



Monitoring des Architectures à base de Microservices : Télémétrie et Reconfiguration

Mohammed Dafaoui
Adam Daia
Aness Rabia



Plan

- I / Mise en contexte et problématique
- II / Outils utilisés
- III / Mise en place du Monitoring
- IV / Conclusion



Mise en contexte

01

Introduction aux architectures
logicielles

02

Problématique

03

Observabilité et télémétrie

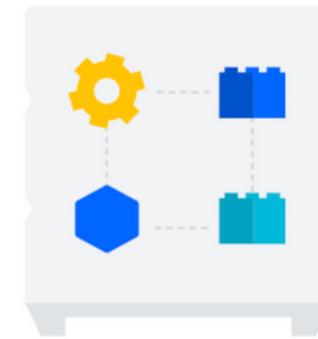
04

Qualité de service

05

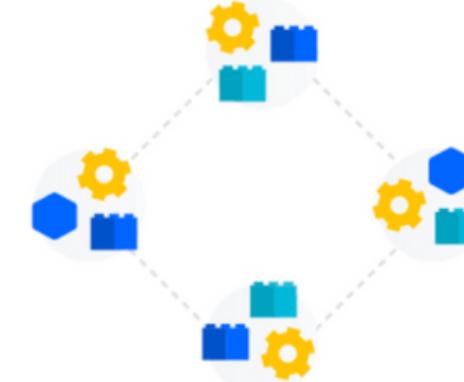
État de l'art

Les architectures



Monolith

- Conception traditionnelle
- Bloc de code indivisible
- Entité unique
- Pas de séparation de composants
- Fonctionnalités rassemblées
- Interfaces utilisateur
- Traitements métier
- Accès aux données

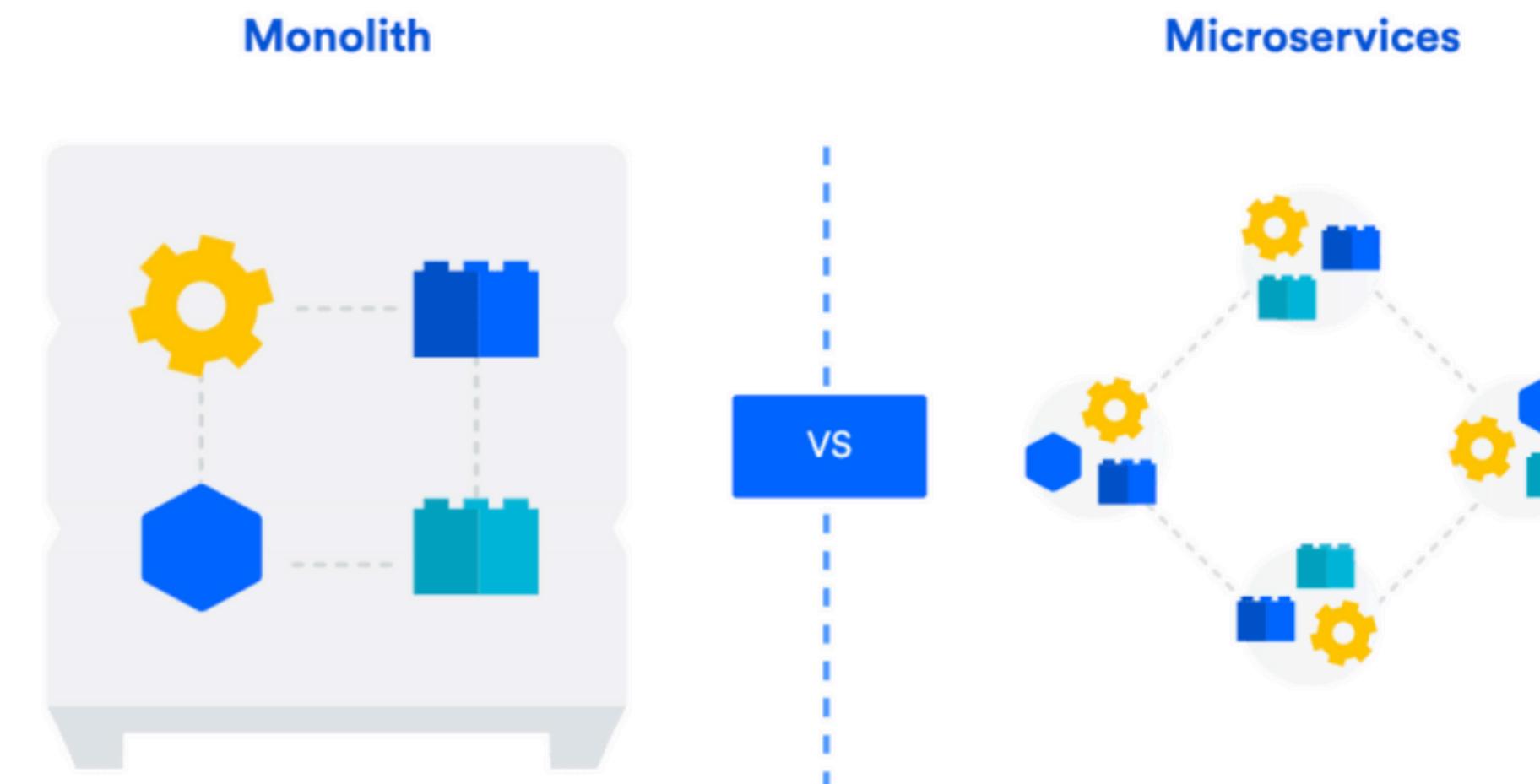


Microservices

- Décomposition en services autonomes
- Faible couplage
- Services indépendants
- Base de code propre
- Logique métier propre
- Base de données propre
- Fonctionnalité spécifique
- Communication via API

Problématique

Quelle architecture, monolithique ou basée sur les microservices, convient le mieux pour répondre aux exigences spécifiques d'un projet ?



Architecture Monolithique

Une architecture monolithique regroupe l'ensemble des fonctionnalités d'une application au sein d'un seul et même bloc de code indivisible. Toutes les couches sont fortement couplées et interdépendantes, formant une entité unique sans séparation de composants. Cette conception unitaire offre une simplicité de mise en œuvre initiale mais présente des limites d'évolutivité et de flexibilité.

Avantages

- Simplicité de mise en œuvre
- Gestion centralisée
- Transactions facilitées
- Déploiement en une seule unité

Inconvénients

- Évolutivité limitée
- Scalabilité difficile
- Redéploiement complet requis
- Manque de flexibilité
- Faible résilience
- Couplage serré des composants



Architecture Micro-service

Une architecture micro-services décompose l'application en services autonomes et faiblement couplés. Chaque service est indépendant avec sa propre base de code, logique métier et base de données. Il implémente une fonctionnalité métier spécifique et communique via des API.

Avantages

- Agilité accrue
- Évolutivité facilitée
- Déploiement et mise à l'échelle indépendants
- Cycles de livraison rapides
- Meilleure résilience
- Technologie polyglotte possible

Inconvénients

- Complexité de conception/gestion
- Défis de développement et tests
- Gouvernance des services
- Gestion des communications réseau
- Monitoring et observabilité cruciaux



Observabilité et télémétrie

L'observabilité fait référence à la collecte et à l'analyse de données provenant d'une grande variété de sources, dans le but de fournir des informations détaillées sur le comportement des applications.

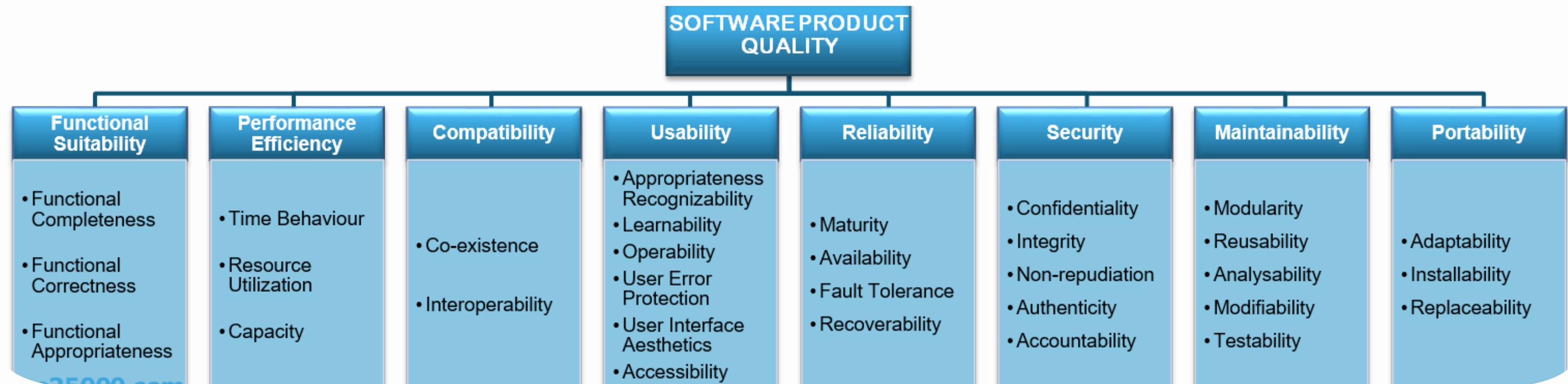


- 🔍 Monitoring efficace des systèmes distribués complexes
- 👥 Favorise la collaboration et la visibilité entre équipes
- 🚑 Permet un diagnostic et une résolution proactifs des problèmes
- ⬆️ Améliore les performances et l'expérience utilisateur
- 🔒 Renforce la sécurité des applications



Qualité de service

La qualité de service (QoS) est un concept clé qui vise à garantir que le système logiciel réponde aux exigences non fonctionnelles des utilisateurs et des parties prenantes.



Qualité de service

Avantages/Inconvénients des architectures sur la QoS

Architecture	Monolithique	Microservices
Avantages	<ul style="list-style-type: none">Gestion des transactions facilitéeCohérence des données	<ul style="list-style-type: none">Scalabilité accrueMeilleure évolutivitéPlus grande résilience
Inconvénients	<ul style="list-style-type: none">Scalabilité limitéeRésilience moindreÉvolutivité difficile	<ul style="list-style-type: none">Complexité opérationnelleCohérence des données réparties

