l'aide de l'API MediaDevices de JavaScript, qui permet d'accéder à la webcam de l'utilisateur de manière sécurisée. Une fois l'image capturée, elle est redimensionnée et compressée pour réduire la taille des données, ce qui améliore les performances lors de la transmission au backend.

//Javascript

async function captureImage() {

const canvas = document.createElement('canvas');

// Configuration du canvas et capture de l'image

const imageData = canvas.toDataURL('image/jpeg', 0.8);

return imageData;

}

**Communication asynchrone avec le backend :**

La communication avec le backend est gérée de manière asynchrone en utilisant les fonctions fetch avec async/await. Cela permet de ne pas bloquer l'interface utilisateur pendant le traitement des requêtes, améliorant ainsi la réactivité de l'application. Cette approche est en cohérence avec le modèle RPC étudié dans le TP2, où le client effectue des appels de procédures à distance sans attendre la réponse immédiate.

//Javascript

async function sendToBackend(imageData) {

try {

const response = await fetch('/api/analyze', {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify({ image: imageData })

});

const result = await response.json();

displayResults(result);

} catch (error) {

console.error('Erreur lors de la communication avec le backend:', error);

}

}

**Gestion des états et des événements :**

Le frontend gère les états de l'application, tels que l'activation de la caméra et le statut de l'analyse en cours, en utilisant des variables d'état et des écouteurs d'événements. Cela assure une synchronisation efficace entre les différentes actions de l'utilisateur et les processus du système.

//Javascript

let isAnalyzing = false;

async function captureAndAnalyze() {

if (isAnalyzing) return;

isAnalyzing = true;

const imageData = await captureImage();

await sendToBackend(imageData);

isAnalyzing = false;

}

**Interface utilisateur intuitive :**

Des efforts particuliers ont été faits pour rendre l'interface utilisateur intuitive et réactive. Les éléments visuels tels que les boutons, les indicateurs de chargement et les messages d'erreur sont gérés pour offrir un retour immédiat à l'utilisateur, améliorant ainsi l'expérience globale.

**Parallèle avec le TP2 :**

L'implémentation frontend reflète les principes de **gestion asynchrone des tâches** et de **communication inter-processus** abordés dans le TP2. En particulier, l'utilisation d'appels asynchrones pour communiquer avec le backend est similaire aux **appels de méthodes à distance (RPC)** étudiés, où le client n'a pas besoin d'attendre la fin du traitement pour continuer ses opérations.

**IMPLEMENTATION BACKEND**

Le backend de notre système est développé avec le framework **Django**, qui offre une structure robuste pour la création de sites web sécurisés et évolutifs. Il agit comme le **gestionnaire central** du système, orchestrant les interactions entre le frontend et le service d'analyse d'images. Les principales responsabilités du backend incluent la gestion des utilisateurs, la validation et le traitement des requêtes, la communication avec l'API Hugging Face, et la mise en place des mesures de sécurité nécessaires.

**Gestion des Utilisateurs**

Le backend gère les fonctionnalités essentielles d'**inscription**, de **connexion** et de **déconnexion** des utilisateurs. Chaque utilisateur dispose d'un **profil** associé qui stocke sa propre clé API Hugging Face de manière sécurisée. Cela permet aux utilisateurs d'utiliser leur propre clé pour les analyses, offrant une personnalisation et une gestion des quotas d'utilisation de l'API.

**Modèle UserProfile :**

Dans models.py, nous avons défini le modèle UserProfile qui étend le modèle utilisateur par défaut de Django pour inclure la clé API Hugging Face :

python

Copy code

from django.contrib.auth.models import User

from django.db import models

class UserProfile(models.Model):

user = models.OneToOneField(User, on\_delete=models.CASCADE)

huggingface\_api\_key = models.CharField(max\_length=100, blank=True, null=True)

Cette implémentation utilise une relation **OneToOneField** pour lier chaque profil utilisateur à un compte utilisateur, garantissant une association unique et sécurisée.

**Traitement des Requêtes et Communication avec le Service d'Analyse**

Lorsqu'une image est soumise pour analyse, le backend effectue plusieurs étapes clés :

1. **Réception et Validation de la Requête :**

Le backend reçoit les données d'image du frontend et valide les informations reçues pour s'assurer qu'elles sont complètes et conformes aux attentes.

1. **Récupération de la Clé API de l'Utilisateur :**

Il récupère la clé API Hugging Face associée à l'utilisateur courant. Si l'utilisateur n'a pas fourni de clé personnelle, une clé API par défaut est utilisée.

