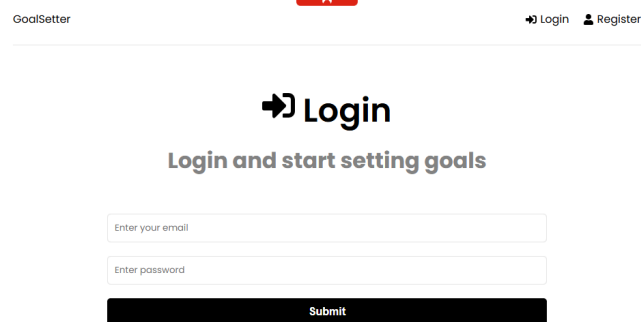


1. Choix et préparation de l'application

Pour cet exercice, nous avons choisi une application open source disponible sur GitHub, développée par Brad Traversy, accessible à l'adresse : <https://github.com/bradtraversy/mern-tutorial>. Cette application est un bon candidat pour notre projet car elle correspond parfaitement aux critères demandés : elle possède un frontend, un backend et une base de données, sans fichiers Docker ni manifestes Kubernetes existants.

Présentation générale de l'application

L'application que nous avons choisie se nomme **GoalSetter**. Elle consiste à noter des “choses à faire”. Elle est une forme de todo-list et peut également servir de coffre-fort (conserver des mots de passe, conserver des tokens). Pour ce faire, il suffit de se connecter (ou de s'inscrire) avec une adresse mail.



The screenshot shows the login page of the GoalSetter application. At the top left is the text "GoalSetter" and at the top right are links for "Login" and "Register". The main heading is "Login" with a key icon. Below it is the subheading "Login and start setting goals". There are two input fields: "Enter your email" and "Enter password". At the bottom is a black "Submit" button.

GoalSetter

Login Register

Login

Login and start setting goals

Enter your email

Enter password

Submit

Register

Please create an account

Welcome cherif

Goals Dashboard

Goal

You have not set any goals

Welcome cherif

Goals Dashboard

Goal

<div>7/13/2025, 9:40:45 AM</div> <div>SOCADY</div> <div>X</div>	<div>7/13/2025, 9:41:01 AM</div> <div>faire à manger pour midi</div> <div>X</div>
<div>7/13/2025, 9:41:20 AM</div> <div>Aller à la pharmacie à 14h</div> <div>X</div>	<div>7/13/2025, 9:41:35 AM</div> <div>Finir le projet FinOps</div> <div>X</div>
<div>7/13/2025, 9:41:59 AM</div> <div>Faire un brainstorming à 20h avec l'équipe</div> <div>X</div>	

L'application est une application web full-stack basée sur la stack MERN, un acronyme pour MongoDB, Express.js, React et Node.js. Cette architecture est très populaire pour développer des applications web modernes et performantes. Le frontend est développé en React et backend est un serveur Node.js utilisant le framework Express.js, qui fournit une API REST pour gérer les requêtes du frontend, la logique métier, et l'interaction avec la base de données.

MongoDB est utilisé comme base de données NoSQL, adaptée à la gestion de données sous forme de documents JSON. Cette base permet une grande flexibilité pour stocker et manipuler les données liées aux utilisateurs, aux sessions, et à d'autres objets métiers de l'application.

Structure technique

Dans le dépôt GitHub, on trouve clairement séparément les dossiers pour chaque composant de l'application. Le dossier frontend/ contient tout le code source React, tandis que le backend est à la racine du projet avec le code serveur Node.js. MongoDB est utilisé via une image Docker officielle (dans notre futur docker-compose) mais n'a pas de code spécifique dans le dépôt : il s'agit d'une base de données externe que l'application utilise.

Composant	Technologie
Frontend	React + Redux
Backend	Node.js + Express
Base de données	MongoDB
API Authentification	JWT (JSON Web Token)
Déploiement	Docker + Docker Compose

Pourquoi ce choix ?

Ce projet remplit les critères clés du travail :