État de l'art

1
Microservice Patterns

2
Evaluating the Performance of Monolithic and Microservices Architectures in an Edge Computing Environment

3
Monolithic vs. Microservice Architecture : Performance and Scalability

4
Stepwise Migration of a Monolith to a Microservices Architecture : Performance and Migration Effort

5
Assessing the impacts of decomposing a monolithic application for microservices

6
From monolithic systems to Microservices : An assessment framework

Outils utilisés

01

OpenTelemetry

02

Postman

03

Artillery

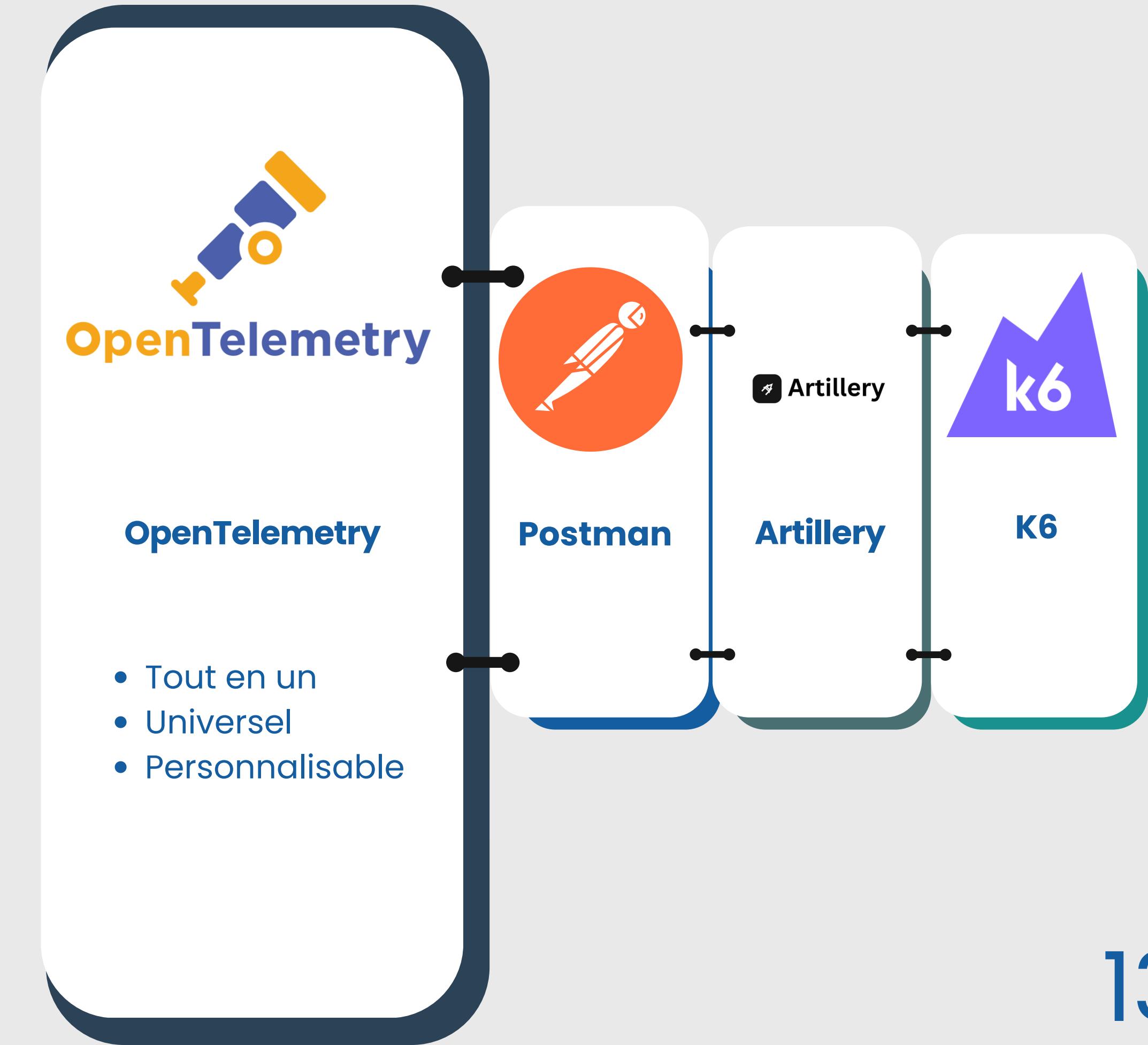
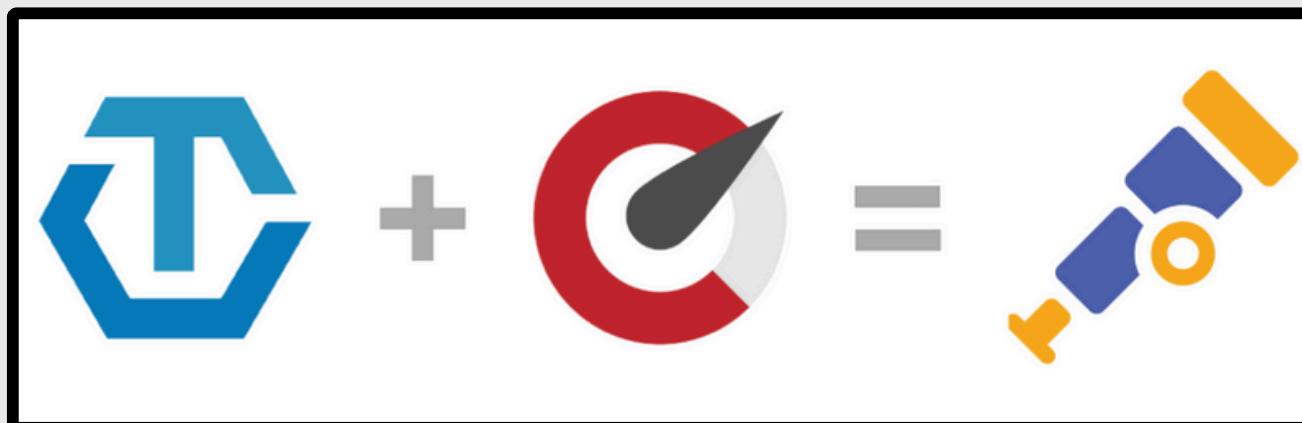
04

K6

Outils utilisés

1

Opentelemetry est une solution open source permettant de prendre en charge tout le processus de télémétrie, de la collection des données jusqu'à leur arrivée dans la base

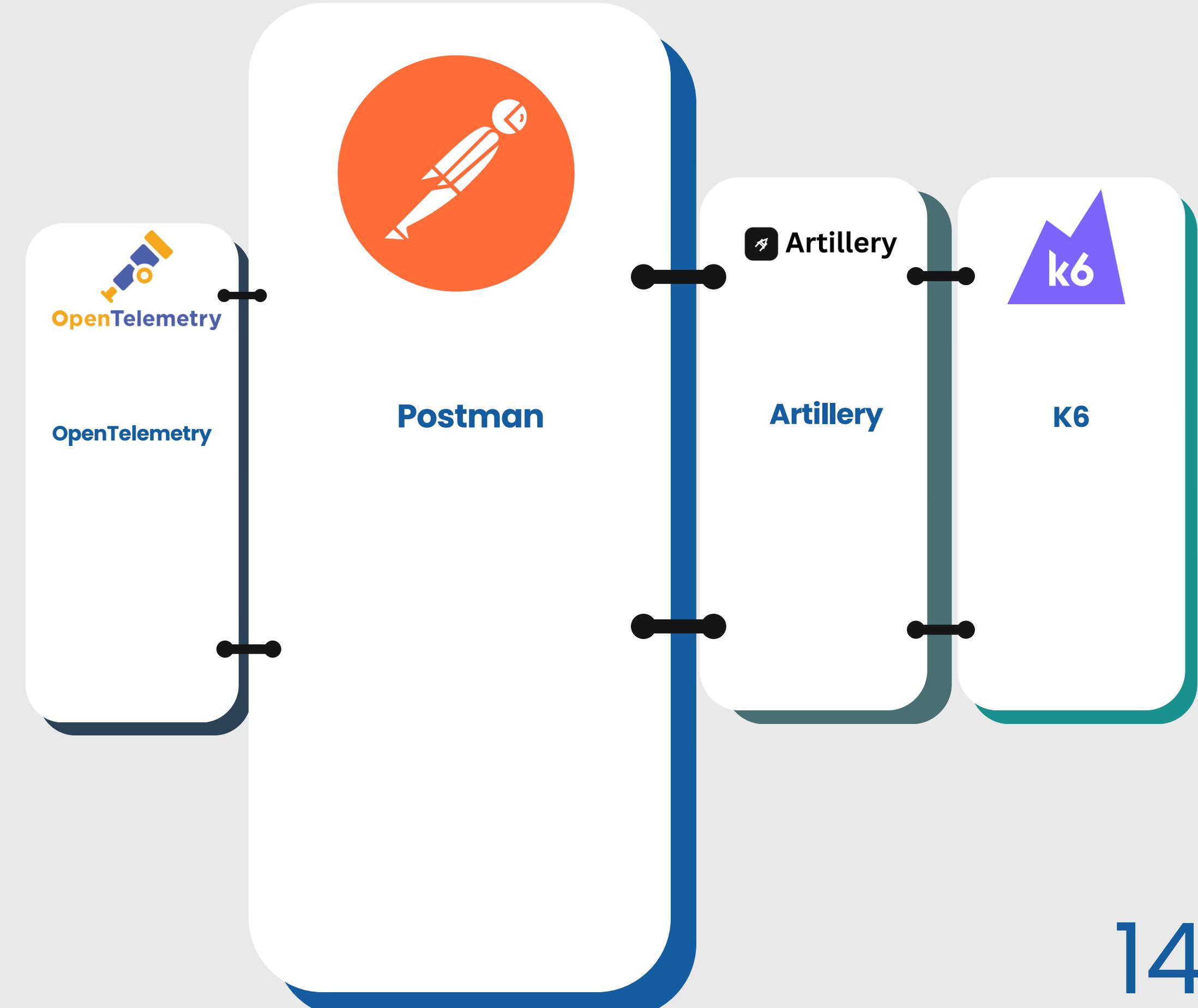


- Tout en un
- Universel
- Personnalisable

2

Outils utilisés

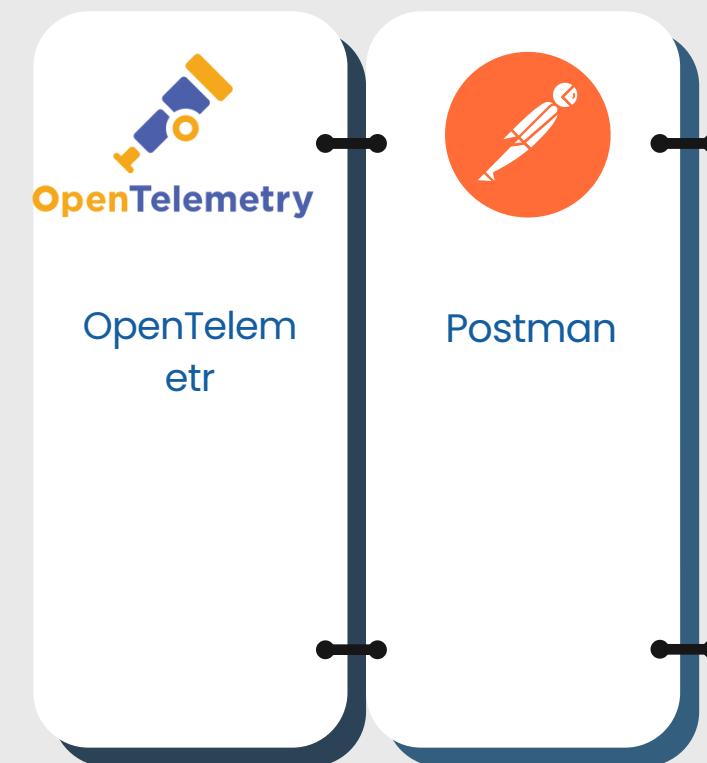
Postman, un outil de test d'API, nous a permis de valider le bon fonctionnement de nos endpoints et microservices en simulant des requêtes et en vérifiant les réponses.



3

Outils utilisés

Artillery, un outil open source de test de charge et de performance, nous a permis de simuler des charges de travail réalistes et de mesurer la réactivité et la stabilité de nos applications sous différentes conditions de charge.

