

## **8INF876 – Conception et architecture des systèmes d'infonuagique**

Déploiement d'une application de détection d'armes à feu

# TABLE DES MATIERES

Table des matières.....	1
Liste des figures .....	3
Liste des acronymes .....	4
Chapitre 1 : Introduction .....	5
1.1 Introduction .....	5
1.2 Problématique .....	5
On-permise (Sur-site) .....	5
PaaS .....	6
Serverless.....	6
Conteneurisation .....	6
1.3 Description du Développement .....	6
Création du modèle de détection d'armes .....	6
Création de l'API .....	7
Création du front end .....	7
Chapitre 2 : Approche de conception.....	8
2.1 Architecture du système .....	8
Le front-end .....	8
Le back-end.....	8
2.2 Outils utilisés.....	9
Front-end .....	9
Back-end .....	10
Conclusion .....	11
Chapitre 3 : Implémentation et Tests.....	12
3.1 Implémentation .....	12
Création du modèle .....	12
Création de l'API .....	13

Dockerization de l'API.....	14
Hébergement du conteneur sur Amazon ECR .....	16
Déploiement du conteneur sur Amazon ECS .....	17
Création et déploiement de l'interface .....	26
3.2 Tests .....	31
Conclusion .....	32
Chapitre 4 : Conclusion.....	33
Leçons apprises.....	33
Problèmes rencontrés .....	33
Perspectives.....	33
Conclusion .....	35

# LISTE DES FIGURES

Figure 1 Architecture générale du système .....	8
Figure 2 Architecture spécifique du système .....	9
Figure 3 Ensemble de données d'entraînement du modèle.....	12
Figure 4 Image originale contenant une arme .....	13
Figure 5 Image prédite par le modèle .....	13
Figure 6 Route /predict de l'API .....	14
Figure 7 Dockerfile pour l'API .....	15
Figure 8 Construction de l'image.....	15
Figure 9 Test de l'image.....	16
Figure 10 Test du conteneur.....	16
Figure 11 Commandes pour héberger le conteneur sur Amazon ECR .....	16
Figure 12 Push de l'image sur Amazon ECR .....	17
Figure 13 Conteneur hébergé sur Amazon ECR .....	17
Figure 14 Création d'une tâche : Configuration du conteneur .....	18
Figure 15 Création d'une tâche : Environnement et ressources.....	19
Figure 16 Tâche créée.....	20
Figure 17 Création d'un service : Environnement.....	21
Figure 18 Création d'un service : Configuration du déploiement.....	22
Figure 19 Création du service : Networking .....	23
Figure 20 Groupe de sécurité .....	24
Figure 21 Service créé.....	25
Figure 22 Test de l'API déployée .....	26
Figure 23 Interface de l'application.....	26
Figure 24 Autorisation de l'accès au public.....	27
Figure 25 L'object ownership .....	28
Figure 26 Bucket elasticbeanstalk .....	29
Figure 27 création d'un environnement server web beanstalk .....	29
Figure 28 Création d'un environnement Beanstalk : Choix de la plateforme.....	30
Figure 29 Création d'un environnement Beanstalk : Téléversement du code .....	30
Figure 30 Création de l'environnement beanstalk : Fin .....	31

## LISTE DES ACRONYMES

PaaS	Platform as a Service
AWS	Amazon Web Services
ECR	Elastic Container Registry
ECS	Elastic Container Service
EC2	Elastic Compute Cloud
CDN	Content Delivery Network
ACL	Access Control List

# CHAPITRE 1 : INTRODUCTION

## 1.1 Introduction

Les progrès récents en Intelligence Artificielle ont ouvert d'immenses possibilités. Des robots organisateurs de stocks intelligents aux simples algorithmes de prédiction du prix d'une maison, ou encore un chatbot qui sait presque tout, l'Intelligence Artificielle semble ne pas avoir de limites.

Pour ce projet, mon but est d'adresser un problème de sécurité : la présence d'armes à feu. Les armes à feu peuvent représenter un danger dans certains lieux publics tels que les aéroports, les parcs publics, les centres commerciaux, et bien d'autres. Cependant, avoir des personnes postées à toutes les heures pour surveiller les passants requiert beaucoup de ressources. De plus, c'est une tâche fastidieuse et difficile pour un humain.