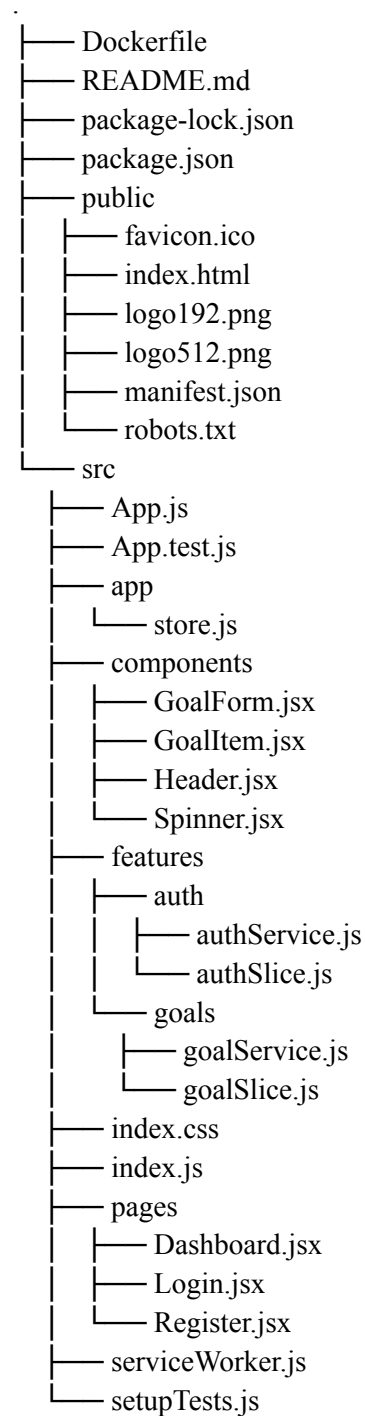
- React est très utilisé en production, ce qui rend l'apprentissage applicable dans de nombreux contextes.
- Base de données flexible : MongoDB s'adapte bien aux applications modernes et facilite la gestion des données.
- Pas de dockerisation ni orchestration Kubernetes : l'absence de Dockerfile et Kubernetes permet de pratiquer la dockerisation et le déploiement Kubernetes de A à Z, ce qui correspond

exactement à l'objectif du devoir.

2. Dockerisation des composants

Frontend

L'architecture de notre frontend se présente comme suit:



Nous avons créé un fichier Dockerfile à l'intérieur de ce dossier et dans ce fichier, nous avons les informations suivantes:

```
FROM node:18-alpine
```

```
WORKDIR /app
```

```
# Copy package files  
COPY package*.json ./
```

```
# Install dependencies  
RUN npm install
```

```
# Copy source code  
COPY . .
```

```
# Expose port  
EXPOSE 3000
```

```
# Set environment variables for React in container  
ENV CHOKIDAR_USEPOLLING=true  
ENV WATCHPACK_POLLING=true
```

```
# Start the application  
CMD ["npm", "start"]
```

Ce fichier dockerfile nous permet de créer un conteneur propre au Frontend qui sera par la suite utilisé dans le docker-compose.yml de notre application

Backend

L'architecture de notre backend se présente comme suit:

```
.  
├── Dockerfile  
├── config  
│   └── db.js  
├── controllers  
│   ├── goalController.js  
│   └── userController.js  
├── middleware  
│   ├── authMiddleware.js  
│   └── errorMiddleware.js  
├── models  
│   ├── goalModel.js  
│   └── userModel.js  
├── routes  
└── goalRoutes.js
```

```
| └─ userRoutes.js
|   └─ server.js
```

Nous avons créé un Dockerfile à l'intérieur du serveur backend dont les informations sont les suivantes:

```
FROM node:18-alpine
```

```
WORKDIR /app
```

```
# Copy package files from root
COPY package*.json ./
```

```
# Install ALL dependencies (frontend + backend)
RUN npm install
```

```
# Copy entire project
COPY . .
```

```
# Change to backend directory for execution
WORKDIR /app
```

```
# Expose port
EXPOSE 5000
```

```
# Start the backend server
CMD ["npm", "run", "server"]
```

Création d'un docker-compose.yml

Nous n'avons pas créé d'image Docker pour Mongo car une image officielle existe déjà. On a juste spécifié Mongo dans notre fichier docker-compose.yml

Notre code renseigné est:

```
version: "3.8"
```

```
services:
```

```
  # MongoDB Database
```

```
  mongodb:
```

```
    image: mongo:6.0
```

```
    container_name: mern-mongodb
```

```
    restart: unless-stopped
```

```
    environment:
```

```
      MONGO_INITDB_DATABASE: mernapp
```

```
    ports:
```

```
      - "27017:27017"
```

```
    volumes:
```

```
      - mongodb_data:/data/db
```

```

networks:
  - mern-network

# Backend Service
backend:
  build:
    context: . # Contexte à la racine car package.json y est
    dockerfile: backend/Dockerfile
  container_name: mern-backend
  restart: unless-stopped
  environment:
    NODE_ENV: development
    PORT: 5000
    MONGO_URI: mongodb://mongodb:27017/mernapp
    JWT_SECRET: abc123
  ports:
    - "5000:5000"
  depends_on:
    - mongodb
  networks:
    - mern-network

# Frontend Service
frontend:
  build:
    context: ./frontend
    dockerfile: Dockerfile
  container_name: mern-frontend
  restart: unless-stopped
  environment:
    REACT_APP_API_URL: http://localhost:5000
    CHOKIDAR_USEPOLLING: true
  ports:
    - "3000:3000"
  depends_on:
    - backend
  networks:
    - mern-network

volumes:
  mongodb_data:

networks:
  mern-network:
    driver: bridge

```

Une fois ces fichiers créés, nous avons effectué un *docker-compose --build* afin de créer des images docker.