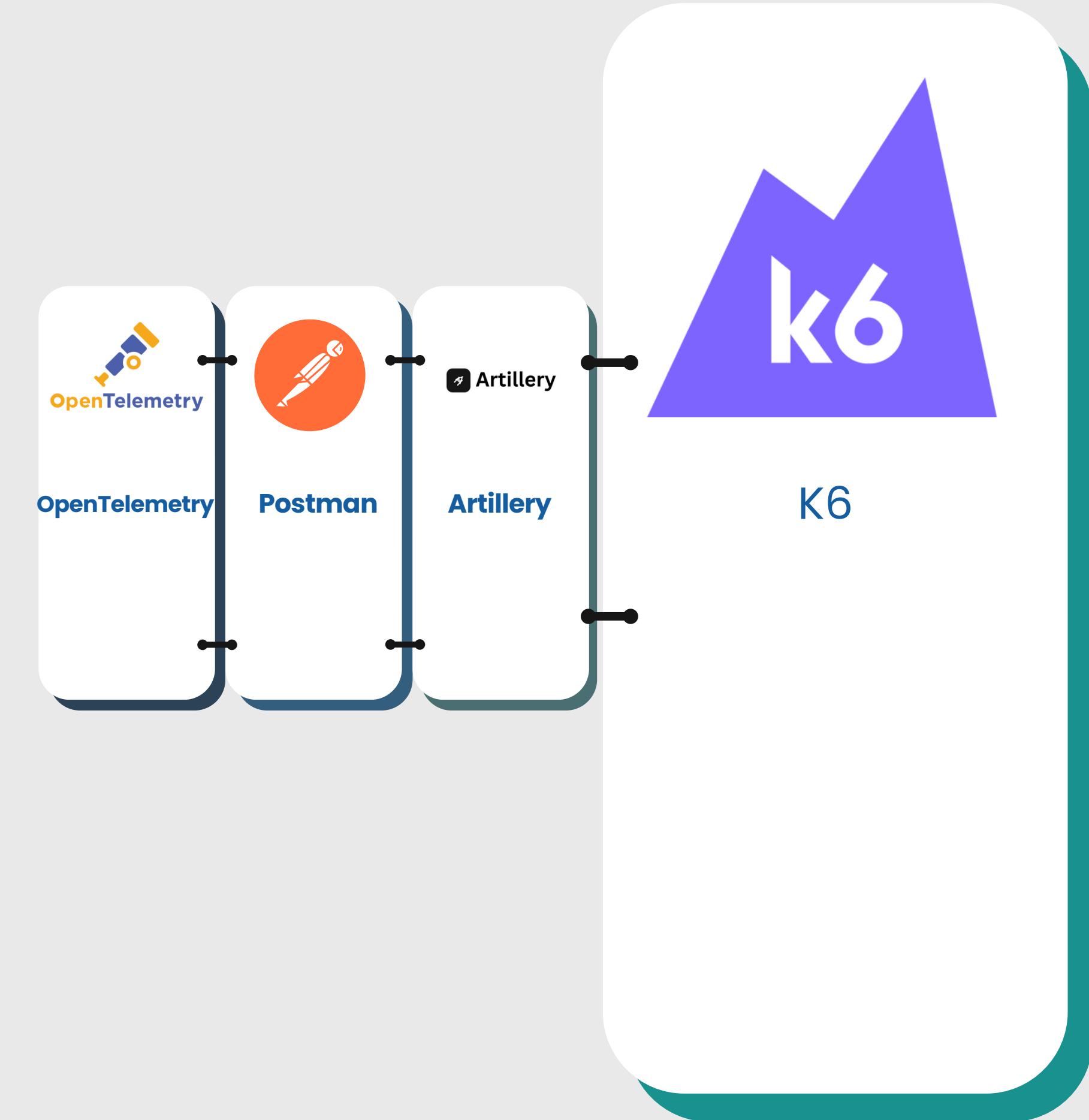


Artillery

Outils utilisés

4

K6, un outil open source de test de charge et de performance, nous permet de simuler des charges de travail réalistes et de mesurer les performances de nos applications avec précision.



Mise en place du Monitoring

01

Structure de notre application

02

Migration vers architecture
micro-service

03

Implémentation de la
télémétrie

04

Résultat des tests

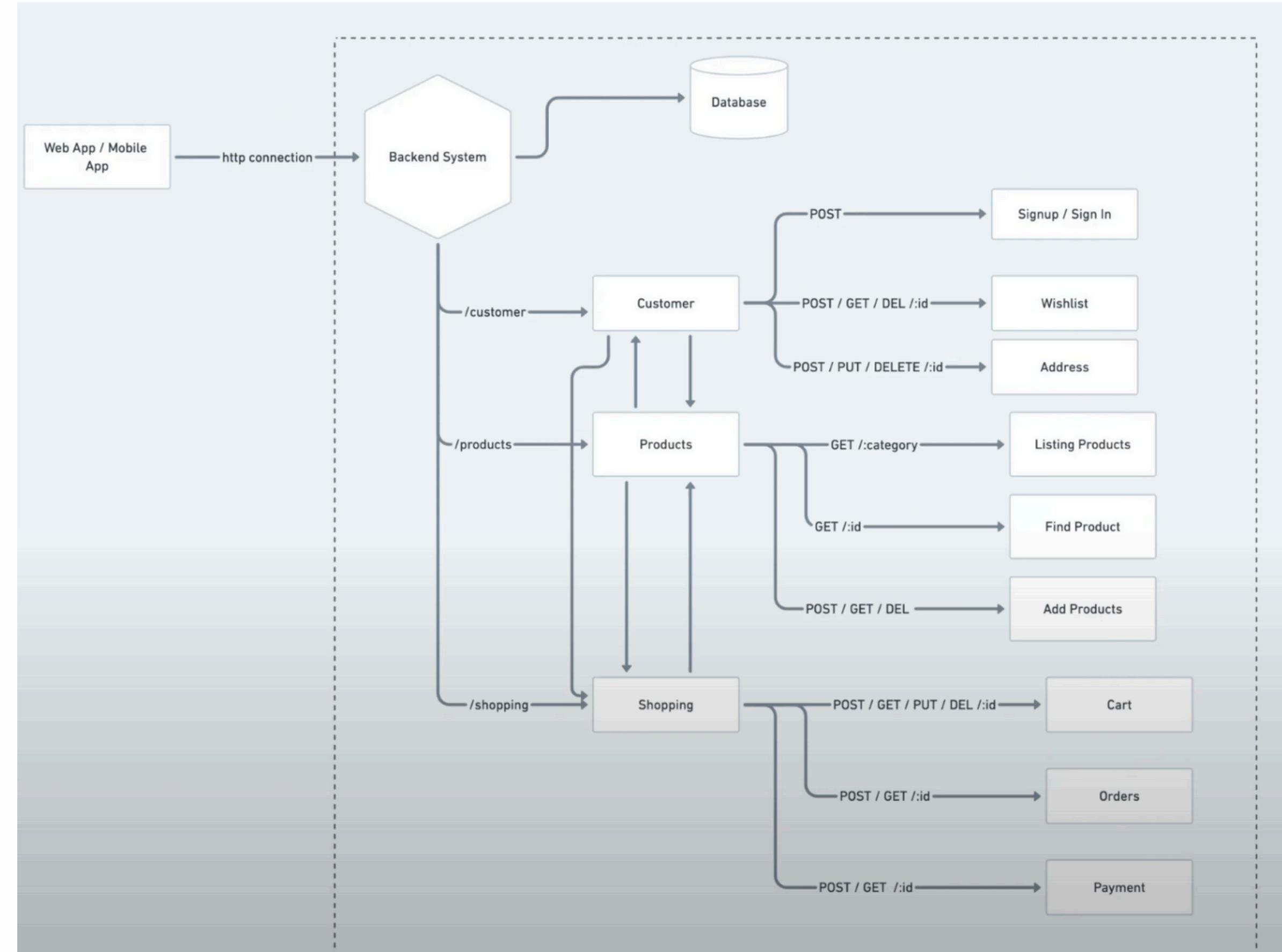
NOTRE APPLICATION

Une application d'E-commerce
qui permet de :

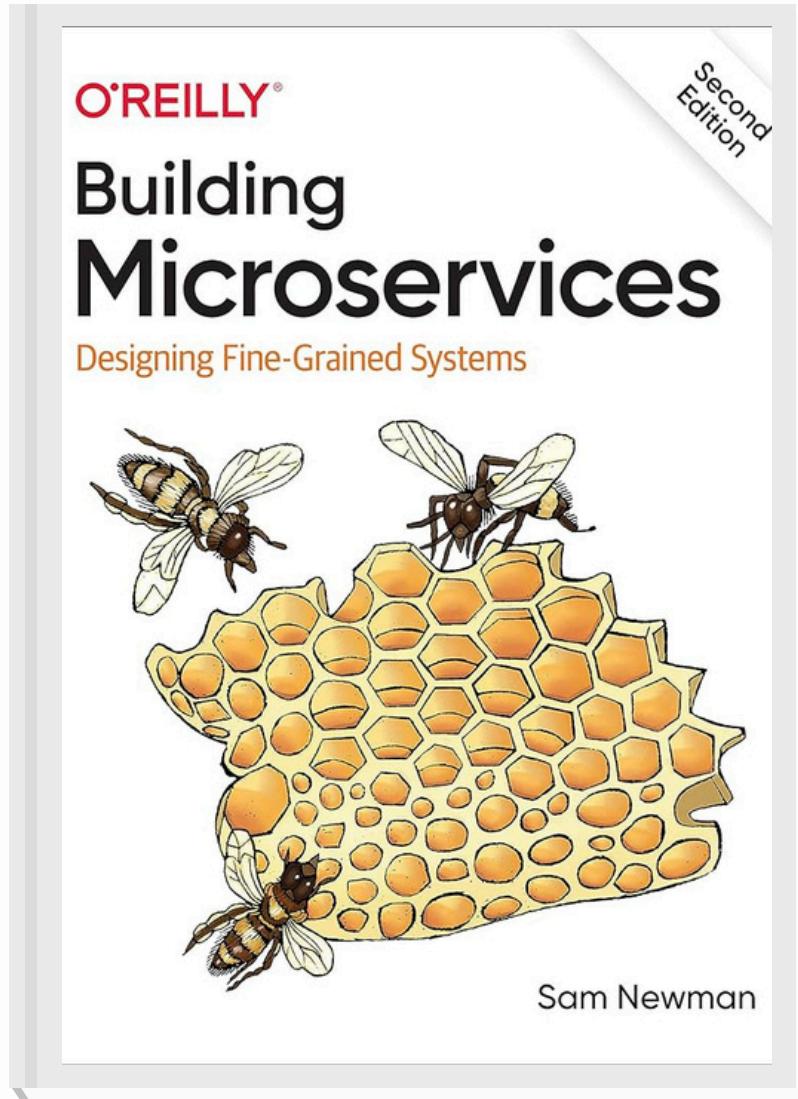
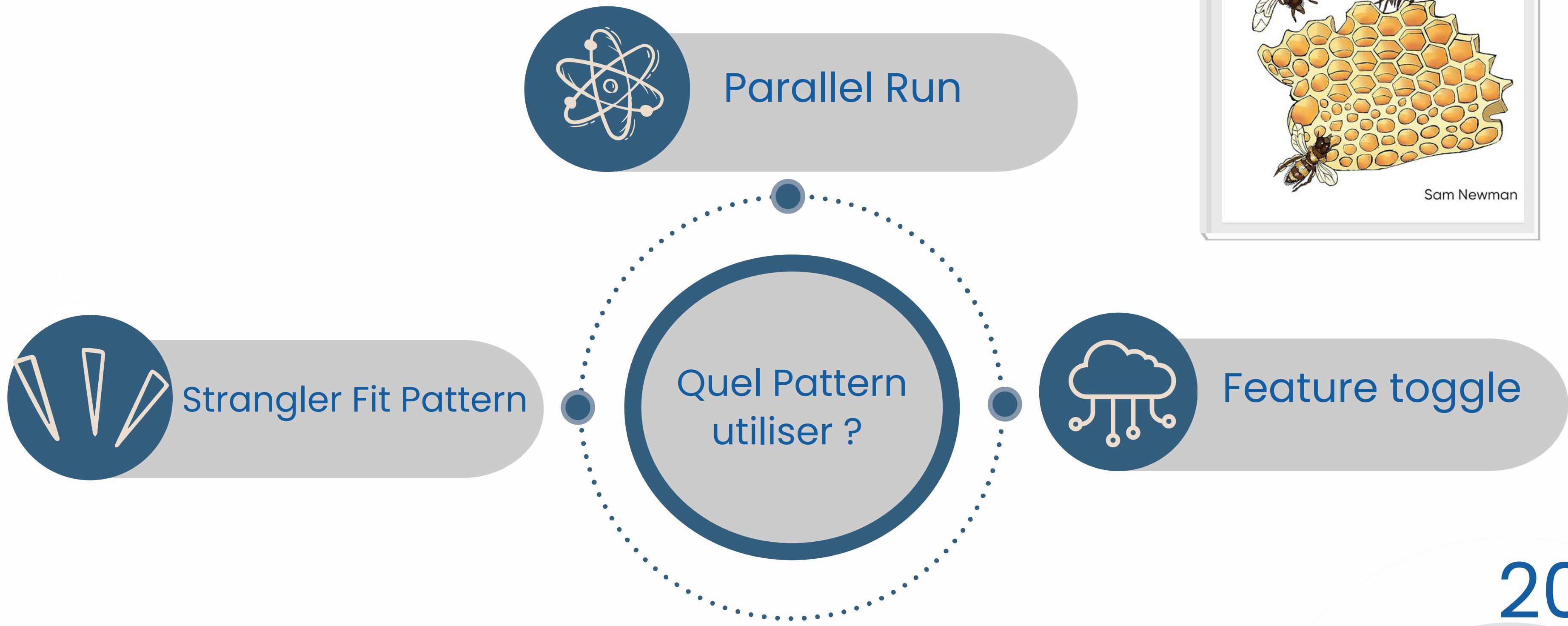
- Lister tous les produits disponibles
- Inscription et connexion des utilisateurs
- Ajouter des produits au panier et wishlist
- Passer des commandes et consulter les commandes



