Design Patterns MINI PROJET

Realiser par: Yassine KADER / Samira AFIZI

Contexte:

À mesure que votre application gagne en popularité, la base d'utilisateurs et le trafic vers votre service API augmentent de manière significative. Cependant, vous commencez à remarquer un goulot d'étranglement de performance alors que le système peine à gérer le nombre croissant de requêtes. Cela peut entraîner des temps de réponse lents, une latence accrue, voire des interruptions de service pendant les périodes d'utilisation intense.

Description du Problème :

L'architecture monolithique existante et la conception de la base de données pourraient ne pas être en mesure de s'adapter horizontalement à la demande croissante. En conséquence, le service API rencontre des problèmes de scalabilité, entravant l'expérience utilisateur et pouvant causer des perturbations dans le service, parmi les problemes:

1. **Latence élevée :** Les utilisateurs subissent des retards dans les temps de réponse, entraînant une mauvaise expérience utilisateur.

2. **Taux d'erreurs accru :** Le système peut commencer à renvoyer des erreurs ou des délais en raison du nombre écrasant de requêtes entrantes.
3. **Épuisement des ressources :** L'infrastructure serveur peut atteindre ses limites, provoquant un épuisement des ressources et des plantages potentiels.

Example:

Avant utiliser **les Designs patterns (Monolithic Architecture (Avec Flask et Python)):**

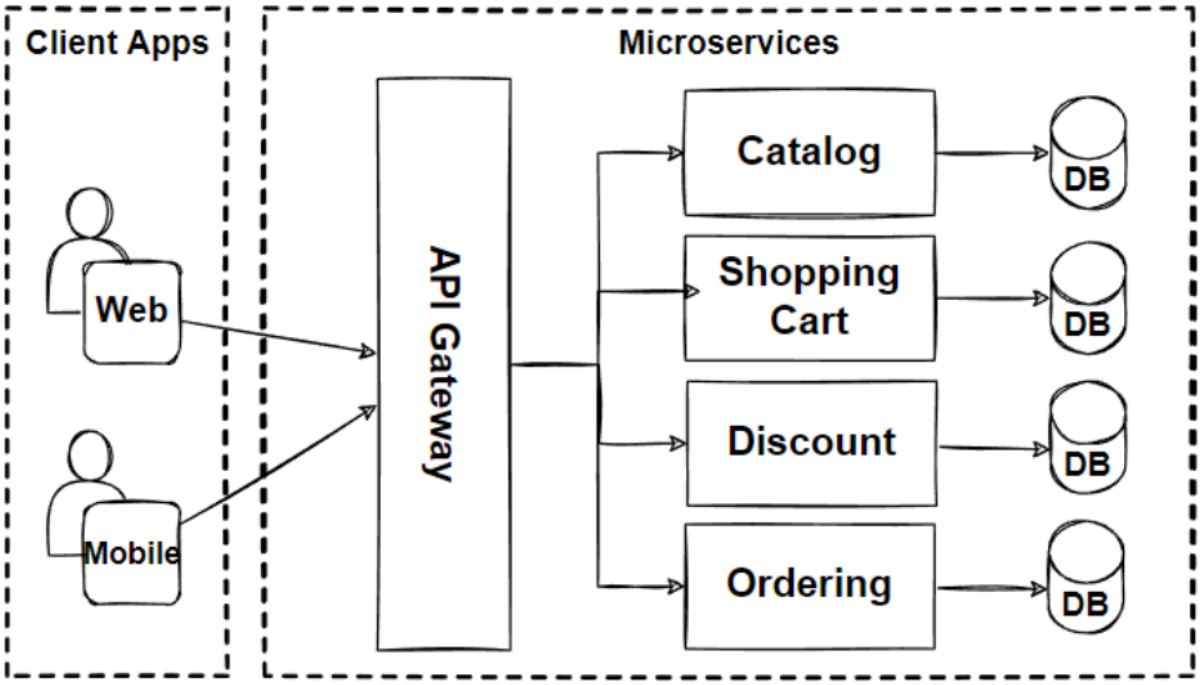
```
# Monolithic Application
from flask import Flask, jsonify

app_monolith = Flask(__name__)

@app_monolith.route('/api/<user_id>/<product_id>')
def get_user_and_product_monolith(user_id, product_id):
    # Simulate fetching user data from a monolithic database
    user_data = {
        'user_id': user_id,
        'username': f'user_{user_id}',
        'email': f'user_{user_id}@example.com'
    }

    # Simulate fetching product data from a monolithic databa
    product_data = {
        'product_id': product_id,
        'name': f'Product_{product_id}',
        'price': 19.99
    }

    # Combine user and product data
    result = {
        'user': user_data,
        'product': product_data
    }

    return jsonify(result)
```

```
if __name__ == '__main__':
    app_monolith.run(port=5000)
```