Ainsi, il devient tout naturel de se tourner vers l'Intelligence Artificielle pour cette tâche automatisable. J'ai donc décidé de créer un modèle de détection d'armes à feu. Un modèle est tout simplement un algorithme qui a été entraîné sur des données afin d'apprendre à accomplir une tâche spécifique : dans notre cas, la détection d'armes à feu. Ce modèle est capable d'analyser des images ainsi que des vidéos en temps réel afin de détecter des armes à feu.

## 1.2 Problématique

Le but est de déployer le système. Le modèle sera servi via une API, qui, sera elle-même hébergée dans le cloud. L'interface, qui est une simple application web sera aussi hébergée dans le cloud.

Pour se faire, plusieurs solutions s'offrent à nous.

### 1.2.1 Critique des méthodes existantes

Pour le déploiement de l'API, nous avons : le déploiement sur site, le Serverless, les PaaS ou encore les conteneurs. Ces méthodes présentent certains des avantages, mais aussi des inconvénients qui font qu'elles ne sont pas les plus optimales pour ce projet.

#### *On-permise (Sur-site)*

Déployer le modèle sur site passe par l'utilisation de nos propres serveurs. Cependant, configurer et gérer ses propres serveurs et machines virtuelles, requiert beaucoup

d'efforts et de l'expertise. De plus, cela coûte chers car les modèles de Deep Learning requièrent beaucoup de ressources.

### *PaaS*

Les PaaS (Platform as a Service) sont des plateformes qui permettent de déployer son code sans avoir à se soucier de l'infrastructure. Bien qu'efficaces, les PaaS ne sont pas assez flexibles car elles nous limitent dans le choix des technologies. De plus, elles ont une scalabilité limitée.

### *Serverless*

Le Serverless consiste à exécuter son code dans le cloud sans avoir à s'occuper de l'infrastructure, c'est la méthode avec le plus haut niveau d'abstraction car tout est géré par le fournisseur cloud. Cependant, le Serverless présente une latence supplémentaire dû au temps de démarrage, ce qui peut ralentir les opérations. De plus, il n'est pas adapté aux applications demandant beaucoup de ressources.

## 1.2.2 Solution proposée : La conteneurisation

La conteneurisation consiste à placer une application, ainsi que toutes ses dépendances dans un conteneur. Les conteneurs sont simples à utiliser et ont l'avantage de pouvoir être mis à l'échelle et s'adapter en fonction de la demande. Cependant, l'orchestration, la mise à l'échelle, la gestion de l'infrastructure des conteneurs requièrent une certaine expertise.

Pour ce projet, nous utilisons les conteneurs parce qu'ils sont simples à créer, nous donnent le contrôle sur les technologies et langages de développement, tout en nous évitant de gérer le système d'exploitation. En outre, leur mise à l'échelle est plus simple, ce qui présente un gros avantage en Deep Learning.

## 1.3 Description du développement

Le projet porte sur la création d'une application de détection d'armes à feu et son déploiement dans le cloud.

Pour se faire, on procède de la façon suivante :

### 1.3.1 Création du modèle

Avant tout, il faut créer un modèle capable de reconnaître des armes à feu. Pour se faire, on collecte des images annotées contenant des armes à feu. Une annotation contient la classe de l'objet présent dans l'image (0 pour arme) et les coordonnées du rectangle entourant l'objet. Ensuite, on entraîne un modèle à l'aide d'un algorithme appelé « YOLOv8 » sur ces données. YOLOv8 est le l'algorithme le plus performant en Avril 2023 pour notre tâche.

### 1.3.2 Création de l'API

Le modèle entraîné est servi grâce à une API qui fournit des routes afin que la partie front end puisse y accéder.

### 1.3.3 Création du front end

Le front end est simplement une interface web permettant d'interagir avec l'API afin de faire des prédictions.

### 1.3.4 Déploiement

Une fois que le front-end et le back-end ont été implémentés, on les déploie dans le cloud. Nous divisons ce processus en deux éléments distincts :

- Pour l'API : La conteneurisation, le déploiement du conteneur, l'orchestration des conteneurs afin de servir l'API.
- Pour le front-end : L'utilisation d'un CDN pour servir l'application publiquement.