##python

def analyze\_image\_view(request):

if request.method == 'POST':

image\_data = request.POST.get('image')

user\_profile = UserProfile.objects.get(user=request.user)

api\_key = user\_profile.huggingface\_api\_key or DEFAULT\_HUGGINGFACE\_API\_KEY

# Appel au service d'analyse avec la clé API appropriée

1. **Appel au Service d'Analyse :**

Le backend envoie une requête au service d'analyse de Hugging Face en utilisant la clé API appropriée, puis attend la réponse de manière asynchrone.

1. **Traitement de la Réponse et Envoi au Frontend :**

Une fois le résultat reçu, il est traité et renvoyé au frontend pour affichage à l'utilisateur.

Cette séquence reflète le **flux de communication asynchrone** étudié dans le TP2, où le backend agit comme un **intermédiaire** entre le client et le service d'analyse, gérant les **appels de méthodes à distance** et la **synchronisation des opérations**.

**Gestion des Erreurs et des Exceptions**

Le backend inclut une gestion robuste des erreurs pour assurer la fiabilité du système. En cas d'échec lors de l'appel à l'API ou de problèmes de validation des données, des messages d'erreur appropriés sont renvoyés au frontend, et les exceptions sont journalisées pour faciliter le débogage.

##python

try:

response = requests.post(

HUGGINGFACE\_API\_URL,

headers={'Authorization': f'Bearer {api\_key}'},

data=image\_data

)

response.raise\_for\_status()

except requests.exceptions.RequestException as e:

# Gestion de l'erreur et renvoi d'une réponse appropriée

**Sécurité et Authentification**

Des mesures de sécurité ont été mises en place pour protéger les données des utilisateurs et assurer la confidentialité des communications :

* **Authentification Obligatoire :** L'accès aux fonctionnalités d'analyse nécessite que l'utilisateur soit authentifié. Django fournit des mécanismes robustes pour gérer l'authentification des utilisateurs.
* **Stockage Sécurisé des Clés API :** Les clés API des utilisateurs sont stockées de manière sécurisée dans la base de données, avec des contrôles d'accès appropriés pour empêcher tout accès non autorisé.
* **Communication Sécurisée via HTTPS :** Le déploiement du backend sur une plateforme sécurisée assure que toutes les communications entre le frontend, le backend et le service d'analyse sont chiffrées.
* **Validation des Données :** Toutes les données reçues du frontend sont soigneusement validées pour prévenir les attaques courantes telles que l'injection SQL ou les scripts intersites (XSS).

**Parallèle avec le TP2 : Communication Inter-processus et Sécurité**

L'implémentation du backend reflète les principes de **communication inter-processus** et de **sécurité** abordés dans le TP2. En agissant comme intermédiaire entre le client et le service d'analyse externe, le backend gère efficacement les **appels de procédures à distance (RPC)** et assure la **synchronisation** des opérations distribuées.

La gestion des utilisateurs et la sécurisation des communications mettent en pratique les concepts de **gestion des ressources partagées** et de **contrôle d'accès**, essentiels dans les systèmes distribués pour maintenir l'intégrité et la confidentialité des données.

**Gestion de la File d'Attente et Concurrence**

Pour gérer les requêtes entrantes de manière efficace, le backend implémente une file d'attente FIFO, similaire à celle étudiée dans le TP2. Cela permet de traiter les demandes des utilisateurs de manière séquentielle ou en parallèle, tout en évitant les conflits et les surcharges du service d'analyse.

##python

import asyncio

from collections import deque

request\_queue = deque()

async def process\_requests():

while True:

if request\_queue:

image\_data, api\_key = request\_queue.popleft()

await call\_analysis\_service(image\_data, api\_key)

await asyncio.sleep(0.1)

Cette approche assure une **gestion optimale des ressources**, en équilibrant la charge et en prévenant les goulots d'étranglement.

**CONTAINERISATION ET DEPLOIEMENT**

La containerisation et le déploiement sont des étapes essentielles pour assurer la portabilité, la scalabilité et la fiabilité de notre système distribué. En utilisant **Docker** pour la containerisation et **Railway** pour le déploiement, nous avons pu isoler les différents composants de notre application, faciliter la gestion des dépendances et déployer le système sur une plateforme cloud avec une configuration minimale.

**Containerisation avec Docker**

La containerisation avec Docker nous permet d'encapsuler le frontend et le backend dans des conteneurs séparés, garantissant ainsi que chaque composant dispose de son environnement d'exécution propre et cohérent. Cela reflète les principes d'**isolation des services** et de **gestion des ressources distribuées** étudiés dans le TP2.