L'image suivante montre que nos conteneurs ont bien été créés:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
03c15c7e226e	mern-tutorial-frontend	"docker-entrypoint.s..."	6 minutes ago	Up 6 minutes	0.0.0.0:3000->3000/tcp	mern-frontend
fd2179988f6f	mern-tutorial-backend	"docker-entrypoint.s..."	6 minutes ago	Up 6 minutes	0.0.0.0:5000->5000/tcp	mern-backend
b036f47698ed	mongo:6.0	"docker-entrypoint.s..."	22 hours ago	Up 6 minutes	0.0.0.0:27017->27017/tcp	mern-mongodb

Le frontend de l'application est à retrouver ici: <http://localhost:3000>

A présent, nous allons passer à la construction de manifests Kubernetes:

3. Construction de manifests Kubernetes

Pour rendre nos conteneurs plus scalables et surveiller leur état, nous allons passer à Kubernetes. Pour ce faire, nous allons créer à la racine un dossier k8s/ qui aura plusieurs dossiers : mongodb, backend, frontend, config/

L'arborescence de notre dossier k8s se présente comme suit:

```

├── backend
│   ├── deployment.yaml
│   ├── secret.yaml
│   └── service.yaml
├── frontend
│   ├── deployment.yaml
│   └── service.yaml
└── mongodb
    ├── deployment.yaml
    ├── pvc.yaml
    └── service.yaml

```

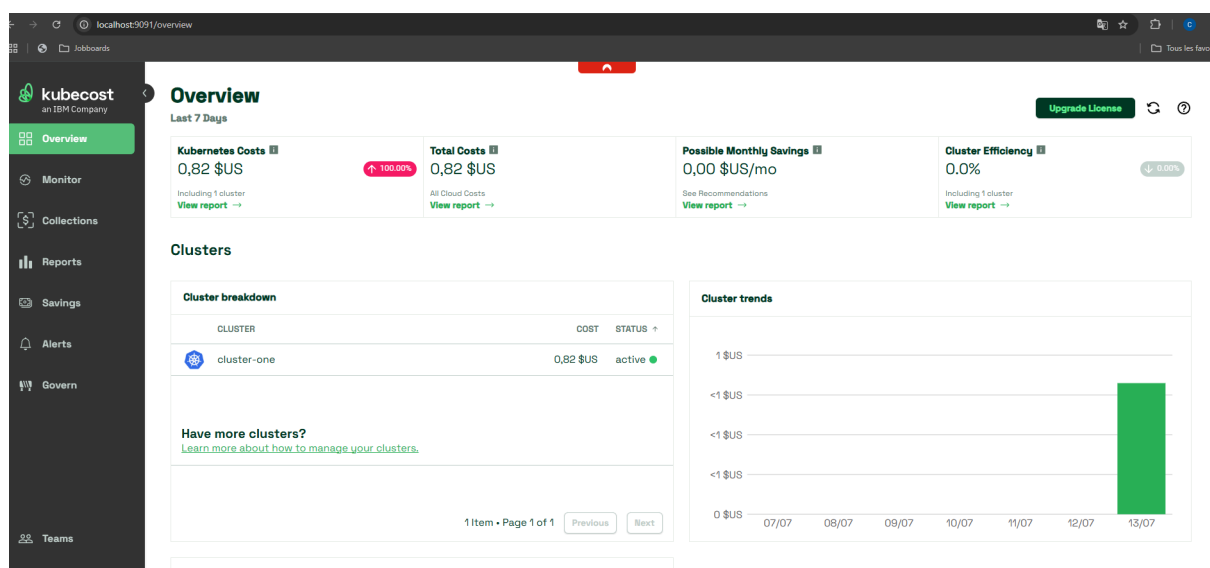
Ensuite, on applique la commande `kubectl apply -f frontend`

4. Déploiement sur Minikube

5. Intégration Kubecost

Après la création d'un compte sur notre application, des ressources kubernetes(pods et services) ont été provisionnées. Kubecost a détecté cette activité dans le cluster et a évalué les coûts associés. Pour le moment, les coûts totaux sur les 07 derniers jours s'élèvent à **0,82\$US**.

Il n'y a actuellement **aucune optimisation détectée** (économies possibles = 0,00 \$US/mois), et **l'efficacité du cluster est de 0 %**, ce qui indique probablement une **utilisation très faible par rapport aux ressources allouées**.