## 1.4 Conclusion

Dans ce chapitre, nous avons eu un aperçu du projet. Ensuite, nous avons vu pourquoi les conteneurs étaient la solution la plus optimale pour notre projet. Finalement, nous avons vu de façon brève les étapes de la réalisation du projet.



## CHAPITRE 2 : APPROCHE DE CONCEPTION

Dans ce chapitre, je présente l'architecture du système. Autrement dit, tous les composants nécessaires à la création et à la mise en production du modèle. D'abord, on verra l'architecture générale, ensuite les technologies utilisées pour implémenter cette architecture.

### 2.1 Architecture du système

Le système est composé de 5 éléments principaux : une interface, un CDN, une API, un conteneur et un repository de conteneurs.

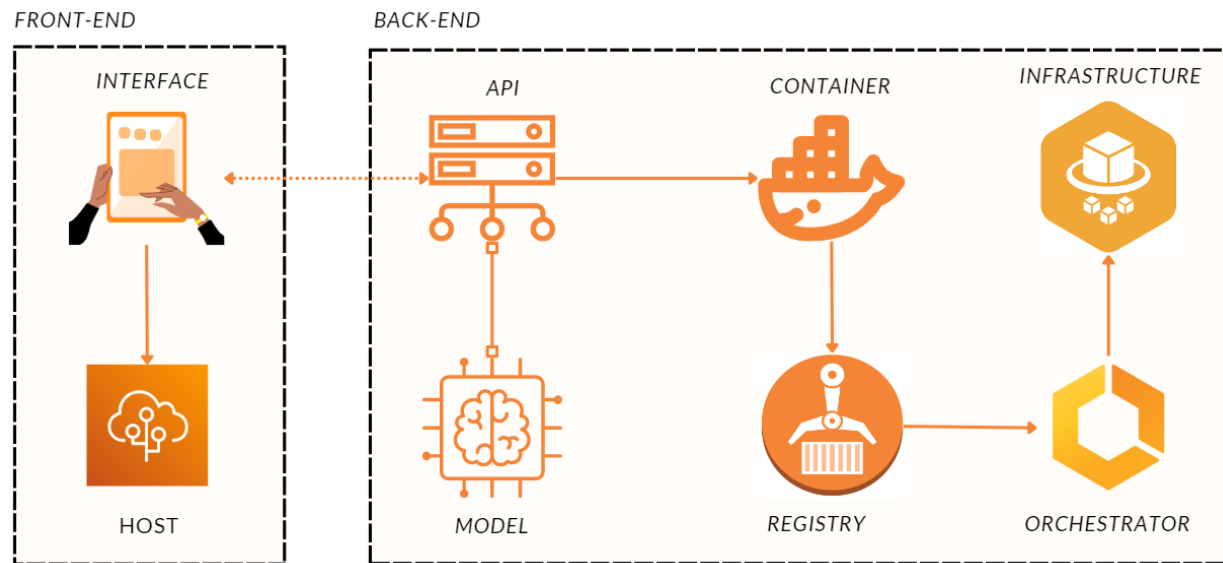


Figure 1 Architecture générale du système

#### 2.1.1 Le front-end

##### *L'interface*

L'interface est une simple page web qui nous permet d'interagir avec l'application.

##### *L'hébergeur*

Pour héberger l'application web, nous utilisons une PaaS, car encore une fois, on pense en termes de scalabilité. De plus, une PaaS nous évite de nous soucier de l'infrastructure.

#### 2.1.2 Le back-end

## API

L'API développée avec Flask permet d'accéder au modèle de détection d'objets.

## Conteneur

Le conteneur contient l'API. Servant ainsi à créer un environnement isolé.

## Registry

Le container repository, ou dépôt de conteneur en français, est utilisé pour héberger le conteneur dans le cloud.

## Orchestrator

L'orchestrator (orchestrateur), permet de manager et déployer les conteneurs Docker sur un cluster.

## Infrastructure

Le cluster a besoin d'être placé quelque part. C'est pourquoi on a besoin d'une infrastructure pour allouer les ressources au cluster.