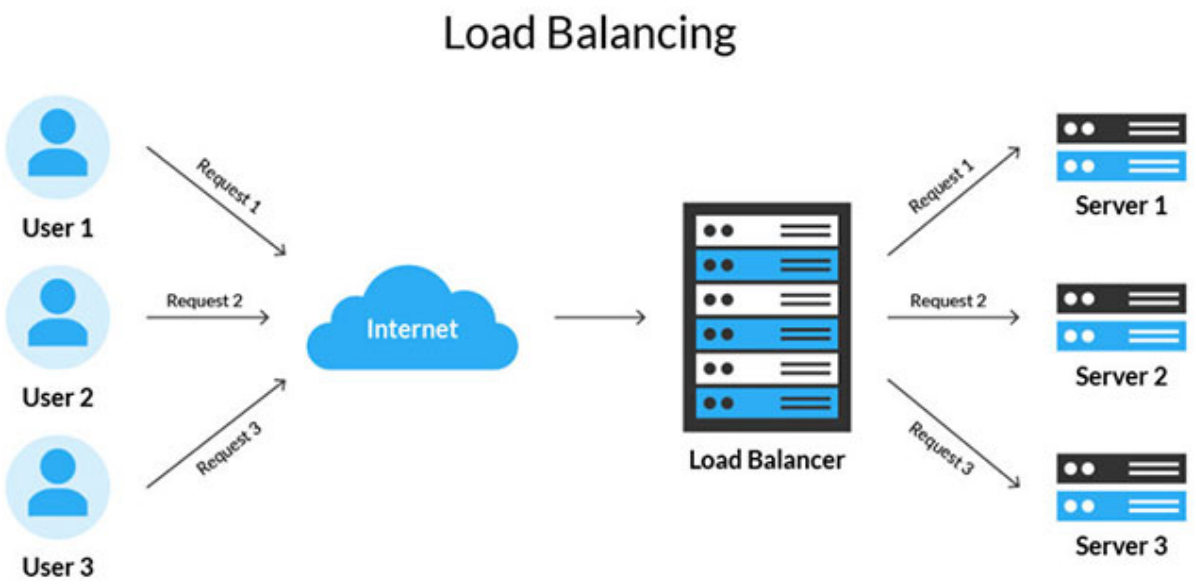
Solution (Architecture de Microservices avec API Gateway):

Patterns de Conception :