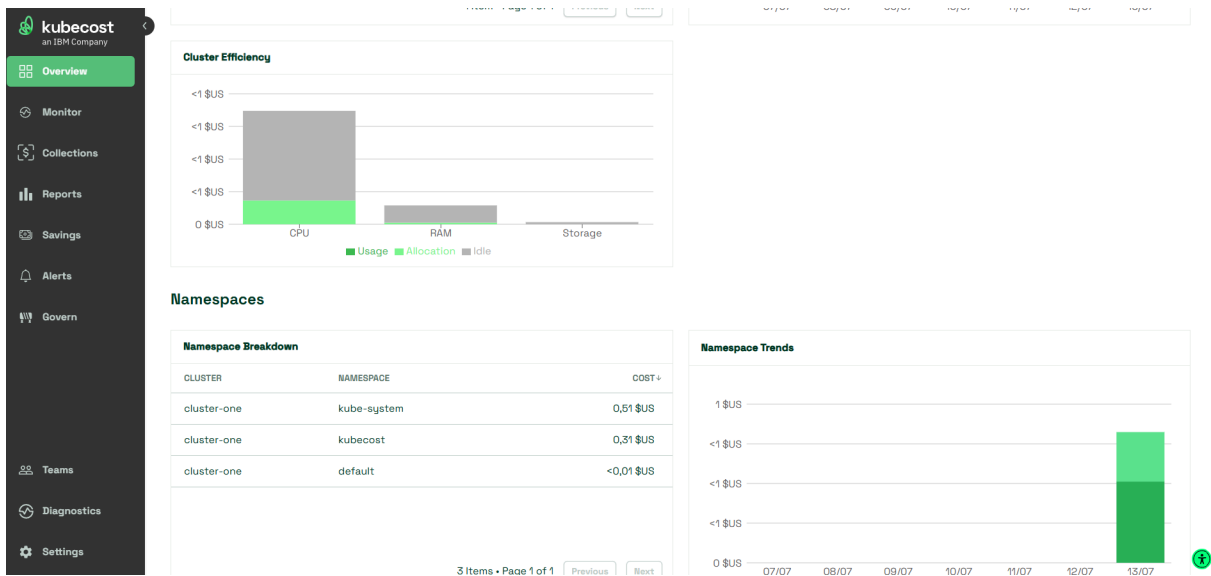


Dans l'image ci-dessous, on peut voir globalement les ressources allouées et utilisées, la répartition des coûts par Namespace qui est une séparation logique au sein d'un cluster Kubernetes.

On constate que :

- Sur la barre CPU, l'Idle(gris) est énorme par rapport à l'usage réelle(vert clair) par conséquent il y'a **surprovisionnement de CPU**
- Sur la RAM, il y'a une petite différence mais ne nécessite pas d'ajustement
- le stockage n'a pas de problème de surprovisionnement

Le namespace le plus utilisé est Kube-system.



Le dashboard ci-dessous correspond **aux coûts d'infrastructure Kubernetes**, ventilés par type de ressource (Node, Disk, Network). On peut voir que le Node est la ressource la plus consommatrice




Allocations

[Upgrade License](#) [Refresh](#) [Help](#)

Cumulative cost for 28 June 2025 through 13 July 2025 by namespace

[Custom](#) [Namespace](#) [Filter](#) [Edit](#) [Save](#) [More](#)

NAME	CPU	GPU	RAM	PV	NETWORK	LB	SHARED	EFFICIENCY	TOTAL COST	
Total	1,60 \$US	0,00 \$US	0,27 \$US	0,03 \$US	0,00 \$US	0,00 \$US	0,00 \$US	0%	1,89 \$US	
__idle__	1,32 \$US	0,00 \$US	0,24 \$US	0,00 \$US	0,00 \$US	0,00 \$US	0,00 \$US	—	1,57 \$US	N/A
kube-system	0,18 \$US	0,00 \$US	<0,01 \$US	0,00 \$US	0,00 \$US	0,00 \$US	0,00 \$US	0.0%	0,19 \$US	0,0% ...
kubecost	0,09 \$US	0,00 \$US	0,01 \$US	0,03 \$US	0,00 \$US	0,00 \$US	0,00 \$US	0.0%	0,13 \$US	0,0% ...
monitoring	0,00 \$US	0,00 \$US	<0,01 \$US	0,00 \$US	0,00 \$US	0,00 \$US	0,00 \$US	0.0%	<0,01 \$US	0,0% ...
default	0,00 \$US	0,00 \$US	0,00 \$US	<0,01 \$US	0,00 \$US	0,00 \$US	0,00 \$US	0.0%	<0,01 \$US	0,0% ...

