# docker-compose, deux exemples d'utilisation

**Denis Arrivault**
Inria de l'Université de Bordeaux - Service d'Expérimentation et de Développement
June 17, 2026

# Table des matières

# Contexte

## Inria et le SED

### Inria de l'Université de Bordeaux

- Créé en 2008 (Inria en 1967).
- 20 équipes projets ( 10 personnes par équipe);

### Le Service d'Expérimentation et de Développement

- Un service de support à la recherche;
- Un pour chaque centre Inria (9 SED donc);
- développements logiciels et plateformes expérimentales;
- +/- 20 IR à Bordeaux/Pau;

**A noter**
Les ingénieurs du SED sont des devs avec une expertise scientifique.

## L'Équipe ASTRAL

**ASTRAL équipe projet Inria commune avec Naval Group**

- Une partie des activités sont classifiées.
- Travail sur un serveur sécurisé accessible uniquement via VPN.
- Développement à distance en mode ‘remote’.
- Le code est partagé avec Naval via leur GitLab derrière un autre VPN.

**En résumé**

Environnement contraignant qui pose des soucis pour le déploiement de microservices dans les phases de développement.

# Cas 1: Elasticsearch/Kibana pour l'affichage de trafic maritime

Cas d'usage

## Problématique

- Base SQLite de données AIS issues d'enregistrements de traffics maritimes (aisstream).
- Besoin d'un outil ponctuel d'affichage dynamique pour ces données.
- La base est sur le serveur derrière VPN.

## Solutions techniques

- Charger les données en RAM + outils de visu : trop lourd (plusieurs GB de données).
- Utiliser Elasticsearch (ES) pour délivrer la donnée rapidement. ✅
- Utiliser Kibana (avec ES) qui a un module d'affichage de carte assez complet. ✅

## Le choix de docker-compose

### Critères de choix

- Un seul serveur pour le dev/calcul avec plusieurs développeurs : volonté de ne pas surcharger le serveur inutilement.
- Pas d'administrateur dédié, le serveur est géré par les devs : volonté de ne pas se perdre dans de l'administration compliquée.
- Docker est déjà installé car utilisé dans la partie CI du projet.
- 2 applicatifs à déployer (Elasticsearch et Kibana).
- Cas simple, on trouve une multitude de docker-compose.yml sur le web.

**Note**

En plus, on n'a pas de soucis de sécurité puisque le serveur est derrière le VPN et les données exposées ne sont pas sensibles.

## Configuration

**docker-compose.yml**

```yaml
services:
  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:8.13.0
    container_name: elasticsearch
    environment:
      - discovery.type=single-node
      - xpack.security.enabled=false
    ports:
      - "9200:9200"
    ulimits:
      memlock:
        soft: -1
        hard: -1
    volumes:
      - es-data:/usr/share/elasticsearch/data
```

```
  kibana:
    image: docker.elastic.co/kibana/kibana:8.13.0
    container_name: kibana
    depends_on:
      - elasticsearch
    environment:
      - ELASTICSEARCH_HOSTS=http://elasticsearch:9200
    ports:
      - "5601:5601"

volumes:
  es-data:
```

Explication

**Fichier de configuration basique**

- Deux services : Elasticsearch (ES) et Kibana.
- Variables d'environnement :
  - `discovery.type` : `single-node` pour une machine de développement ou un serveur de test local (pas de découverte de cluster),
  - `xpack.security.enabled` : `false` pour désactiver l'authentification et le chiffrement HTTP.
  - `ELASTICSEARCH_HOSTS` : indique à Kibana où trouver ES.
- Limitation de ressources : les limites de verrouillage de la mémoire sont désactivées.
- Mappage des ports sur localhost : 9200 pour ES et 5601 pour Kibana.

- Un volume persistant pour les données d'ES monté sur /usr/share/elasticsearch/data dans le node ES.