**Configuration du Dockerfile pour le Backend**

Pour le backend développé avec Django, le Dockerfile définit l'image de base, installe les dépendances requises et lance le serveur d'application.

dockerfile

# Dockerfile pour le Backend

FROM python:3.9-slim

# Définition du répertoire de travail

WORKDIR /app

# Copie des fichiers de dépendances

COPY requirements.txt .

# Installation des dépendances

RUN pip install --no-cache-dir -r requirements.txt

# Copie du code source

COPY . .

# Exposition du port

EXPOSE 8000

# Commande de lancement

CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]

**Configuration du Dockerfile pour le Frontend**

Le frontend étant une application web statique utilisant HTML, CSS et JavaScript, le déploiement est plus simple. Toutefois, si nous utilisons un serveur pour servir les fichiers statiques, un Dockerfile peut être créé.

dockerfile

# Dockerfile pour le Frontend

FROM nginx:alpine

# Copie des fichiers du frontend

COPY . /usr/share/nginx/html

# Exposition du port

EXPOSE 80

# Lancement de Nginx

CMD ["nginx", "-g", "daemon off;"]

**Docker Compose pour l'Orchestration**

Pour orchestrer le déploiement des conteneurs du frontend et du backend, nous utilisons **Docker Compose**. Cela permet de définir et de gérer plusieurs conteneurs en tant que services interconnectés, reflétant la **configuration du réseau distribué**.

yaml

# docker-compose.yml

version: '3'

services:

backend:

build: ./backend

ports:

- "8000:8000"

networks:

- app-network

frontend:

build: ./frontend

ports:

- "80:80"

networks:

- app-network

networks:

app-network:

**Déploiement sur Railway**

**Railway** est une plateforme cloud qui simplifie le déploiement d'applications conteneurisées. En déployant notre application sur Railway, nous bénéficions de la scalabilité automatique, de la gestion simplifiée des ressources et de la surveillance intégrée.

**Configuration du Déploiement**

Pour déployer sur Railway, nous procédons comme suit :

1. **Intégration avec le Référentiel GitHub** : Railway peut être connecté à notre dépôt GitHub pour automatiser le déploiement à chaque nouveau commit.
2. **Définition des Services** : Dans le tableau de bord Railway, nous définissons deux services distincts pour le frontend et le backend, en spécifiant les Dockerfile correspondants.
3. **Configuration des Variables d'Environnement** : Les clés API et autres configurations sensibles sont gérées via les variables d'environnement sécurisées de Railway.
4. **Exposition des Ports** : Nous nous assurons que les ports exposés dans les Dockerfile correspondent aux ports configurés sur Railway.

**Avantages du Déploiement sur Railway**

* **Scalabilité Automatique** : Railway ajuste automatiquement les ressources en fonction de la charge, ce qui est essentiel pour un système distribué.
* **Gestion Simplifiée des Services** : La plateforme permet de gérer facilement les différents services, reflétant la **gestion des ressources distribuées** étudiée dans le TP2.
* **Surveillance et Logs Intégrés** : Railway fournit des outils de surveillance et de journalisation qui facilitent le suivi des performances et le débogage.

**Parallèles avec le TP2**

La containerisation et le déploiement de notre application illustrent plusieurs concepts clés des systèmes distribués :

* **Isolation des Services** : Chaque composant de l'application est isolé dans son propre conteneur, similaire à la séparation des processus dans une baie de serveurs.
* **Gestion des Ressources Distribuées** : L'utilisation de Docker et de Railway nous permet de gérer les ressources de manière efficace, en allouant dynamiquement la puissance de calcul et la mémoire en fonction des besoins.
* **Scalabilité et Flexibilité** : Le système peut être facilement étendu en ajoutant plus de conteneurs ou en ajustant les configurations, ce qui est essentiel pour les systèmes distribués à grande échelle.
* **Configuration du Réseau Distribué** : Docker Compose facilite la configuration du réseau entre les conteneurs, assurant une communication fluide entre le frontend et le backend, similaire à la configuration réseau dans les systèmes distribués.

**Sécurité et Fiabilité**

En déployant notre application sur Railway avec des conteneurs Docker, nous bénéficions également de :

* **Mises à Jour Simplifiées** : Les conteneurs peuvent être mis à jour indépendamment, réduisant les temps d'arrêt et améliorant la fiabilité.
* **Gestion des Versions** : Les images Docker peuvent être versionnées, permettant de revenir à une version précédente en cas de problème.
* **Sécurité Renforcée** : L'isolation des conteneurs limite la surface d'attaque potentielle, améliorant ainsi la sécurité globale du système.