## 2.2 Outils utilisés

De façon plus spécifique, l'architecture est présentée dans le schéma ci-dessous. On peut voir les outils utilisés sur l'image ci-dessous :

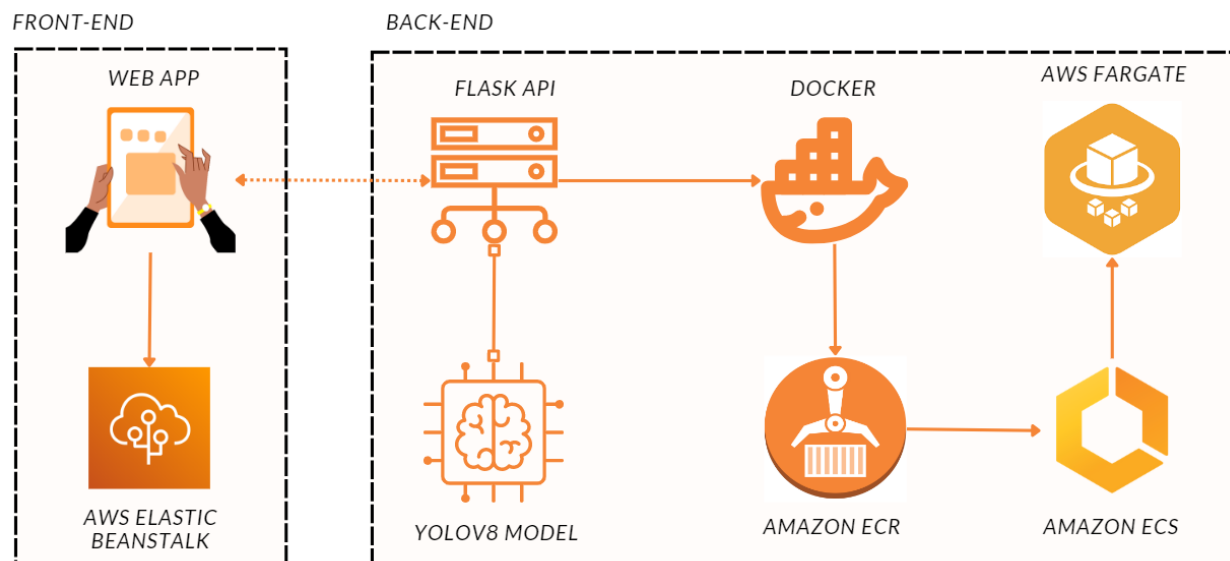


Figure 2 Architecture spécifique du système

## Front-end

Pour le front-end, deux outils sont utilisés.

### *Application web*



La partie front-end, ou l'application web est développée en HTML, CSS, et JavaScript.

### *AWS Elastic Beanstalk*



AWS Elastic Beanstalk est une PaaS (Plateform as a Service) spécialement conçue pour le déploiement d'applications web. On l'utilisera pour héberger l'application web. Beanstalk, gère entièrement l'infrastructure et l'auto-scaling. Puisqu'on a pas besoin d'un grand contrôle sur l'infrastructure du front-end contrairement à l'API, Elastic Beanstack est un excellent choix.

## Back-end

### *Flask*



Le modèle est contenu dans une API Flask qui fournit des routes. Ces routes (ou points de terminaison) seront utilisées par la partie front-end pour communiquer avec l'API afin d'analyser des images ou vidéos.

### *Docker*



Le conteneur de l'API sera construit à l'aide de la technologie Docker. L'avantage d'avoir un conteneur spécialement pour l'API est que la mise à l'échelle se fera indépendamment du front-end. Ce qui est avantageux puisque le front end et l'API n'auront pas les mêmes demandes en ressources.

### *Amazon ECR*



Amazon ECR (Elastic Container Registry) est un registre de conteneur. On y héberge le conteneur afin que les autres services comme Amazon ECS (Elastic Container Service) puissent y accéder.

### *Amazon ECS*



Amazon ECS (Elastic Container Services), est un orchestrateur de conteneurs. Il permet de manager et déployer les conteneurs Docker sur un cluster.

### *AWS Fargate*



Amazon Fargate est une infrastructure qui permet d'héberger notre cluster ECS. Ce service permet d'allouer les ressources dont le cluster a besoin pour fonctionner. Contrairement à un Amazon EC2, on n'a pas besoin de gérer manuellement l'infrastructure.

## Conclusion

Dans ce chapitre, nous avons élaboré une architecture efficace pour notre système. Amazon nous fournit une abondance de services, ainsi c'est un choix idéal pour ce projet.