### Résultat

On lance les services sur le serveur:

- `docker-compose up -d`

On transfère les ports du serveur sur notre machine (Kibana sur 5602 ici en l'occurrence):

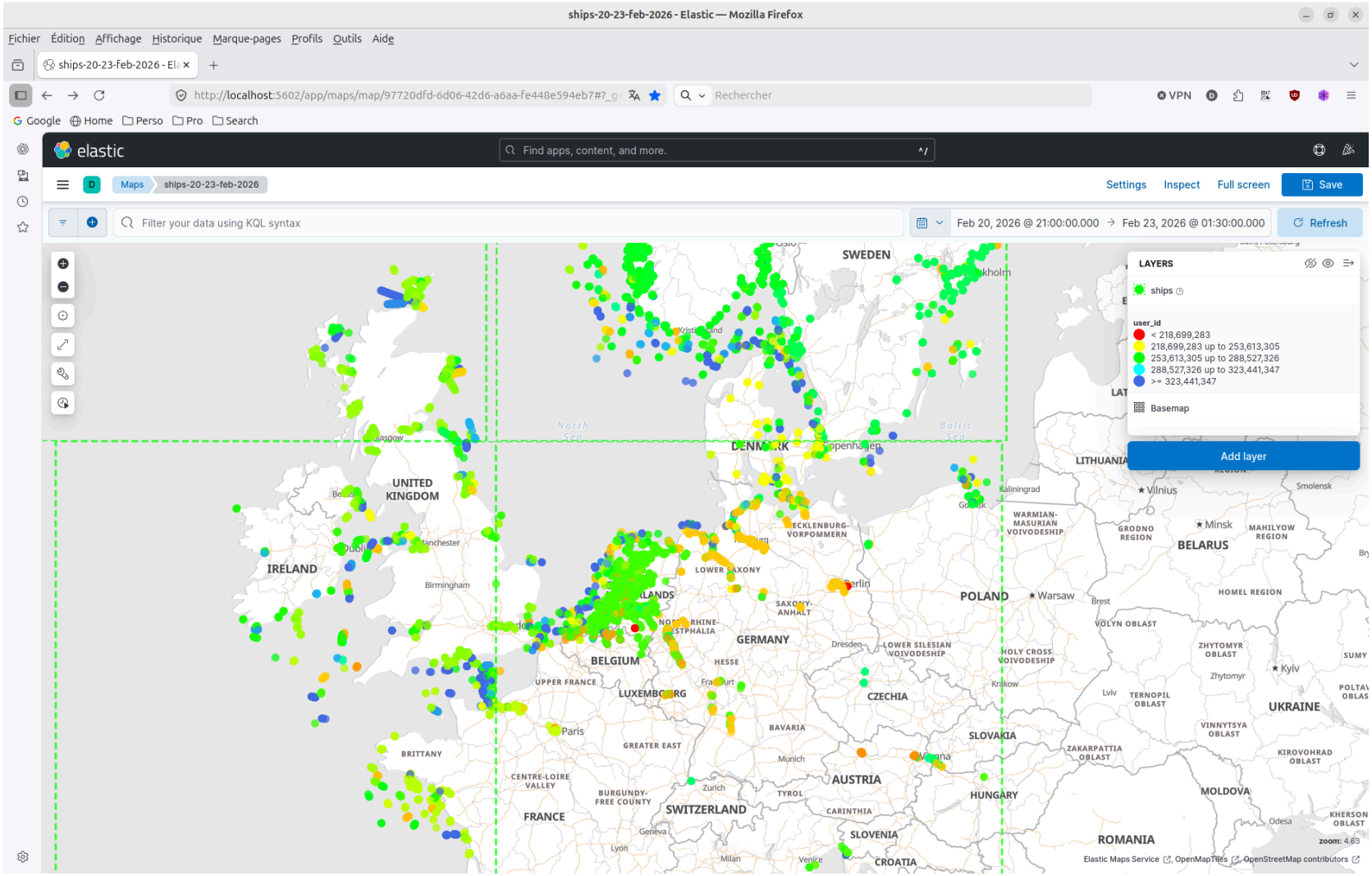


Figure 1: Données aisstream.io.

# Cas 2: Déploiement d’un LLM avec Ollama sous contrainte d’isolation

Cas d'usage

**Problématique**

- Développement en remote sur le serveur avec VSCode.
- Pas d'IA intégrée pour éviter toute fuite de donnée (Copilote désinstallé et/ou désactivé).
- On souhaite utiliser un LLM isolé, au moins pour la completion de code et le chat.
- On a à disposition un serveur avec deux Tesla V100 en 16 GB.
- Consignes de Naval : modèles en local en formats safetensors et plutôt Ollama qu'ils utilisent chez eux.

**Question**

Comment déployer avec Ollama un modèle qui soit isolé du réseau?

## Le choix de docker-compose

### Critères de choix

- Isoler Ollama du réseau => facile avec docker-compose.
- Déploiement d'un proxy inverse pour exposer les modèles => Nginx.
- Ajout simple d'un applicatif open-webui.