**CODE SOURCE ET INSTRUCTIONS DE DEPLOIEMENT**

Le code source complet du projet est disponible sur GitHub à l'adresse suivante : <https://github.com/josephsamijona/ReconnaissanceObjetWebcam.git>. Vous pouvez cloner ce dépôt pour examiner le code en détail ou pour exécuter l'application sur votre propre machine.

**Instructions pour Lancer le Projet**

**Prérequis**

* **Python 3.10** : Assurez-vous d'avoir Python installé sur votre machine.
* **Git** : Pour cloner le dépôt GitHub.
* **Docker** (optionnel) : Pour exécuter l'application dans des conteneurs.
* **Clé API Hugging Face** : Nécessaire pour accéder au service d'analyse d'images.

**Étapes pour Exécuter le Projet**

1. **Cloner le Dépôt GitHub**

Ouvrez votre terminal et exécutez la commande suivante :

git clone https://github.com/josephsamijona/ReconnaissanceObjetWebcam.git

1. **Naviguer dans le Répertoire du Projet**

cd ReconnaissanceObjetWebcam

1. **Créer un Environnement Virtuel**

Il est recommandé d'utiliser un environnement virtuel pour isoler les dépendances du projet.

python -m venv env

source env/bin/activate # Sous Windows : env\Scripts\activate

1. **Installer les Dépendances**

pip install -r requirements.txt

1. **Configurer la Clé API Hugging Face**

Le projet nécessite une clé API Hugging Face pour fonctionner. Comme la clé API n'est pas incluse dans le code source pour des raisons de sécurité, vous devez fournir la vôtre.

**Obtention de la Clé API :**

* + Rendez-vous sur le site de [Hugging Face](https://huggingface.co/).
  + Créez un compte si vous n'en avez pas déjà un.
  + Accédez à vos paramètres de profil et générez une **Access Token** avec les permissions appropriées.

**Configurer la Clé API dans le Projet :**

* + Créez un fichier .env à la racine du projet.
  + Ajoutez-y la ligne suivante :

.env

HUGGINGFACE\_API\_KEY=VotreCléAPIIci

* + Assurez-vous de ne pas commettre ce fichier dans le contrôle de version.

1. **Appliquer les Migrations de la Base de Données**

python manage.py migrate

1. **Créer un Superutilisateur (Optionnel)**

Si vous souhaitez accéder à l'interface d'administration de Django :

python manage.py createsuperuser

1. **Lancer le Serveur de Développement**

python manage.py runserver

1. **Accéder à l'Application**

Ouvrez votre navigateur web et rendez-vous sur http://localhost:8000 pour utiliser l'application.

**Utilisation de Docker (Optionnel)**

Si vous préférez exécuter l'application dans des conteneurs Docker, assurez-vous que Docker est installé et suivez les étapes suivantes :

1. **Construire les Images Docker**

docker-compose build

1. **Démarrer les Conteneurs**

docker-compose up

1. **Accéder à l'Application**

Ouvrez votre navigateur web et rendez-vous sur http://localhost (ou le port spécifié dans le docker-compose.yml).

**Remarques Importantes**

* **Clé API Non Incluse :** Pour des raisons de sécurité, la clé API Hugging Face n'est pas incluse dans le code source. Vous devez fournir votre propre clé API comme indiqué ci-dessus.
* **Fournisseur du Service d'Analyse :** Nous utilisons l'API de Hugging Face pour l'analyse des images. Assurez-vous de respecter les conditions d'utilisation de leur service.
* **Mise en Ligne du Projet :** Le projet est déployé en ligne et accessible à l'adresse suivante : [URL du Projet en Ligne](https://adresse-du-projet-en-ligne.com). Vous pouvez tester l'application sans avoir à l'installer localement.

**CONCLUSION**

Ce projet a permis de mettre en œuvre les principes des systèmes distribués à travers un système de reconnaissance d'objets en temps réel. En combinant un frontend interactif, un backend robuste et un service d'analyse basé sur une API externe, nous avons démontré comment les concepts théoriques peuvent être appliqués pour répondre à des besoins réels. L'intégration de Docker et le déploiement sur Railway ont renforcé la scalabilité et la portabilité du système, tout en assurant une gestion efficace des ressources. Ce travail met en avant l'importance de la communication inter-services, de la sécurité et de la performance dans la conception de solutions distribuées modernes.