# CHAPITRE 3 : IMPLEMENTATION ET TESTS

Le projet porte sur la création d'une application de détection d'armes. Cette application sera hébergée dans le cloud. Ainsi, nous utiliserons plusieurs technologies.

## 3.1 Implémentation

La réalisation de ce projet est passée par plusieurs étapes que nous pouvons regrouper en quatre étapes principales : La création du modèle, la création de l'API, la conteneurisation et le déploiement de l'API, la création de l'interface et le déploiement de l'interface.

### Création du modèle

Le modèle a été entraîné en utilisant l'algorithme « YOLOv8 » sur un ensemble de données (dataset) contenant des images d'armes à feu.

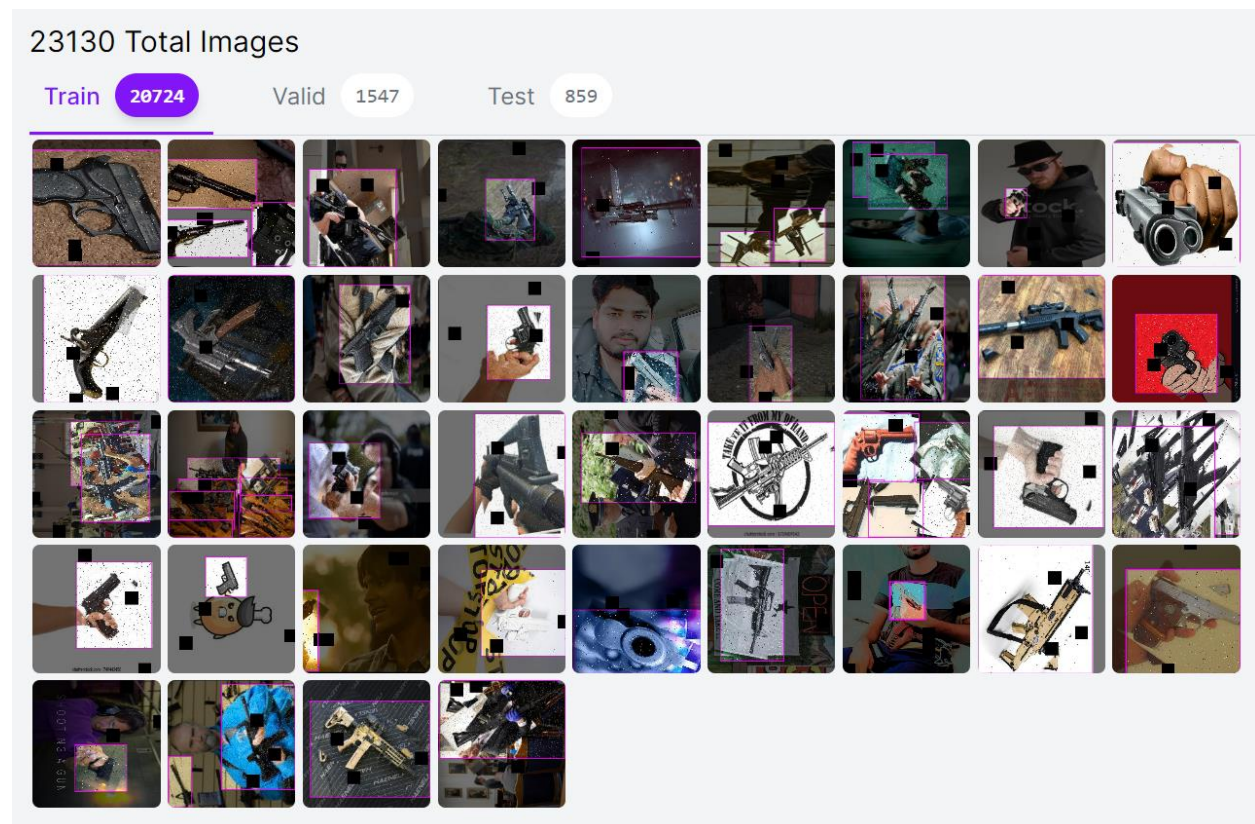


Figure 3 Ensemble de données d'entraînement du modèle.