### Configuration

- Un node isolé `ollama-runner` sans accès internet qui joue les modèles.
- Un node `open-webui`.
- Un node `proxy` avec Nginx pour exposer `ollama-runner` et `open-webui` sur un réseau public.

## Configurations

**LLM disclosure**

Ces fichiers ont été réalisés avec l'aide souvent utile (et parfois contre-productive) de perplexity.ai...

**docker-compose.yml**

```yaml
services:
 ollama-runner:
   image: ollama/ollama:latest
   container_name: ollama-runner
   command: ["serve"]
   volumes:
     - ollama_models:/root/.ollama
     - ../models:/models
   restart: unless-stopped
   deploy:
     resources:
```

```
      reservations:
        devices:
          - driver: nvidia
            device_ids: ["0","1"]
            capabilities: [gpu]
  networks:
    ollama-runner:
      ipv4_address: 172.19.1.82
  environment:
    - OLLAMA_HOST=0.0.0.0
    - OLLAMA_KEEP_ALIVE=-1
```

**Node ollama-runner**

- Volume persistant ollama_models pour les modèles enregistrés.
- Volume partagé avec le répertoire où les modèles ont été téléchargés.
- Déclaration explicite de 2 GPU (sinon on a des soucis d'utilisation...).
- Une IP sur le réseau interne.
- La variable OLLAMA_KEEP_ALIVE=-1 assure que le modèle reste en mémoire.

## Configuration

**docker-compose.yml**

```yaml
open-webui:
  image: ghcr.io/open-webui/open-webui:main
  container_name: open-webui
  depends_on:
    - ollama-runner
  networks:
    - ollama-runner
  environment:
    - OLLAMA_BASE_URL=http://ollama-runner:11434
    - WEBUI_AUTH=true
    - WEBUI_SECRET_KEY=${WEBUI_SECRET_KEY}
    - ENABLE_TOOL_GRANT=false
    - ENABLE_RAG=false
  volumes:
    - open-webui:/app/backend/data
```

**Node open-webui**

- Il dépend de ollama-runner et partage son réseau.
- OLLAMA_BASE_URL permet de trouver Ollama avec les bonnes variables (WEBUI_SECRET_KEY dans un .env à la racine).
- Un volume persistant qui permet de garder l'historique et les configurations.

## Configuration

**docker-compose.yml**

```yaml
proxy:
  image: nginx:stable-alpine3.20
  container_name: proxy
  volumes:
    - ./nginx:/etc/nginx/conf.d
  networks:
    ollama-runner:
      ipv4_address: 172.19.1.83
    webui-access:
      ipv4_address: 172.19.2.99
  logging:
    driver: "json-file"
    options:
      max-size: "10m"
      max-file: "3"
  restart: always
```

```
  ports:
    - "11434:11434"   # Ollama API via Nginx
    - "3000:3000"     # Open WebUI via Nginx
```

**Node proxy**

- IPs sur le réseau ollama-runner et sur le réseau exposé webui-access.
- On expose Ollama sur le port 11434 et open-webui sur le port 3000.

## Configuration

**docker-compose.yml**

```yaml
volumes:
 ollama_models:
   name: ollama_models
 open-webui:
   name: open-webui

networks:
 ollama-runner:
   name: ollama-runner
   driver: bridge
   internal: true
   ipam:
     config:
       - subnet: 172.19.1.80/28
 webui-access:
   name: webui-access
```

```
    driver: bridge
    ipam:
      config:
        - subnet: 172.19.2.0/24
          gateway: 172.19.2.1
```

**Volumes et réseaux**

- 2 volumes.
- Le driver `bridge` créé un réseau pont privé entre les conteneurs et le host.
- 2 réseaux créés par de le driver `bridge` dont un isolé de l'extérieur (`internal: true`).

## Configuration

**nginx/ollama.conf**

```
 Nginx running in "proxy" container

# Open WebUI on port 3000
server {
    listen 3000;
    server_name _;

    # Proxy all UI requests to the open-webui container (port 8080 in container)
    location / {
        proxy_pass http://open-webui:8080;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
```

```
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_buffering off;
    }
}


# Ollama API on port 11434
server {
    listen 11434;
    server_name _;

    location / {
        proxy_pass http://172.19.1.82:11434;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_buffering off;
    }
}
```

## Nginx configuration

Là, c'est un peu plus opaque pour moi...

Nginx écoute en même temps :

- 0.0.0.0:3000 pour l'interface WebUI.
- 0.0.0.0:11434 pour l'API Ollama.