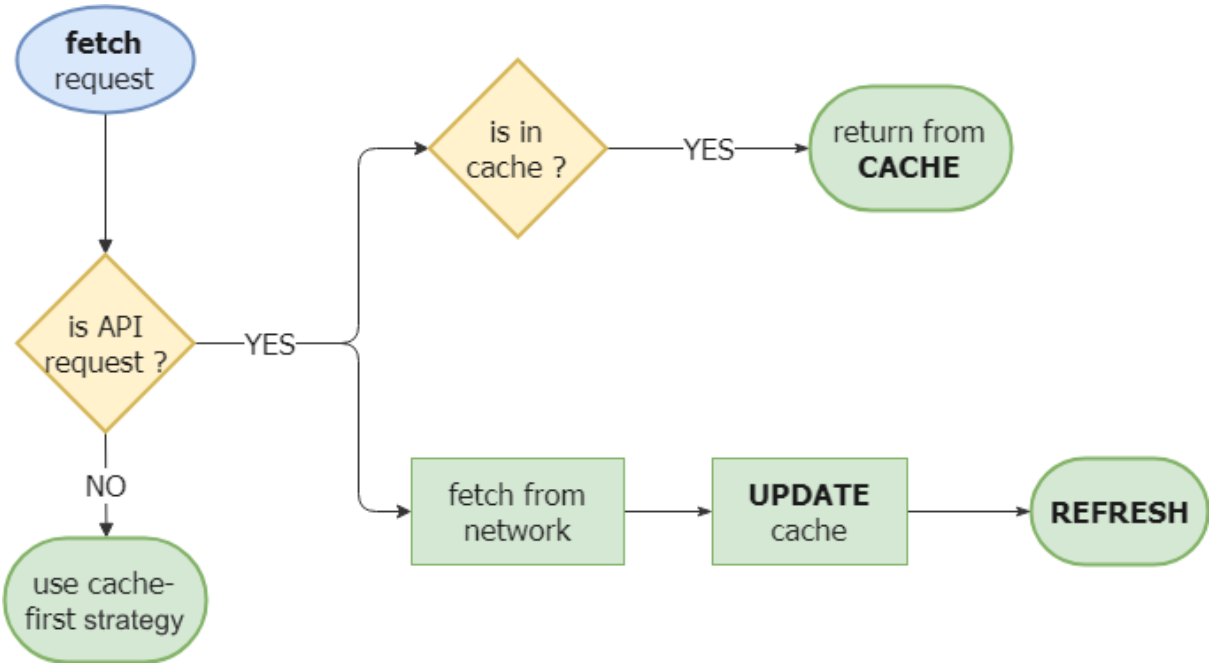
- `Microservices Architecture` **:** Divisez l'application monolithique en services plus petits, indépendants, pouvant être développés, déployés et mis à l'échelle indépendamment. Chaque microservice gère une fonctionnalité métier spécifique, permettant une meilleure scalabilité et maintenabilité.
- `API Gateway` **:** Implémentez une passerelle API pour gérer et router les requêtes entrantes vers les microservices appropriés. La passerelle API agit comme un point d'entrée unique pour les clients, gérant des tâches telles que l'authentification, la limitation de taux et l'équilibrage de charge.



- `Équilibrage de Charge` (Load Balancing) **:** Intégrez des mécanismes d'équilibrage de charge pour distribuer le trafic entrant sur plusieurs instances de chaque microservice. Cela garantit qu'aucun service unique ne devient un goulot d'étranglement et que la charge est répartie uniformément.

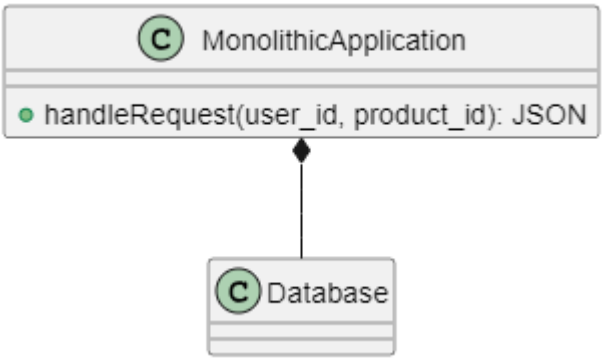


- `Stratégie de Mise en Cache` **:** Implémentez une stratégie de mise en cache pour réduire la charge sur la base de données et améliorer les temps de réponse des opérations fréquemment consultées.