Architecture Monolithique

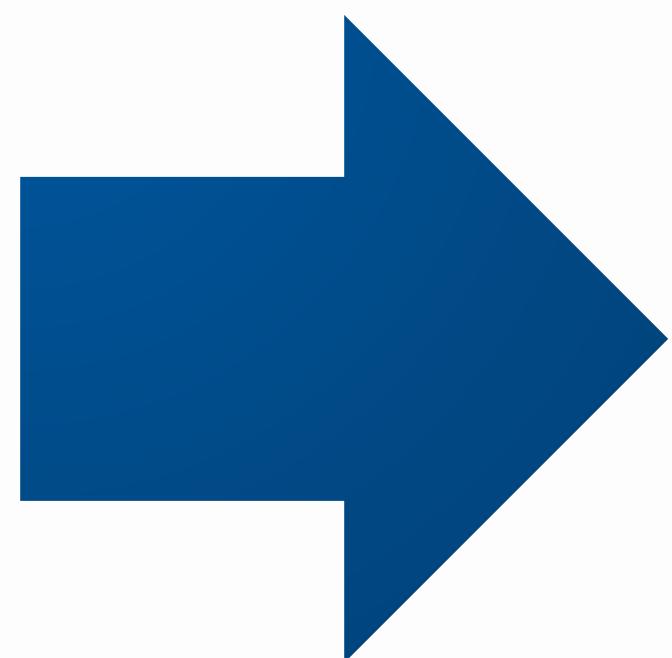


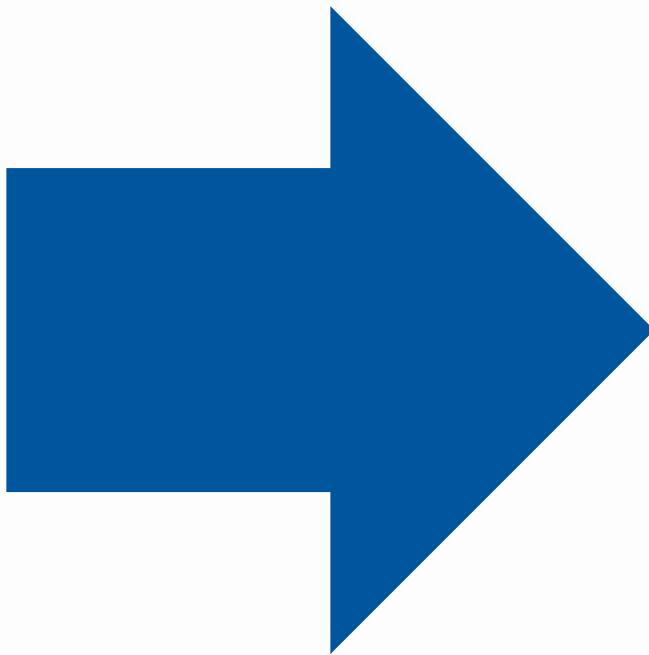
Migration



Monolith tree

```
.  
  api  
    customer.js  
    index.js  
    middlewares  
      auth.js  
    products.js  
    shopping.js  
  config  
    index.js  
  database  
    connection.js  
    index.js  
    models  
      Address.js  
      Customer.js  
      Order.js  
      Product.js  
      index.js  
    repository  
      customer-repository.js  
      product-repository.js  
      shopping-repository.js  
  express-app.js  
  index.js  
  sampledata.json  
  services  
    customer-service.js  
    product-service.js  
    shopping-service.js  
  utils  
    app-errors.js  
    error-handler.js  
    index.js
```





Shopping

```
.  
  └── api  
    ├── app-events.js  
    ├── index.js  
    └── middlewares  
      └── auth.js  
    └── shopping.js  
  └── config  
    └── index.js  
  └── database  
    ├── connection.js  
    └── index.js  
  └── models  
    ├── Cart.js  
    ├── Order.js  
    └── index.js  
  └── repository  
    └── shopping-repository.js  
  └── express-app.js  
  └── index.js  
  └── sampledata.json  
  └── services  
    ├── shopping-service.js  
    └── shopping-service.test.js  
  └── utils  
    ├── app-errors.js  
    └── error-handler.js  
  └── index.js
```

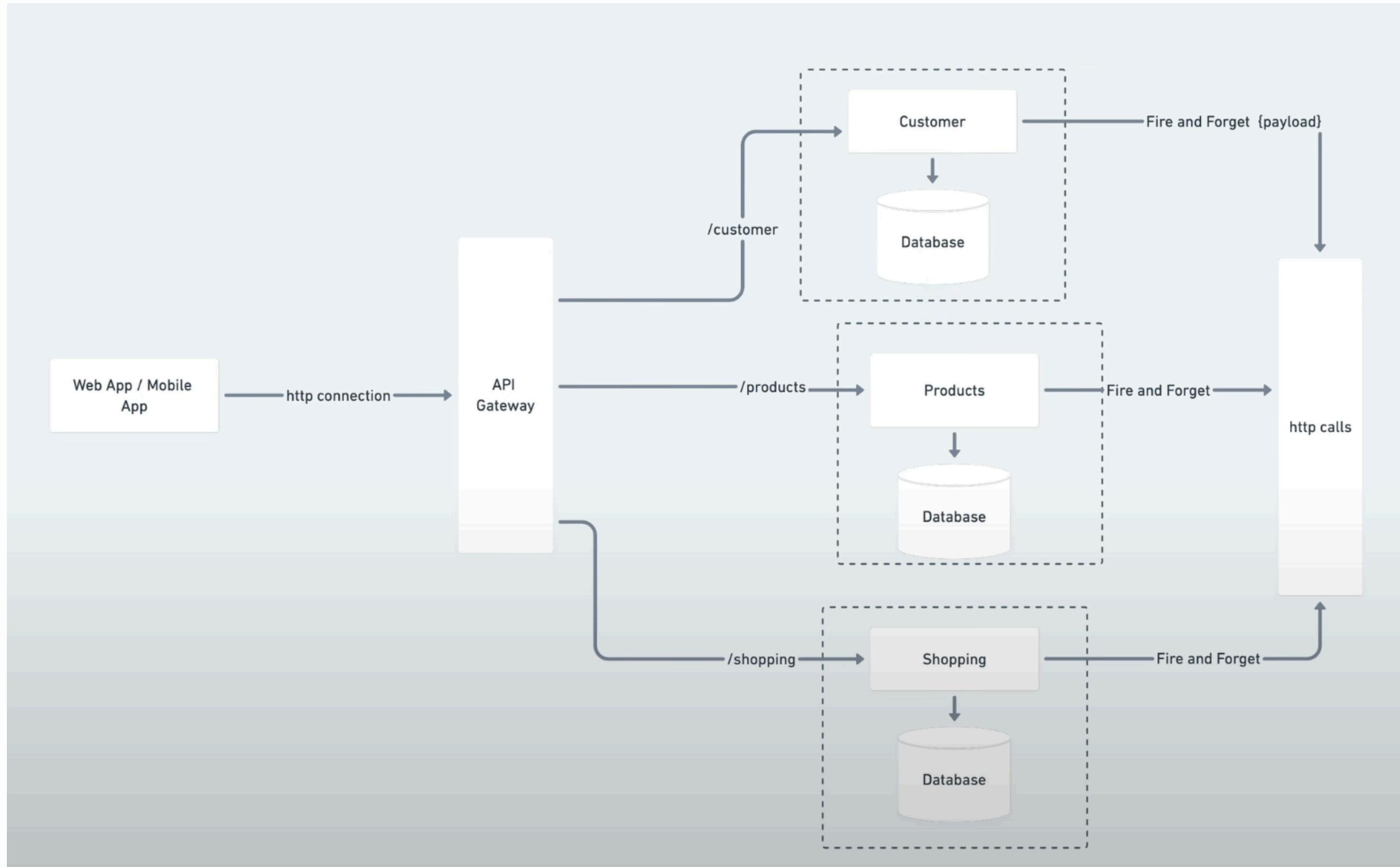
Customer

```
.  
  └── api  
    ├── app-events.js  
    ├── customer.js  
    ├── index.js  
    └── middlewares  
      └── auth.js  
  └── config  
    └── index.js  
  └── database  
    ├── connection.js  
    └── index.js  
  └── models  
    ├── Address.js  
    ├── Customer.js  
    └── index.js  
  └── repository  
    └── customer-repository.js  
  └── express-app.js  
  └── index.js  
  └── sampledata.json  
  └── services  
    ├── customer-service.js  
    └── customer-service.test.js  
  └── utils  
    ├── app-errors.js  
    └── error-handler.js  
  └── index.js
```