Ces ports sont exposés vers l'extérieur via les ports du docker-compose.yml (ex: "3000:3000", "11434:11434").

## Résultat

- `docker-compose up -d`.
- Le chargement des modèles au format safetensors nécessite une conversion en GGUF FP16 puis une quantification en Q4_K_M.
- Ensuite les modèles sont accessibles via le service Ollama sur le port 11434 ou dans open-webui sur le port 3000.
- On transfère les ports sur la machine cible (open-webui est sur le port 8080 dans l'exemple ci-dessous).

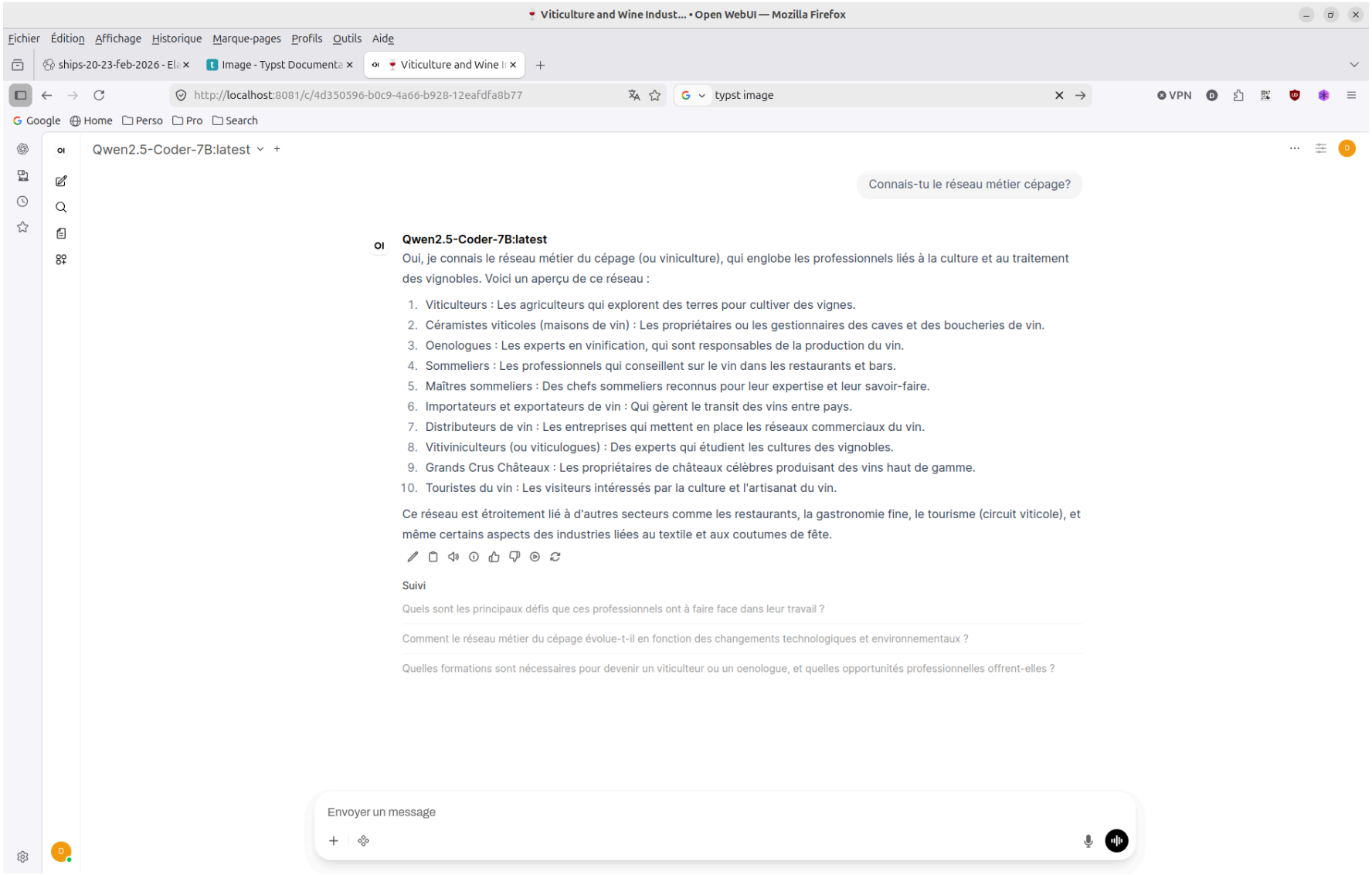


Figure 2: Open-webui.

# Merci

Questions?