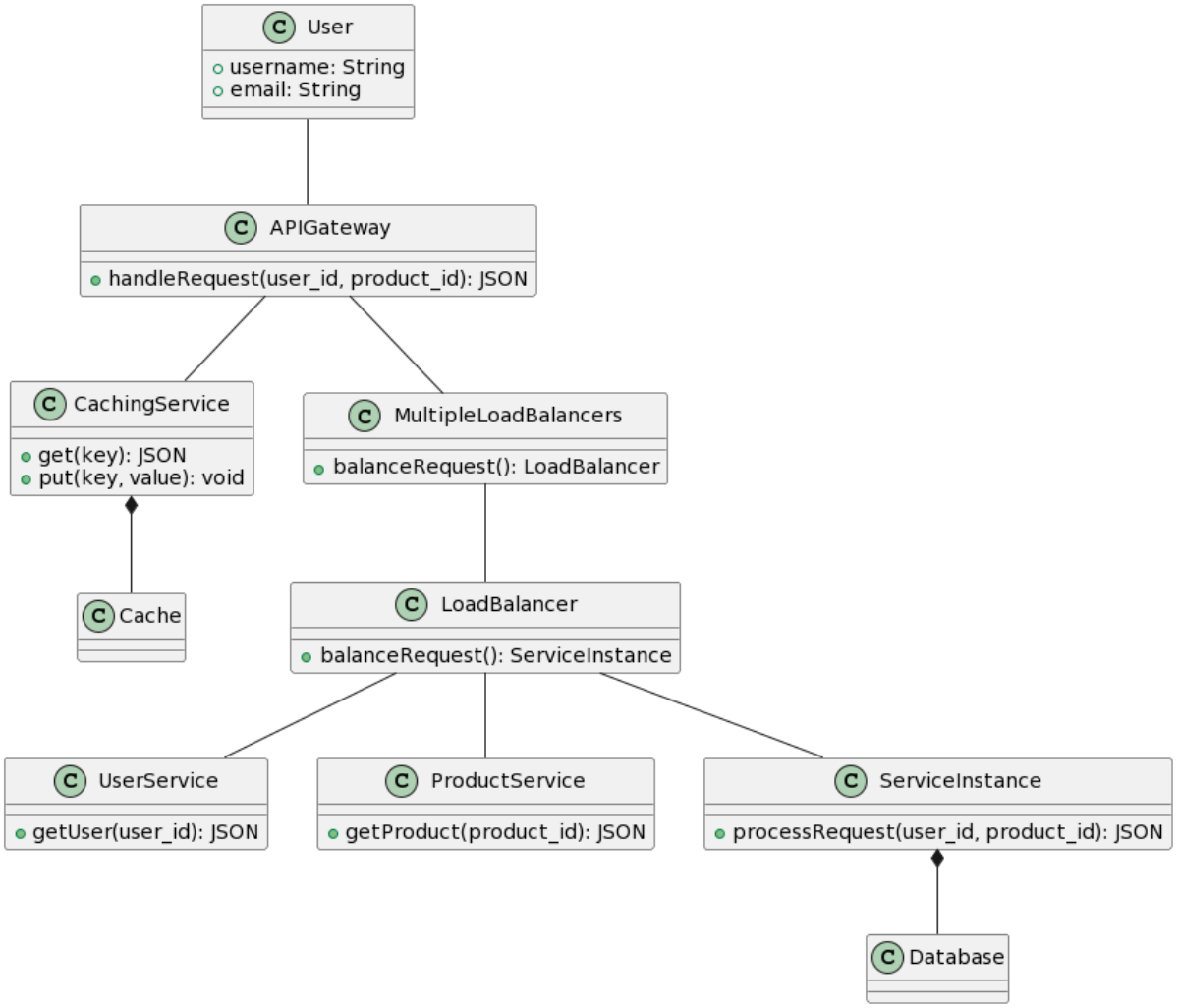


Implementations des solution:

Avant Implementation:



Apres Implementation:




```
//plant Uml code for the oslution diagrame
@startuml
class User {
  +username: String
```

```
  +email: String
}

class APIGateway {
  +handleRequest(user_id, product_id): JSON
}

class UserService {
  +getUser(user_id): JSON
}

class ProductService {
  +getProduct(product_id): JSON
}

class LoadBalancer {
  +balanceRequest(): ServiceInstance
}

class CachingService {
  +get(key): JSON
  +put(key, value): void
}

class ServiceInstance {
  +processRequest(user_id, product_id): JSON
}

class MultipleLoadBalancers {
  +balanceRequest(): LoadBalancer
}

APIGateway -- MultipleLoadBalancers
APIGateway -- CachingService
LoadBalancer -- UserService
LoadBalancer -- ProductService

LoadBalancer -- ServiceInstance
```

```
ServiceInstance *-- Database
CachingService *-- Cache
User -- APIGateway
MultipleLoadBalancers -- LoadBalancer
@enduml
```

Explication De Solution:

1. **User (Utilisateur) :**
   - La classe `User` représente les informations relatives à un utilisateur, telles que le nom d'utilisateur (`username`) et l'adresse e-mail (`email`).
2. **APIGateway (Passerelle API) :**
   - La classe `APIGateway` représente le point d'entrée principal pour les requêtes provenant des utilisateurs. Elle est responsable de la gestion des demandes vers les services appropriés.
3. **UserService (Service Utilisateur) et ProductService (Service Produit) :**
   - Les classes `UserService` et `ProductService` sont des microservices qui fournissent respectivement des informations sur les utilisateurs et les produits.
4. **LoadBalancer (Équilibreur de Charge) :**
   - La classe `LoadBalancer` est responsable de la répartition des requêtes entre plusieurs instances des services, assurant une utilisation équilibrée des ressources.
5. **CachingService (Service de Mise en Cache) :**
   - La classe `CachingService` représente un mécanisme de mise en cache, stockant temporairement des données fréquemment utilisées pour améliorer les temps de réponse.
6. **ServiceInstance (Instance de Service) :**
   - La classe `ServiceInstance` représente une instance spécifique d'un microservice (`UserService` ou `ProductService`). Elle traite les requêtes et interagit avec la base de données.
7. **MultipleLoadBalancers (Plusieurs Équilibreurs de Charge) :**
   - La classe `MultipleLoadBalancers` symbolise la possibilité d'avoir plusieurs équilibreurs de charge, chacun pouvant répartir les requêtes vers différentes

instances des services.

Relations :

- `APIGateway` communique avec `MultipleLoadBalancers` pour diriger les requêtes.
- Les services (`UserService` et `ProductService`) sont connectés à l'équilibreur de charge (`LoadBalancer`), indiquant que le répartiteur de charge est responsable de la distribution des requêtes entre les instances.
- `LoadBalancer` communique avec `ServiceInstance` pour acheminer les requêtes vers une instance spécifique du service.
- `CachingService` est utilisé par `APIGateway` pour améliorer les performances en stockant temporairement des données fréquemment consultées.
- La classe `User` est connectée à `APIGateway`, symbolisant que les demandes proviennent directement des utilisateurs vers la passerelle API.

Comment peut-on implémenter cette solution ?

Microservices:

- **Description :** Ce modèle de conception consiste à diviser une application monolithique en services plus petits et indépendants. Chaque service est responsable d'une fonctionnalité métier spécifique et peut être développé, déployé et mis à l'échelle indépendamment.
- **Implémentation :** Les services `Utilisateur` et `Produit` représentent des microservices indépendants.

```
#User service
from flask import Flask, jsonify

app_user = Flask(__name__)

@app_user.route('/user/<user_id>')
def get_user(user_id):
    # Simuler la récupération des données utilisateur depu
    user_data = {
        'user_id': user_id,
```

```
        'username': f'user_{user_id}',
        'email': f'user_{user_id}@example.com'
    }
    return jsonify(user_data)

if __name__ == '__main__':
    app_user.run(port=5001)
```

```
#Product service
from flask import Flask, jsonify

app_product = Flask(__name__)

@app_product.route('/product/<product_id>')
def get_product(product_id):
    product_data = {
        'product_id': product_id,
        'name': f'Produit_{product_id}',
        'price': 19.99
    }
    return jsonify(product_data)

if __name__ == '__main__':
    app_product.run(port=5002)
```