Lorsqu'on lui envoie de nouvelles images, le modèle prédit s'il y a une arme et dessine le résultat. On peut le voir sur les deux images ci-dessous :



Figure 4 Image originale contenant une arme



Figure 5 Image prédite par le modèle

## Création de l'API

L'API développée avec Flask contient une route de type « POST ». Cette route se situe au point d'accès « /predict ». Elle effectue plusieurs étapes :

1. Récupère et lit le fichier téléchargé.
2. Transforme le fichier en image compatible avec le modèle.
3. Effectue des prédictions sur l'image.
4. Encode et retourne les résultats de la prédiction en base64 string.

L'image résultante est convertie en base64 afin qu'on puisse l'envoyer à travers le réseau.

```
@app.route('/predict', methods=['POST'])
def predict():
    file = request.files['file']

    image = processor.process_image(file)

    predicted_image = model.predict_image(image)

    encoded_image_data = processor.image_to_base64(predicted_image)

    return str(encoded_image_data)
```

**Figure 6 Route /predict de l'API**

## Dockerization de l'API

Finalement, l'API est placée dans un conteneur docker. Le Dockerfile suit plusieurs étapes :

- Spécifie une image légère de python « python-3.11.3-slim-buster »,
- Spécifie le dossier de travail (WORKDIR)
- Copy et installe les dépendances. On utilise `--no-cache-dir` pour ne pas utiliser de cache afin de garder le conteneur léger.
- Copy l'application dans le dossier /app
- Expose le port 5000
- Définit l'environnement sur production.
- Lance l'application de façon accessible par n'importe quelle adresse IP (0.0.0.0).

```

FROM python:3.11.3-slim-buster

WORKDIR /app

COPY requirements.txt /app/

RUN pip3 install --no-cache-dir -r requirements.txt

COPY . /app

EXPOSE 5000

ENV FLASK_ENV=production

CMD [ "python3", "-m" , "flask", "run", "--host=0.0.0.0"]

```

Figure 7 Dockerfile pour l'API

L'image est maintenant prête à être construite :

```

PS D:\UQAC\Guns-Detections-Project\API> docker build -t gun-detection-api:latest .
[+] Building 599.0s (11/11) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 622B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/python:3.11.3-slim-buster
=> [1/6] FROM docker.io/library/python:3.11.3-slim-buster@sha256:8c8998725afb2b71
=> [internal] load build context
=> => transferring context: 1.54MB
=> CACHED [2/6] WORKDIR /app
=> CACHED [3/6] COPY requirements.txt /app/
=> [4/6] RUN apt-get update && apt-get install -y libgl1-mesa-glx
=> [5/6] RUN pip3 install --no-cache-dir -r requirements.txt
=> [6/6] COPY . /app
=> exporting to image
=> => exporting layers
=> => writing image sha256:42c7c574aeefc9b0faafd2a14559b773cf48e831f1725495ca8893
=> => naming to docker.io/library/gun-detection-api:latest

```

Figure 8 Construction de l'image

On peut lancer un test pour vérifier que tout se passe bien :



```
PS D:\UQAC\Guns-Detections-Project\API> docker run -p 5000:5000 gun-detection-api:latest
'FLASK_ENV' is deprecated and will not be used in Flask 2.3. Use 'FLASK_DEBUG' instead.
'FLASK_ENV' is deprecated and will not be used in Flask 2.3. Use 'FLASK_DEBUG' instead.
'FLASK_ENV' is deprecated and will not be used in Flask 2.3. Use 'FLASK_DEBUG' instead.
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://172.17.0.3:5000
Press CTRL+C to quit
172.17.0.1 - - [22/Apr/2023 19:00:49] "GET / HTTP/1.1" 200 - .0 is the tag
```

Figure 9 Test de l'image

L'API est bien lancée, il est temps de la déployée comme on peut le voir.

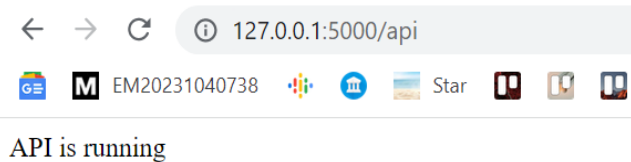


Figure 10 Test du conteneur

## Hébergement du conteneur sur Amazon ECR

Maintenant, il faut héberger l'image étape consiste à déployer le conteneur sur Amazon ECR. Les commandes à exécuter sont disponibles sur [Amazon ECR](#).