**kubecost**
an IBM Company

Overview

Monitor

Allocations

Assets

Cloud Costs

Clusters

External Costs

Efficiency

Network

Costs

Teams

Diagnostics

Settings

Cluster Details

cluster-one

Provider	custom
Version	N/A
Region	N/A
Health Score	N/A

Nodes	1
Namespaces	3
Pods	18
Controllers	9

Total Cost	0,82 \$US
Estimated Monthly Savings	
Workload Efficiency	N/A
Spending Trend	N/A

No Budget Set

[Add a New Budget](#)


Assets Last 7 days		
NAME	CREDITS	COST
Node	0,00 \$US	0,81 \$US
Disk	0,00 \$US	0,01 \$US
ClusterManagement	0,00 \$US	0,00 \$US

Efficiency

NAME	REQUESTED	USAGE
CPU	0% Efficiency	
RAM	0% Efficiency	
GPU	N/A Efficiency	

Asset Trends

Date	Asset Trends (\$US)
13/07/2025	0.82



kubecost

an IBM Company

Overview

Monitor

Allocations

Assets

Cloud Costs

Clusters

External Costs

Efficiency

Network

Follow

Teams

Diagnostics

Settings

Assets Last 7 days


NAME	CREDITS	COST
Node	0,00 \$US	0,81 \$US
Disk	0,00 \$US	0,01 \$US
ClusterManagement	0,00 \$US	0,00 \$US
LoadBalancer	0,00 \$US	0,00 \$US
Network	0,00 \$US	0,00 \$US

View in Assets

Namespaces Last 7 days

NAME	# OF PODS	COST	EFFICIENCY
kube-system	8	0,51 \$US	0%
kubecost	4	0,31 \$US	0%
default	6	<0,01 \$US	0%





kubecost

an IBM Company

Assets

Cloud Costs

Clusters

External Costs

Efficiency

Network

Collections

Reports

Savings

Follow

Teams

Diagnostics

Efficiency Report

View Kubernetes efficiency by various dimensions and time windows.



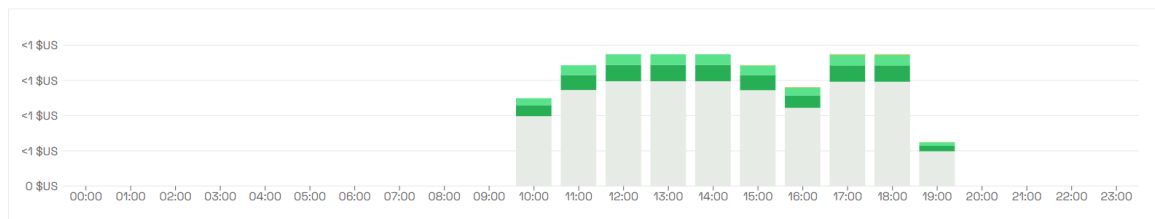
NAME	WORKLOAD IDLE	INFRA IDLE	TOTAL IDLE	TOTAL COST	CLUSTER EFFICIENCY
Total	0,16 \$US	0,66 \$US	0,81 \$US	0,82 \$US	0.0%
cluster-one	0,16 \$US	0,66 \$US	0,81 \$US	0,82 \$US	0.0%

Allocations

Upgrade License ↺ ⓘ

Cumulative cost for 13 July 2025 by namespace

Custom Namespace Filter Edit Save ...



NAME	CPU	GPU	RAM	PV	NETWORK	LB	SHARED	EFFICIENCY	TOTAL COST
Total	1,60 \$US	0,00 \$US	0,27 \$US	0,03 \$US	0,00 \$US	0,00 \$US	0,00 \$US	0%	1,89 \$US
__idle__	1,32 \$US	0,00 \$US	0,24 \$US	0,00 \$US	0,00 \$US	0,00 \$US	0,00 \$US	—	1,57 \$US (N/A)
kube-system	0,18 \$US	0,00 \$US	<0,01 \$US	0,00 \$US	0,00 \$US	0,00 \$US	0,00 \$US	0.0%	0,19 \$US (0,0%) ...
kubecost	0,09 \$US	0,00 \$US	0,01 \$US	0,03 \$US	0,00 \$US	0,00 \$US	0,00 \$US	0.0%	0,13 \$US (0,0%) ...
monitoring	0,00 \$US	0,00 \$US	<0,01 \$US	0,00 \$US	0,00 \$US	0,00 \$US	0,00 \$US	0.0%	<0,01 \$US (0,0%) ...
default	0,00 \$US	0,00 \$US	0,00 \$US	<0,01 \$US	0,00 \$US	0,00 \$US	0,00 \$US	0.0%	<0,01 \$US (0,0%) ... ⓘ

Après évaluation, nous avons constaté que les requêtes pour la création d'un compte a un coût fixe soit 0,22\$US ainsi nous pouvons avoir le tableau d'estimation de coût sur la création de compte suivant :

Coût	unitaire	Total
Temps réel par heure	0.22\$US	0.22\$US
Sur un mois	0.22\$US * 30	6.6\$US
sur un an	6.6\$US * 12	79.2\$US

De plus, la majorité de ce coût est liée aux ressources **réservées mais non utilisées** (idle), qui représentent **1,57 \$US** soit **plus de 83 %** du coût total.