Passerelle API(Api Gateway) :

- **Description :** La `passerelle API` agit comme un point d'entrée unique pour les clients, gérant des tâches telles que le routage des requêtes vers les microservices appropriés, la gestion de l'authentification et la fourniture d'une interface unifiée.
- **Implémentation :** Le service de `la passerelle API` route les requêtes vers les services Utilisateur et Produit.

```
# Api Gateway
import requests

app_gateway = Flask(__name__)
```

```
USER_SERVICE_URL = 'http://localhost:5001/user/'
PRODUCT_SERVICE_URL = 'http://localhost:5002/product/'

@app_gateway.route('/api/<user_id>/<product_id>')
def get_user_and_product(user_id, product_id):
    user_response = requests.get(f'{USER_SERVICE_URL}{user
    user_data = user_response.json()

    product_response = requests.get(f'{PRODUCT_SERVICE_URL
    product_data = product_response.json()

    result = {
        'user': user_data,
        'product': product_data
    }

    return jsonify(result)

if __name__ == '__main__':
    app_gateway.run(port=5000)
```

Load Balancing(Nginx) :

Nginx est un serveur web et un serveur proxy inversé largement utilisé. Il est réputé pour ses performances élevées, sa scalabilité et sa capacité à gérer efficacement les connexions simultanées.

Implémentation :

Dans l'architecture de microservices, Nginx peut être utilisé comme équilibreur de charge pour répartir le trafic entrant sur plusieurs instances des services `Utilisateur` et `Produit`. Voici un exemple de fichier de configuration Nginx (nginx.conf) pour l'équilibrage de charge :

```
http {
    upstream user_service {
        server localhost:5001;
        server localhost:5002;
                server localhost:5003;
                server localhost:5004;
                server localhost:5005;
    }

    upstream product_service {
        server localhost:5006;
        server localhost:5007;
                server localhost:5008;
                server localhost:5009;
                server localhost:5010;
    }

    server {
        listen 80;

        location /api/user/ {
            #Algrithm used for load balancing is "Round Ro
            proxy_pass http://user_service;
        }

        location /api/product/ {
            #Algrithm used for load balancing is "Round Ro
            proxy_pass http://product_service;
        }
    }
}
```

Dans cet configuration :

- `upstream` définit un groupe de serveurs (instances des services Utilisateur et Produit).
- La directive `proxy_pass` route les requêtes vers l'un des serveurs du groupe en amont.

Cashing Service(redis):

Présentation de Redis :

Redis est un magasin de données en mémoire utilisé comme couche de mise en cache. Il prend en charge diverses structures de données telles que les chaînes, les hachages, les listes, etc.

Implémentation :
Dans l'architecture de microservices, Redis peut être utilisé pour mettre en cache les données fréquemment consultées, réduisant ainsi la charge sur la base de données. Voici un exemple simple de l'intégration de la mise en cache Redis dans le Service Utilisateur :

```
from flask import Flask, jsonify
import redis
import json

app_user = Flask(__name__)
redis_client = redis.StrictRedis(host='localhost', port=63

@app_user.route('/user/<user_id>')
def get_user(user_id):
    cached_user_data = redis_client.get(f'user:{user_id}')

    if cached_user_data:
        user_data = json.loads(cached_user_data)
    else:
        user_data = {
            'user_id': user_id,
            'username': f'user_{user_id}',
            'email': f'user_{user_id}@example.com'
        }
        redis_client.set(f'user:{user_id}', json.dumps(use
```

```
    return jsonify(user_data)

#if __name__ == '__main__':
#    app_user.run(port=5001)
```

Dans cet exemple :

- `redis.StrictRedis` se connecte à un serveur Redis en cours d'exécution en local.
- Avant de récupérer les données de la base de données, le service vérifie si les données sont déjà dans le cache Redis (méthode `get`).
- Si les données sont dans le cache, elles sont renvoyées. Sinon, les données sont récupérées de la base de données, puis stockées dans le cache pour les requêtes futures (méthode `set`).