Product

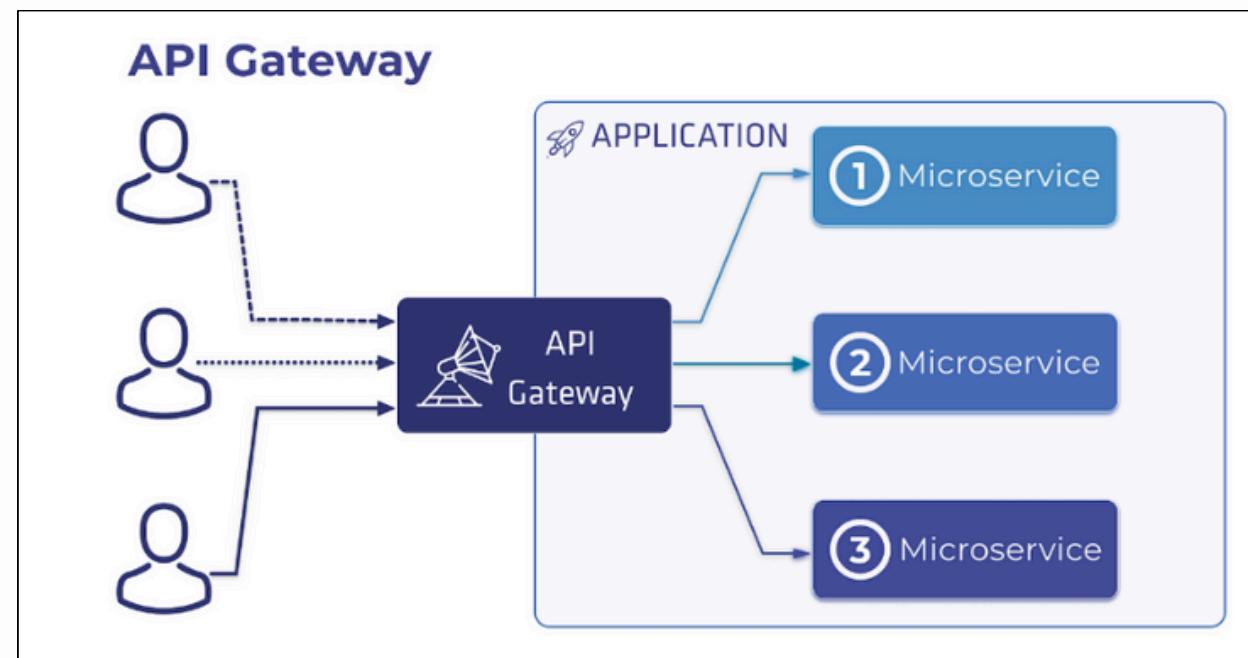
```
.  
  └── api  
    ├── app-events.js  
    ├── index.js  
    └── middlewares  
      └── auth.js  
    └── products.js  
  └── config  
    └── index.js  
  └── database  
    ├── connection.js  
    └── index.js  
  └── models  
    └── Product.js  
    └── index.js  
  └── repository  
    └── product-repository.js  
  └── express-app.js  
  └── index.js  
  └── sampledata.json  
  └── services  
    └── product-service.js  
    └── product-service.test.js  
  └── utils  
    ├── app-errors.js  
    └── error-handler.js  
  └── index.js
```

Architecture Microservices



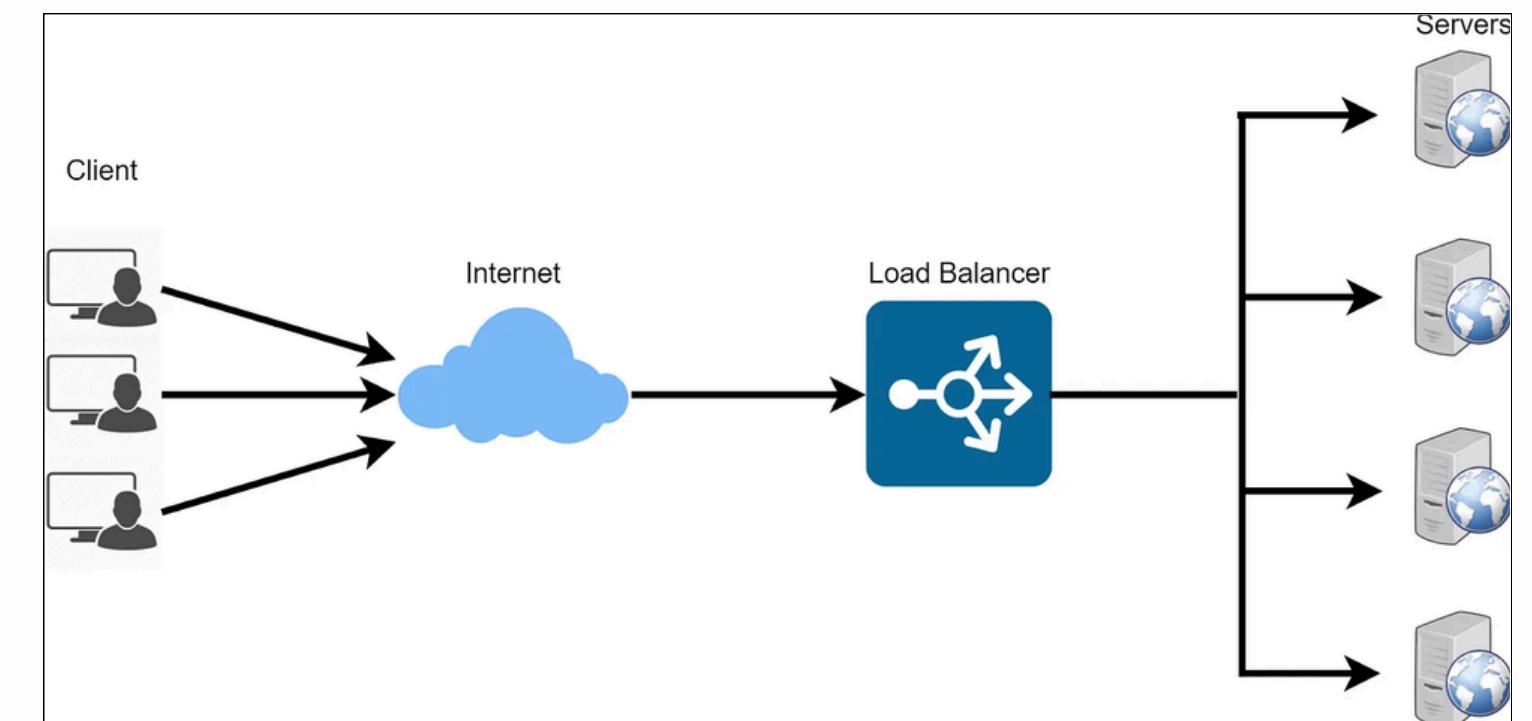
Réconfiguration

API GATEWAY



Redirige les requêtes

LOAD BALANCER



Répartit le trafic

Mise en place de la télémetrie

```
version: '3.8'
services:
  monolith-ter:
    container_name: monolith-ter1
    build:
      context: .
      dockerfile: Dockerfile
    environment:
      - OTEL_SERVICE_NAME=test-ter
      - OTEL_METRICS_EXPORTER=prometheus
      - OTEL_TRACES_EXPORTER=jaeger
      - OTEL_EXPORTER_JAEGER_ENDPOINT=http://jaeger:14250
      - MONGODB_URI=mongodb://mongo:27017/amazon_demo
    ports:
      - "8001:8001"

  jaeger:
    container_name: jaeger-ter1
    image: jaegertracing/all-in-one:latest
    ports:
      - "16686:16686"
      - "14250:14250"

  mongo:
    container_name: mongol
    image: mongo:latest
    ports:
      - "27017:27017"

  prometheus:
    image: prom/prometheus:latest
    ports:
      - "9090:9090"

  grafana:
    container_name: grafana-ter1
    image: grafana/grafana:latest
    ports:
      - "3000:3000"
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=admin
    depends_on:
      - prometheus
    volumes:
      - ./grafana:/etc/grafana/provisioning
```



```
# Téléchargez et copiez le JAR de l'exportateur Jaeger dans le répertoire de votre application
ADD https://repo1.maven.org/maven2/io/opentelemetry/opentelemetry-exporter-jaeger/1.34.1/opentelemetry-expo

# Téléchargez et copiez l'agent OpenTelemetry Java dans le répertoire de votre application
ADD https://repo1.maven.org/maven2/io/opentelemetry/javaagent/opentelemetry-javaagent/1.29.0/opentelemetry-j

# Spécifiez l'utilisation de l'agent OpenTelemetry Java avec l'exportateur Jaeger
ENV JAVA_TOOL_OPTIONS "-javaagent:./opentelemetry-javaagent-1.29.0.jar"
```

Tests avec Postman



HTTP Microservices / Customer / PRODUCTS_CREATE

POST http://localhost:8001/product/create

Save ▾

Send ▾

Params Authorization Headers (9) **Body** • Scripts Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON ▾

Beautify

```
1 {  
2   .... "name": "Apples",  
3   .... "desc": "great Quality of Apple",  
4   .... "type": "fruits",  
5   .... "banner": "http://codergogoi.com/youtube/images/apples.jpeg",  
6   .... "unit": 1,  
7   .... "price": 140,  
8   .... "available": true,  
9   .... "supplier": "Golden seed firming"  
10 }
```

Tests avec K6



```
● ● ●

import http from 'k6/http';
import { sleep } from 'k6';

export default function () {
    // Simulate POST request to /signup endpoint
    http.post('http://localhost:8001/signup', JSON.stringify({
        email: 'test@example.com',
        password: 'password123',
        phone: '1234567890'
    }), {
        headers: {
            'Content-Type': 'application/json',
        },
    });

    // Simulate POST request to /login endpoint
    http.post('http://localhost:8001/login', JSON.stringify({
        email: 'test@example.com',
        password: 'password123'
    }), {
        headers: {
            'Content-Type': 'application/json',
        },
    });
}
```