Les namespaces réellement actifs sont :

- **kube-system** : 0,19 \$US
- **kubecost** : 0,13 \$US
- **monitoring** et **default** : négligeables (<0,01 \$US chacun)

On observe que **l'efficacité du cluster est de 0 %**, ce qui signifie qu'aucune ressource réservée (CPU/RAM) n'est pleinement exploitée par les workloads déployés.

Budgets

Budgets

Looks like you don't have any budgets yet. Why not create one?

Budget Info ×

Name

Type

Allocations

Spending cap and cadence

Budget Cap

Resets

Monthly

 on the

31st

Workloads

Cluster

cluster-one

×

Cluster

Search...

Add Filter

Actions New Action

Nous avons également fixé un budget de 200\$US sur kubecost afin d'avoir des alertes en cas de dépassement.

Cluster Details

cluster-one

Upgrade License ↻
Custom

Provider	custom
Version	N/A
Region	N/A
Health Score	N/A

Nodes	1
Namespaces	4
Pods	28
Controllers	16

Total Cost	1,89 \$US
Estimated Monthly Savings	
Workload Efficiency	N/A
Spending Trend	N/A

0,32 \$US Spent

Of 200,00 \$US Monthly Budget
Resets: 31/07/2025

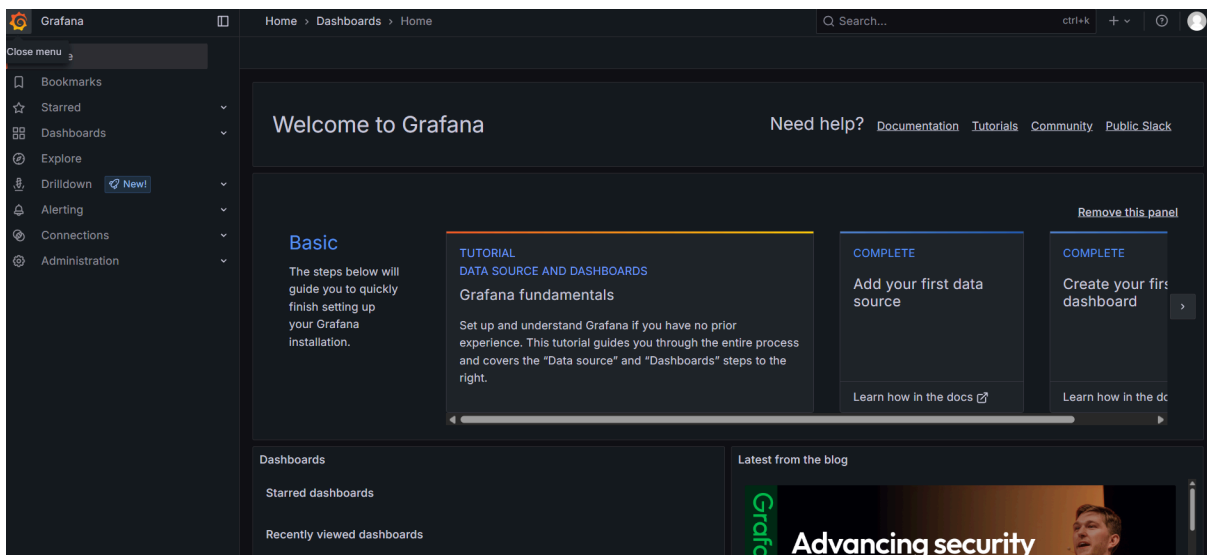
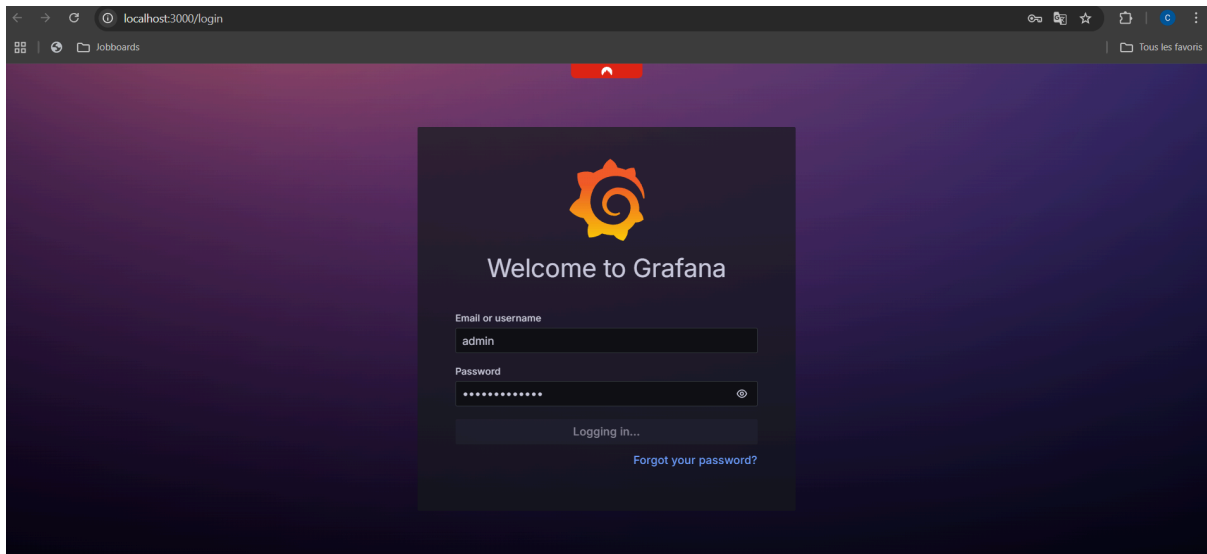
Efficiency	
NAME	• REQUESTED • USAGE
CPU	0% Efficiency
RAM	0% Efficiency
GPU	N/A Efficiency