Remarque importante : Les exemples précédents sont des démonstrations simplifiées à des fins pédagogiques. Dans des situations réelles, le développement d'API ou de backends implique des considérations plus vastes, notamment l'utilisation de bases de données, la gestion des problèmes de sécurité, la mise en œuvre de stratégies de sauvegarde, etc.

Complexité du Monde Réel :
Dans des environnements de production réels, les développeurs doivent faire face à divers défis, tels que :

1. **Bases de Données :** L'utilisation de bases de données relationnelles ou NoSQL pour stocker et gérer les données de manière efficace.
2. **Sécurité :** La mise en œuvre de mesures de sécurité robustes, y compris l'authentification, l'autorisation, la gestion des sessions, la protection contre les attaques par injection SQL, et la sécurisation des communications.
3. **Gestion des Erreurs :** La gestion appropriée des erreurs et des exceptions, y compris la journalisation détaillée pour faciliter le débogage et la résolution des problèmes.
4. **Scalabilité Horizontale :** La mise en œuvre de la scalabilité horizontale, où de nouveaux serveurs ou instances peuvent être ajoutés pour répondre à une demande croissante.

5. **Conteneurisation :** L'utilisation de technologies de conteneurisation comme Docker pour encapsuler et déployer des applications de manière cohérente.
6. **Orchestration :** L'utilisation de Docker Compose pour gérer des applications multi-conteneurs et Kubernetes pour orchestrer et automatiser le déploiement, la mise à l'échelle et la gestion des conteneurs.
7. **Monitoring et Analyse des Performances :** La mise en place de mécanismes de surveillance pour suivre les performances de l'application, détecter les goulots d'étranglement et résoudre les problèmes de manière proactive.

Technologies Complémentaires :
En plus de Flask (ou tout autre framework web) pour le développement des API, d'autres technologies complémentaires jouent un rôle crucial dans des scénarios réels. Voici quelques-unes :

- **Docker et Docker Compose :** Pour la conteneurisation d'applications, facilitant le déploiement et la gestion des dépendances.
- **Kubernetes :** Pour l'orchestration et l'automatisation des conteneurs dans des environnements de production à grande échelle.
- **Bases de Données :** Choix de bases de données adaptées aux besoins, comme PostgreSQL, MongoDB, Redis, etc.
- **Systèmes de Gestion de Versions :** Utilisation de systèmes comme Git pour la gestion du code source et la collaboration.
- **Outils de CI/CD :** Mise en place de pipelines d'intégration continue (CI) et de déploiement continu (CD) pour automatiser les tests et le déploiement.

Conclusion:

L'adoption judicieuse d'une architecture de microservices, appuyée par des patterns de conception pertinents, a démontré son efficacité en atténuant significativement le goulot d'étranglement lié à la scalabilité. Cette transition garantit que le service API est désormais en mesure de gérer des charges en constante augmentation tout en offrant des performances optimisées et une fiabilité renforcée. Toutefois, il est impératif de souligner que la clé du succès réside dans une surveillance continue et une optimisation proactive. Ces mesures sont cruciales pour maintenir la scalabilité du système et pour réaliser des améliorations continues, particulièrement à mesure que l'application

continue son expansion. En définitive, cette évolution vers une architecture de microservices représente une réponse stratégique aux défis de croissance, établissant une base solide pour soutenir le développement futur de l'application avec agilité et robustesse.