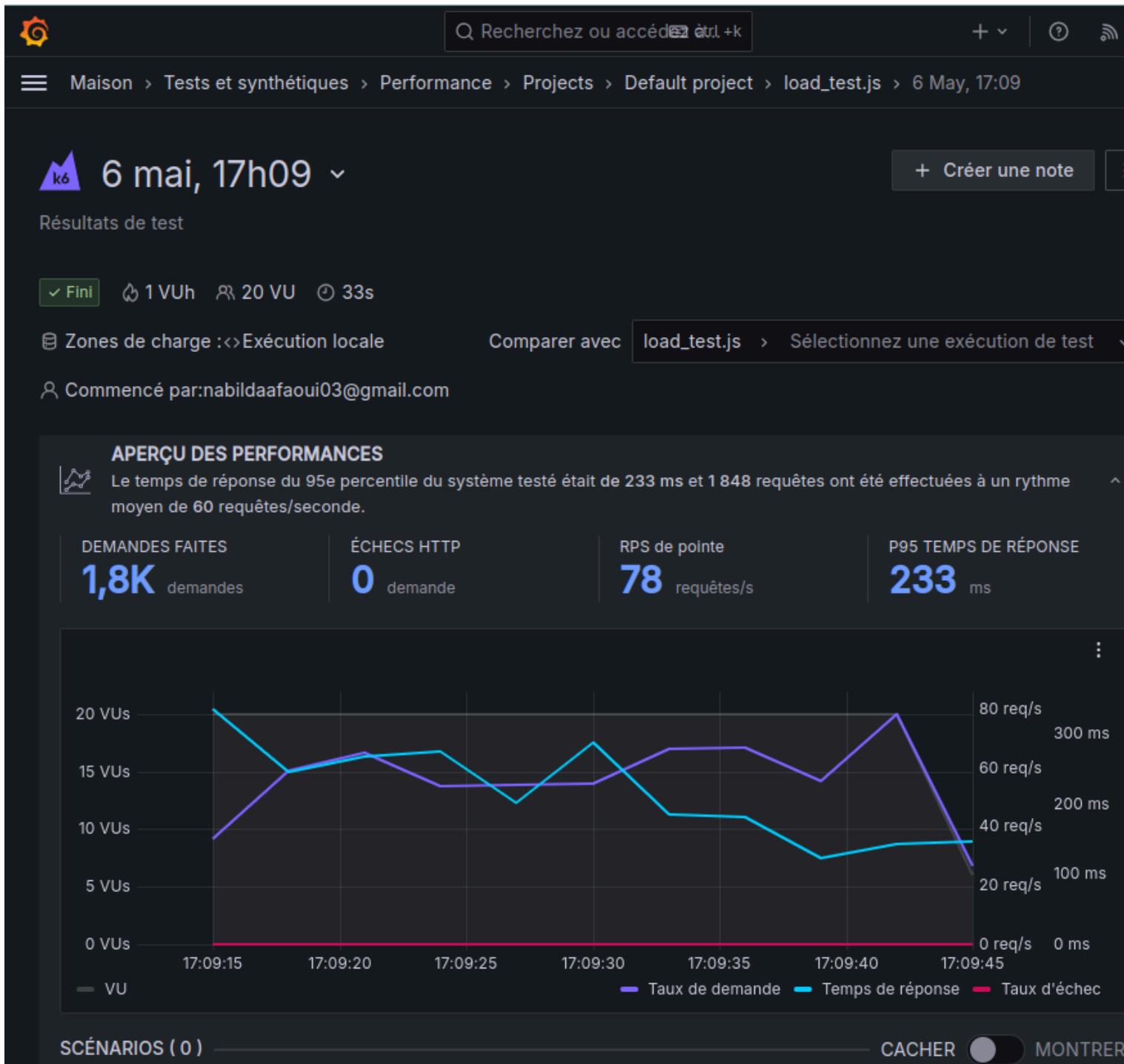
Pour lancer les tests



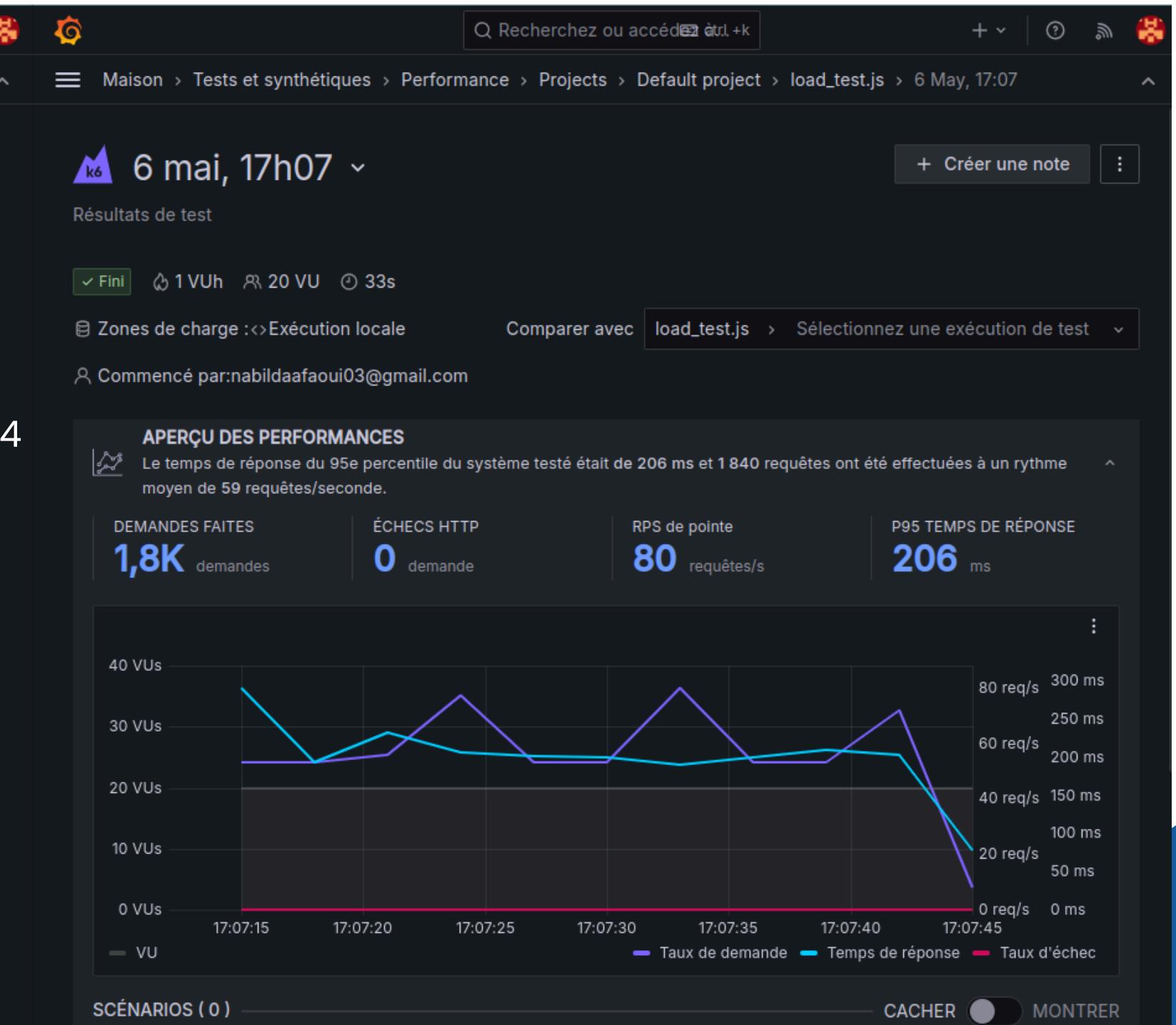
```
k6 run --vus 10 --duration 30s loadtest.js
```