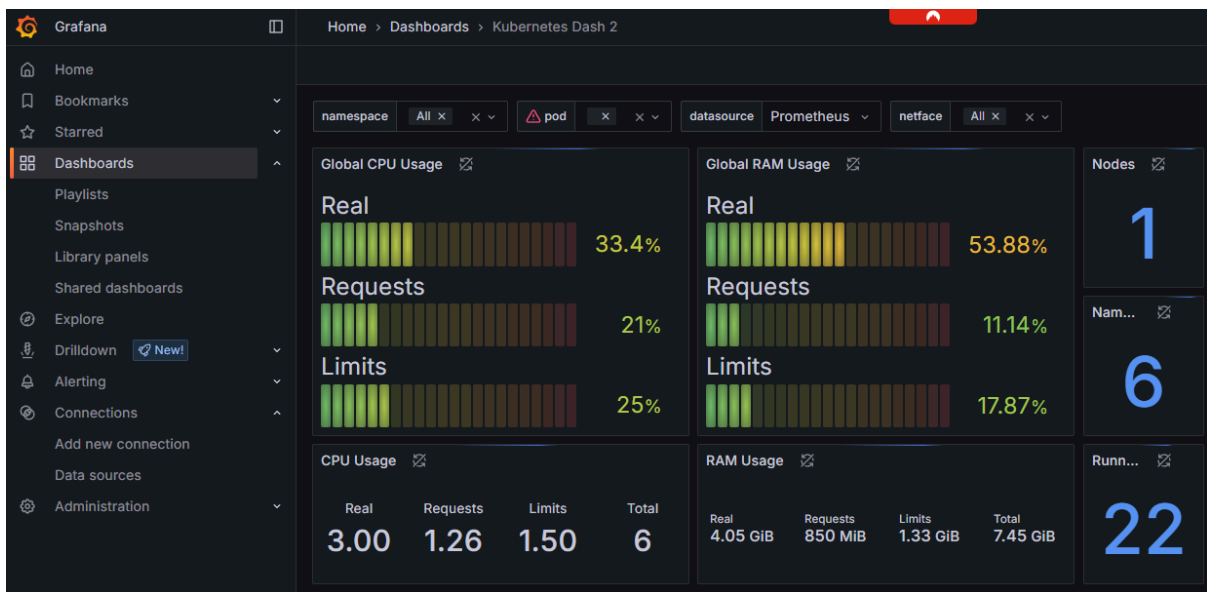
Cependant, il n'est pas possible d'avoir une prédiction de coût mensuel ou annuel sur kubecost comme sur d'autres plateformes comme AWS, Microsoft Azure etc ...