Test : 20 VUs

Monolith



Microservices

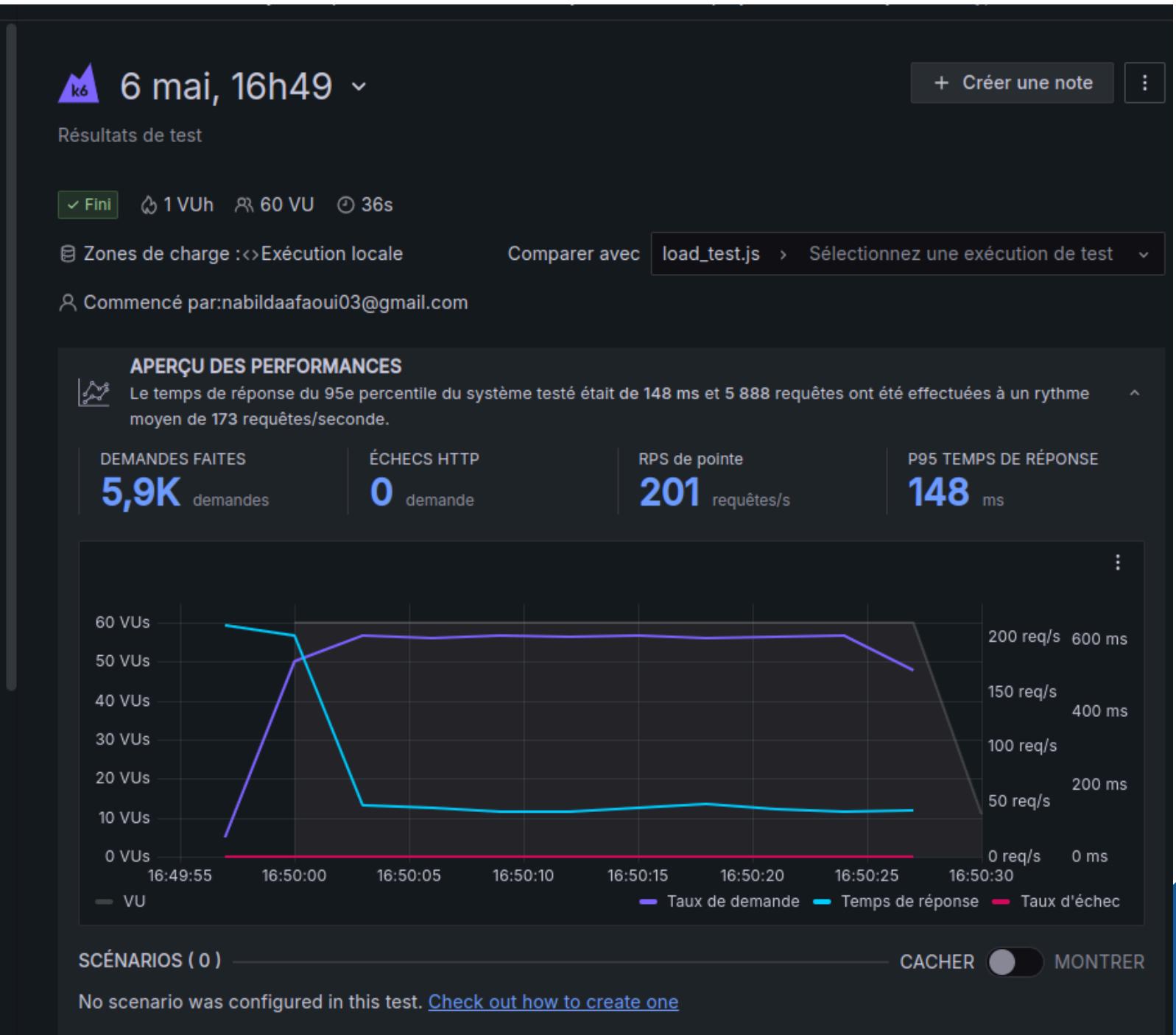


Test : 60 VUs

Monolith

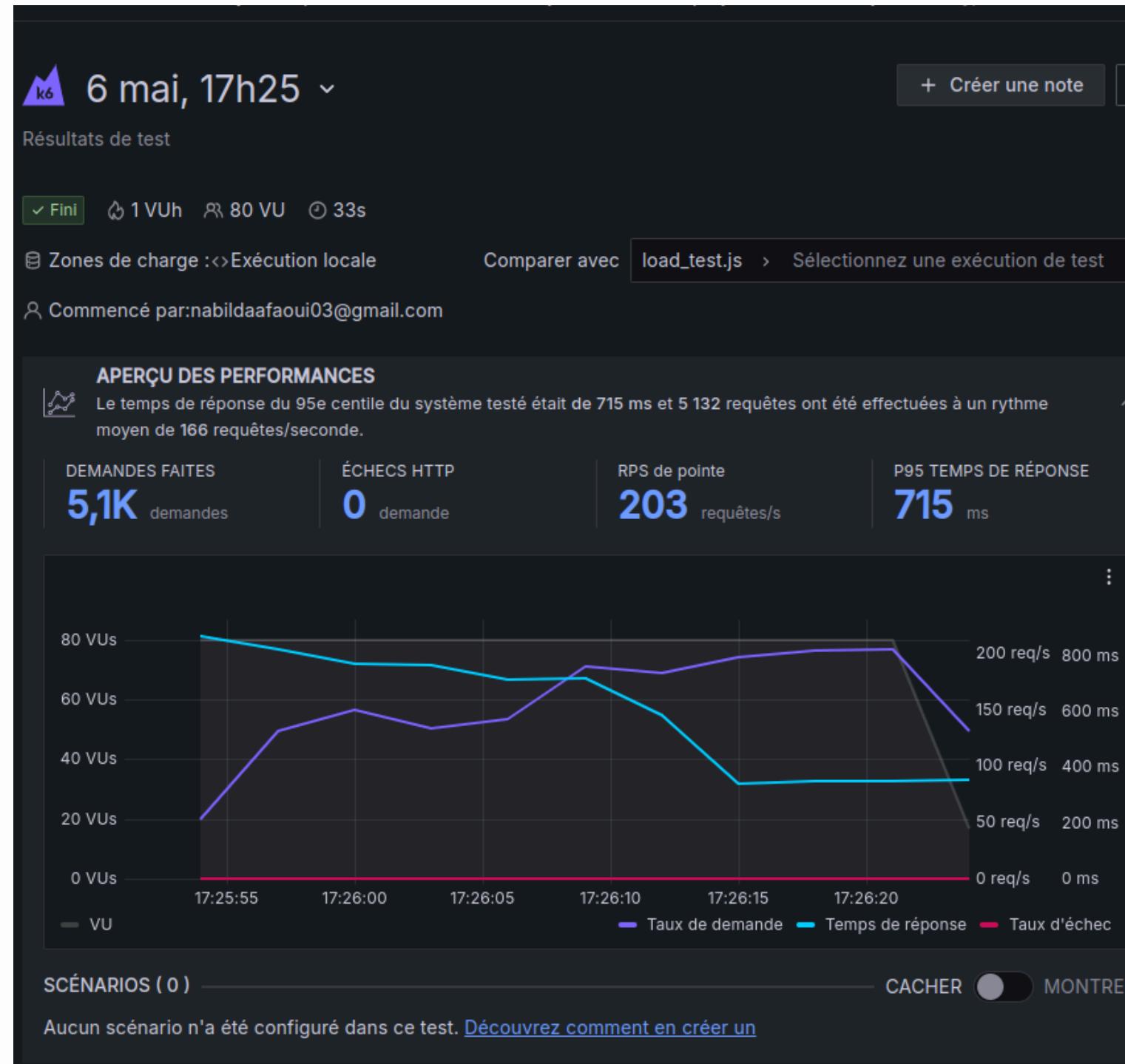


Microservices

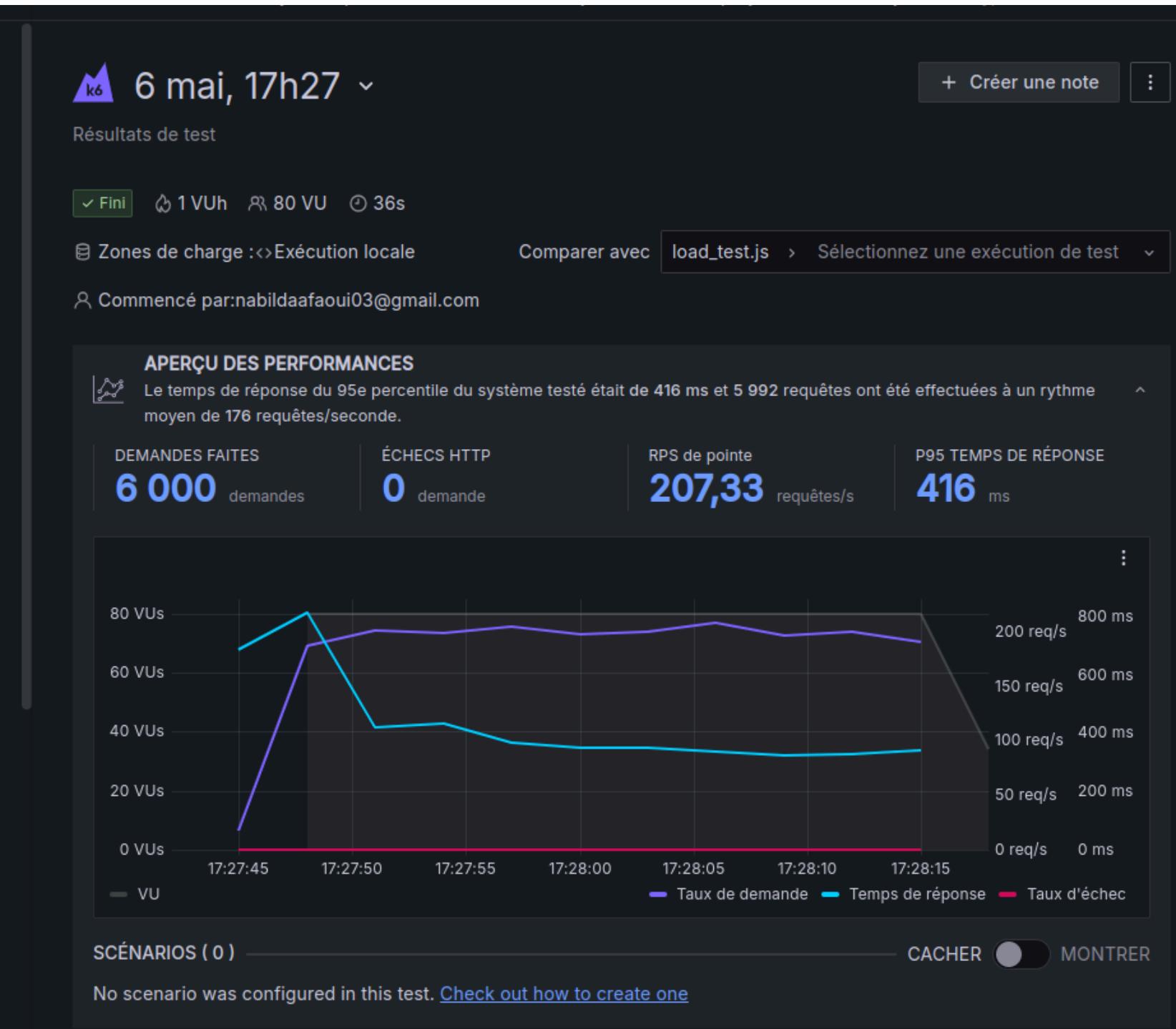


Test : 80 VUs

Monolith

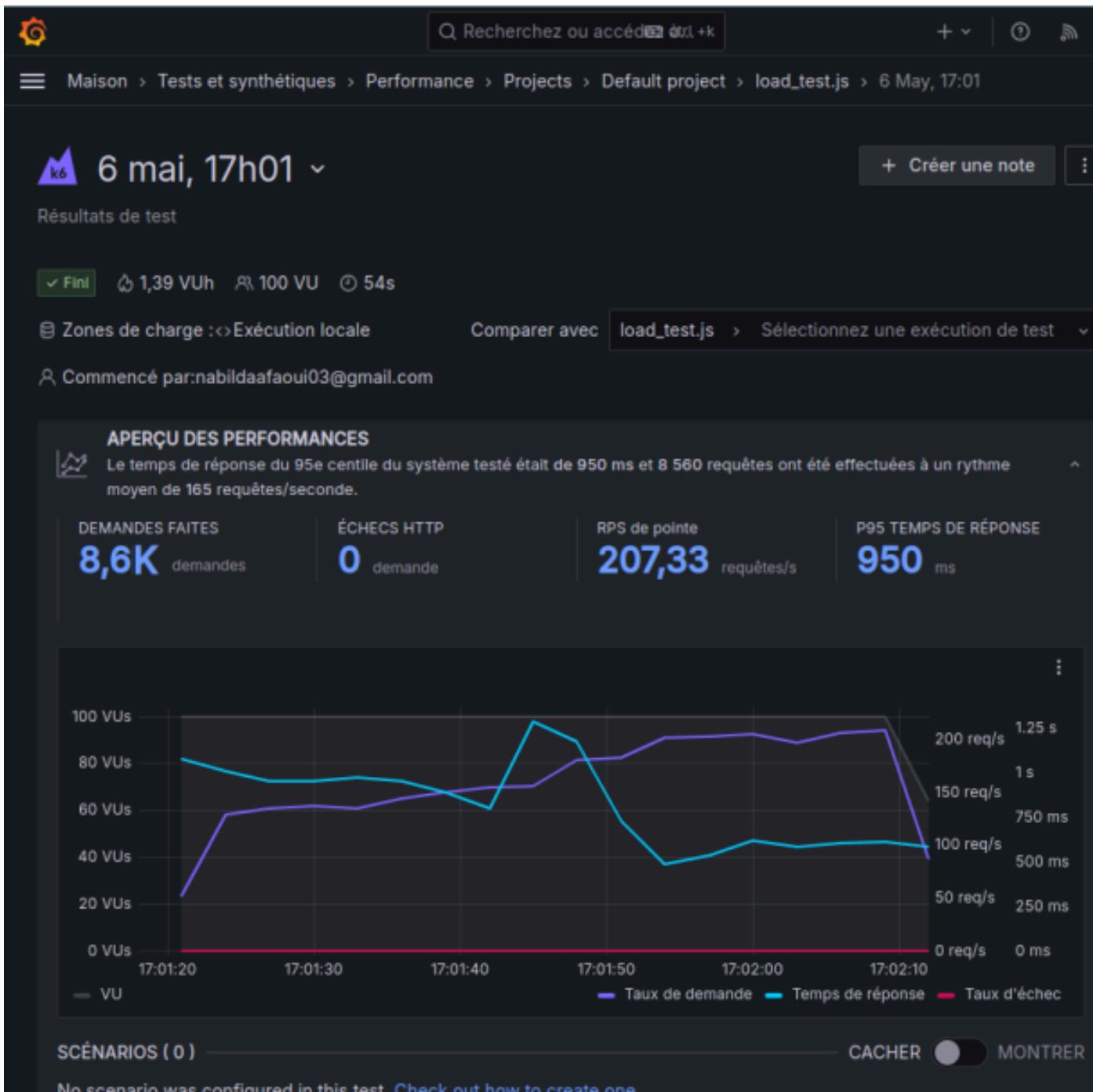


Microservices

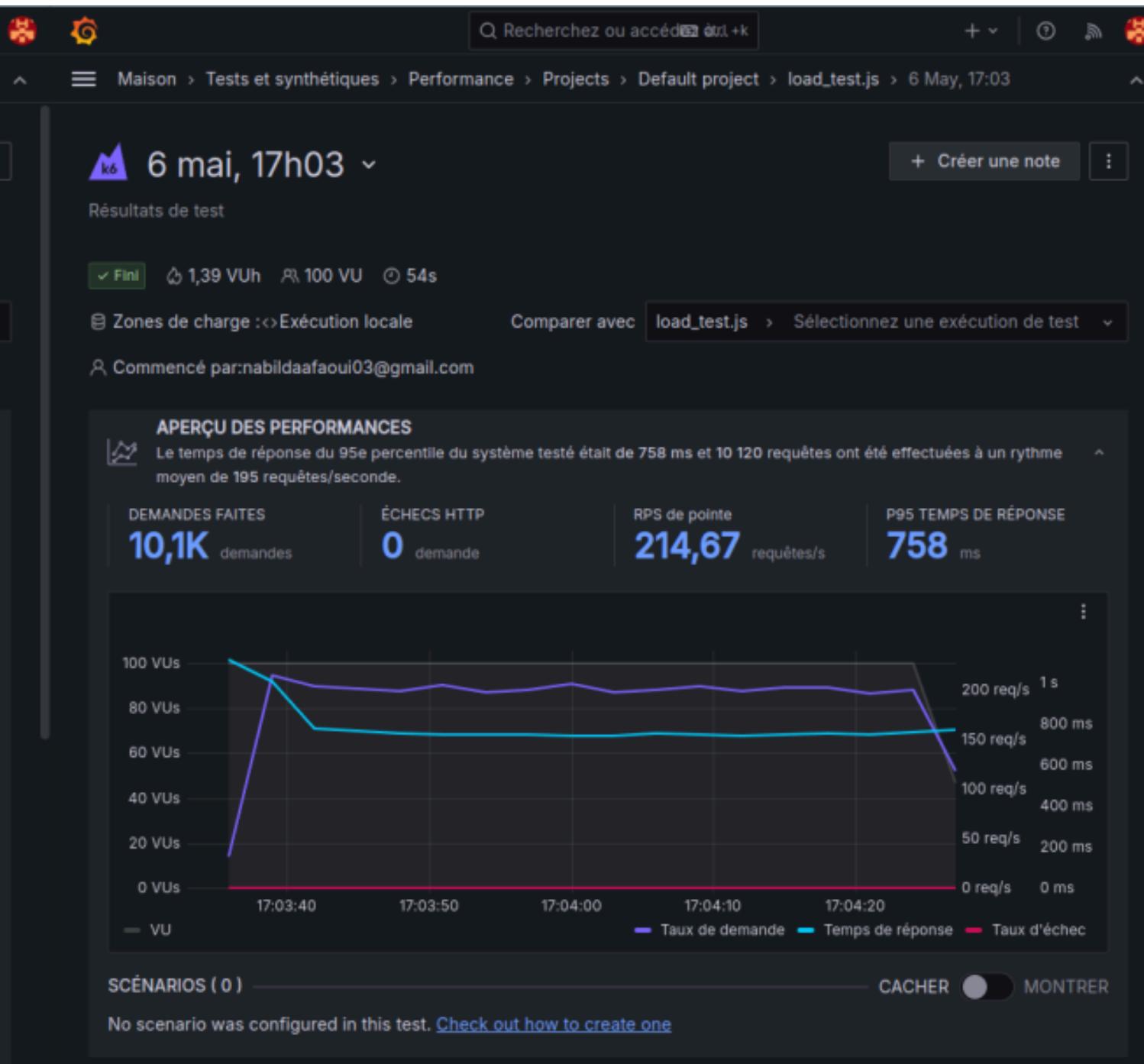


Test : 100 VUs

Monolith

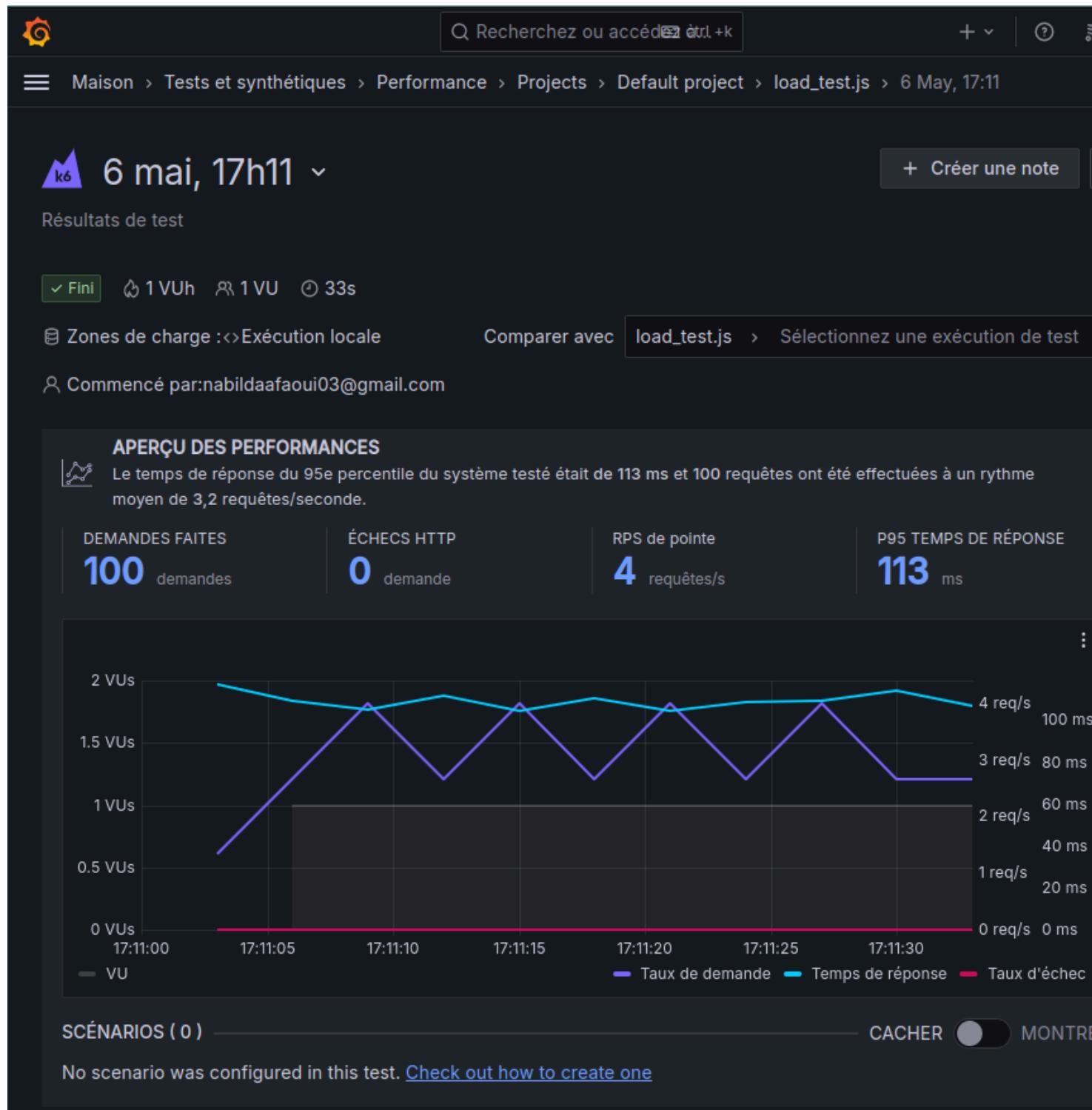


Microservices

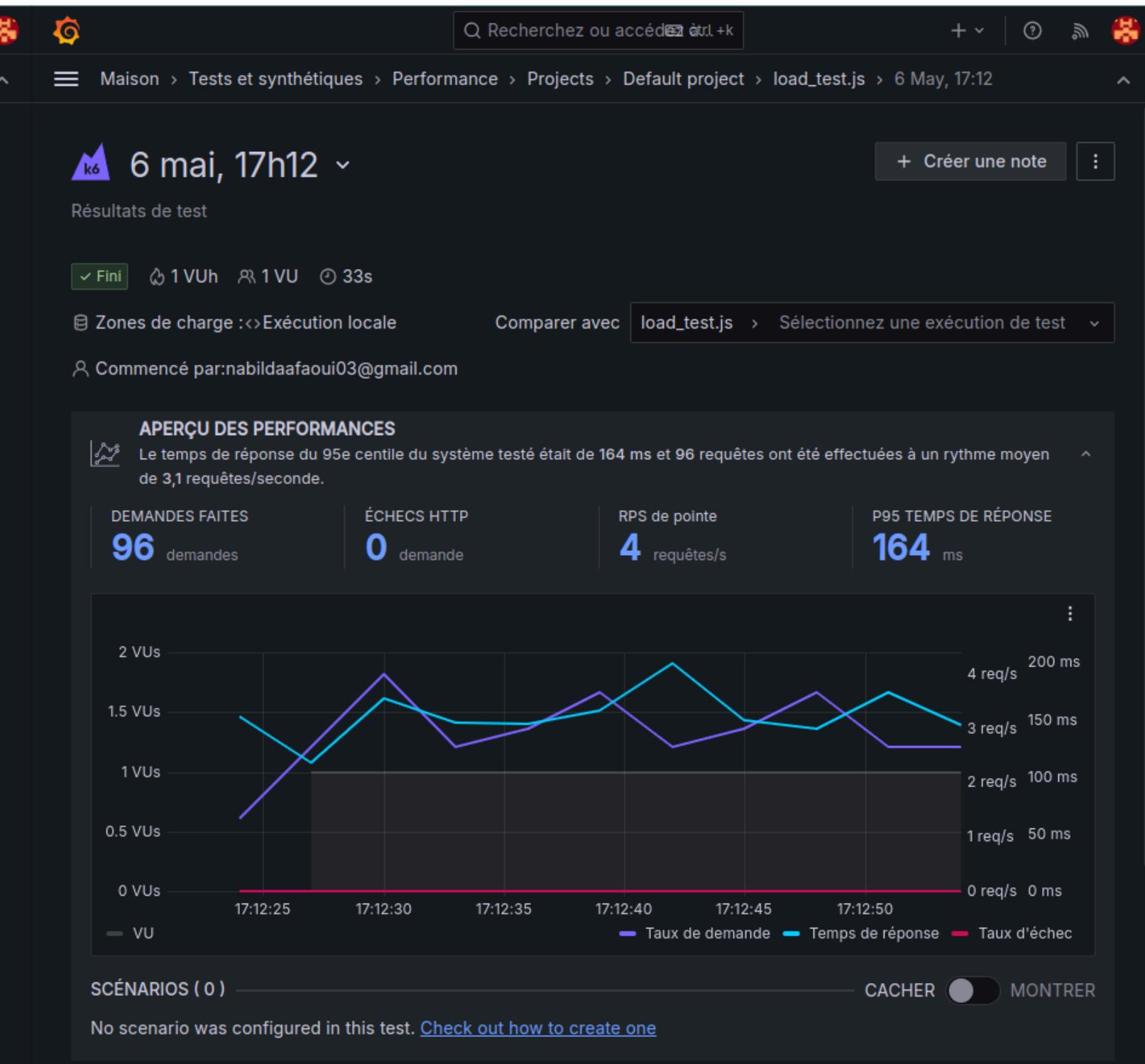


Test : 1 VU

Monolith



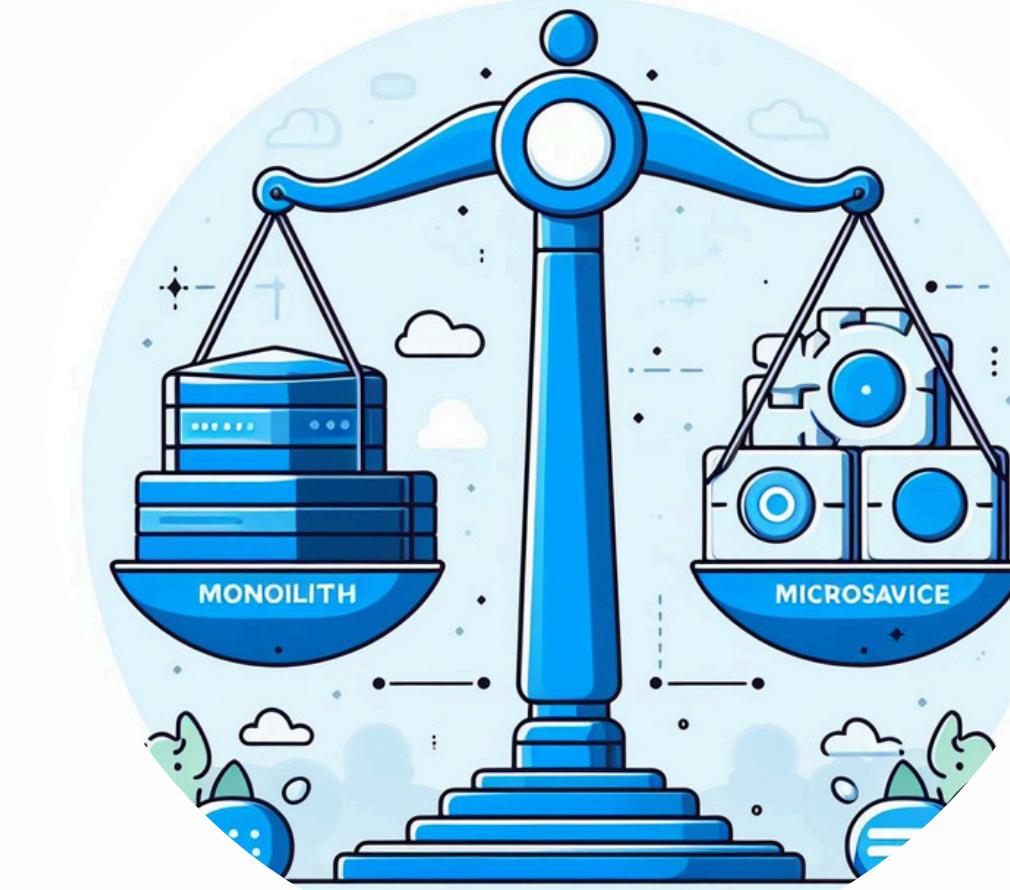
Microservices



Conclusion

AUCUNE APPROCHE PARFAITE, UN COMPROMIS À ÉVALUER

- Le choix dépend des priorités du projet
- Évaluation minutieuse des exigences cruciales



- | | |
|--|---|
| <ul style="list-style-type: none">• Simplicité initiale• Transactions facilitées• Évolutivité limitée• Manque de modularité | <ul style="list-style-type: none">• Flexibilité• Déploiement indépendant• Complexité de gestion• Cohérence des données |
|--|---|



UNIVERSITÉ DE
MONTPELLIER

THANK YOU

FOR YOUR ATTENTION

Mai 2024