6. Surveillance avec Prometheus + Grafana

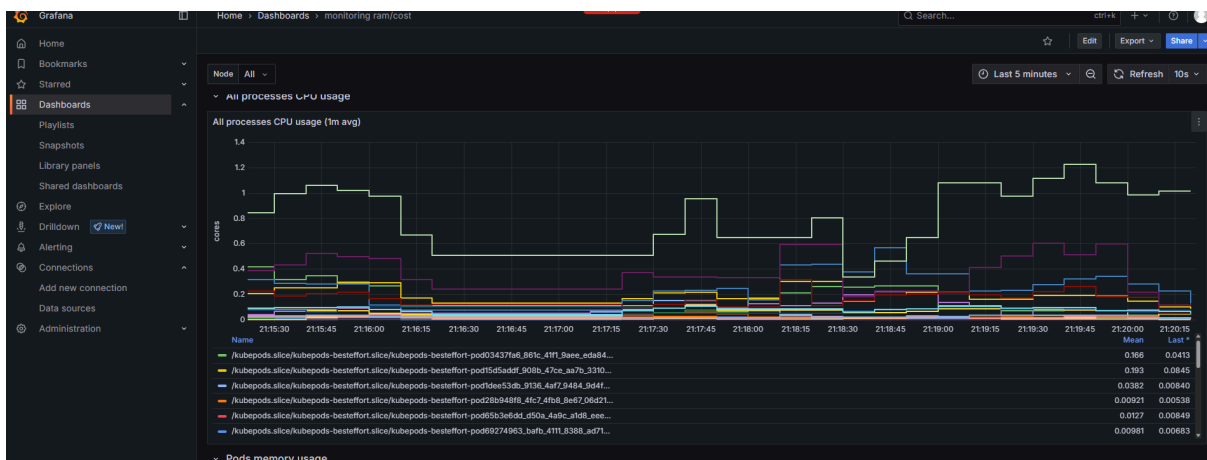
Pour assurer un suivi en temps réel de l'état de notre cluster Kubernetes ainsi que des applications déployées, nous avons mis en place une solution de monitoring basée sur la stack **Prometheus + Grafana**.

Prometheus a été installé via Helm, scrutant tous les pods/nodes grâce au service discovery Kubernetes. Grafana, également installé via Helm, a été configuré pour utiliser Prometheus comme source de données, permettant la création de dashboards personnalisés.

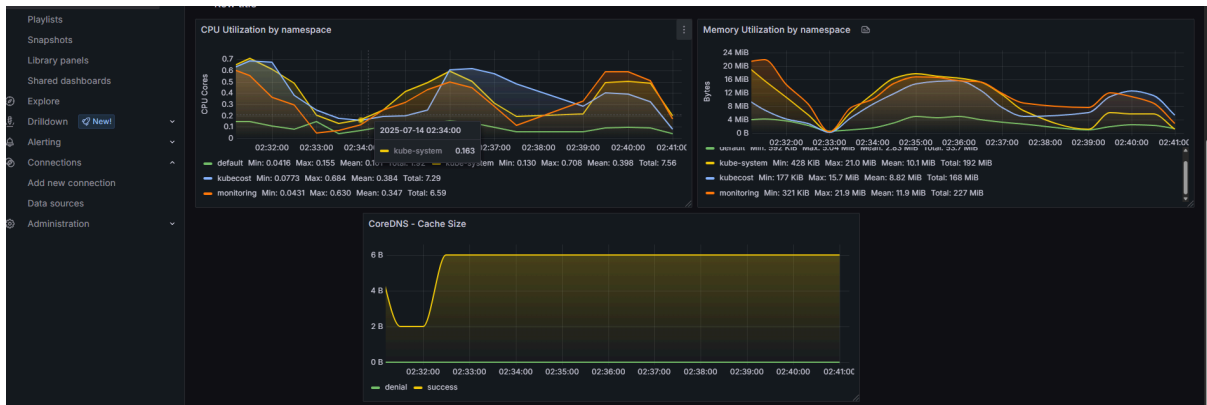




Ce dashboard Grafana offre une vue synthétique de l'utilisation globale du CPU et de la RAM dans le cluster Kubernetes. On y compare la consommation réelle avec les ressources demandées (requests) et allouées (limits), ce qui permet de détecter rapidement un surprovisionnement ou un risque de saturation. Ce type de visualisation est essentiel pour optimiser les configurations et garantir une utilisation efficace des ressources du cluster.



Ce graphique montre l'évolution de l'utilisation CPU par pod sur les dernières minutes. Il permet de comparer facilement la consommation des différents pods et d'identifier ceux qui sollicitent le plus de ressources. Ce suivi détaillé est utile pour repérer des comportements anormaux ou optimiser la répartition des charges dans le cluster.



Ce dashboard affiche l'utilisation du CPU et de la mémoire par namespace dans le cluster, ainsi que l'évolution de la taille du cache CoreDNS. Cette vue permet d'identifier rapidement les namespaces ou services les plus consommateurs en ressources, et de suivre le comportement du cache DNS, ce qui aide à anticiper d'éventuels problèmes de performance ou de saturation.



Ce graphique présente la limite CPU définie pour un pod spécifique (ici, un pod Kubecost). La courbe plate indique que la limite reste constante à 1,5 cœurs sur toute la période observée, ce qui permet de s'assurer que le pod ne dépassera jamais cette capacité CPU. Cette visualisation est utile pour vérifier que les limites de ressources sont correctement appliquées et respectées dans le cluster.

Annexes