1. Retrieve an authentication token and authenticate your Docker client to your registry.

Use the AWS CLI:

```
aws ecr get-login-password --region us-east-2 | docker login --username AWS --password-stdin
amazonaws.com
```

Note: If you receive an error using the AWS CLI, make sure that you have the latest version of the AWS CLI and Docker installed.

2. Build your Docker image using the following command. For information on building a Docker file from scratch see the instructions [here](#). You can skip this step if your image is already built:

```
docker build -t gun-detection-api .
```

3. After the build completes, tag your image so you can push the image to this repository:

```
docker tag gun-detection-api:latest amazonaws.com/gun-detection-api:latest
```

4. Run the following command to push this image to your newly created AWS repository:

```
docker push amazonaws.com/gun-detection-api:latest
```

Figure 11 Commandes pour héberger le conteneur sur Amazon ECR

```

PS D:\UQAC\Guns-Detections-Project\API> docker tag gun-detection-api:latest 934857846971.dkr.ecr.us-east-2.amazonaws.com/gun-detection-api:latest
PS D:\UQAC\Guns-Detections-Project\API> docker push 934857846971.dkr.ecr.us-east-2.amazonaws.com/gun-detection-api:latest
The push refers to repository [934857846971.dkr.ecr.us-east-2.amazonaws.com/gun-detection-api:latest]
816f469395d6: Pushed
9468123f4c72: Pushed
17c27498df34: Pushed
4213ec48bbd4: Pushed
611ef6bd92ae: Pushed
1ea6e28a93aa: Pushed
12bb686bbe7f: Pushed
c327d855c71f: Pushed
ccc60df26c61: Pushed
61a5c84a1270: Pushed
latest: digest: sha256:99dee0df75187acccb87729535e56383669e1b29cc26d11f3a294de1895d9fc0
PS D:\UQAC\Guns-Detections-Project\API>

```

Figure 12 Push de l'image sur Amazon ECR

On peut voir que le conteneur a été poussé.

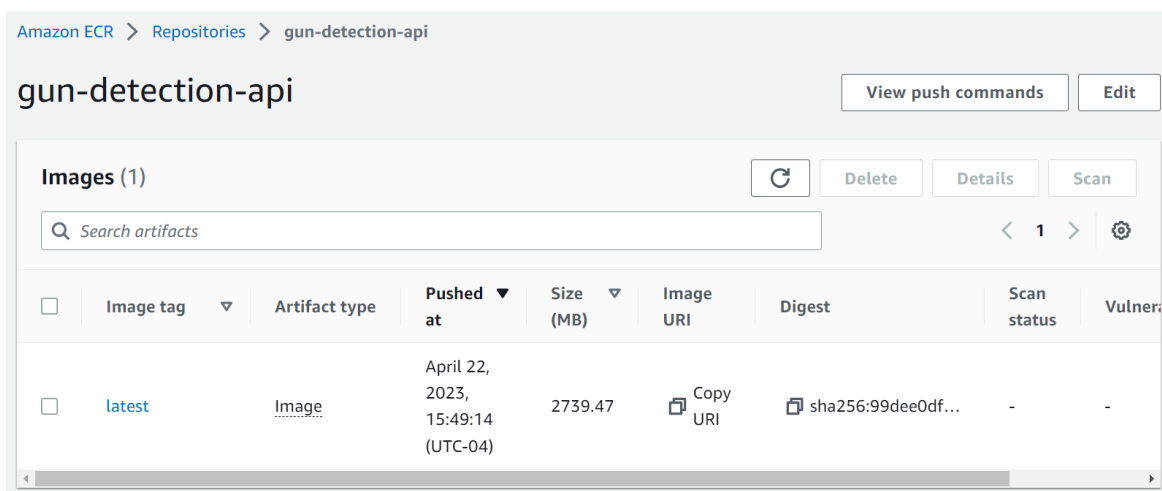


Figure 13 Conteneur hébergé sur Amazon ECR

## Déploiement du conteneur sur Amazon ECS

Pour déployer notre API sur Amazon ECS, on utilise le conteneur créé et hébergé précédemment sur ECR. ECS a besoin de deux éléments :

- Une tâche : qui spécifie l'image du conteneur à lancer ainsi que ses ressources ;
- Un service : qui assure la gestion de la tâche notamment son démarrage et son auto-scaling.

### Création de la tâche

Tout d'abord, il faut accéder à [Amazon ECS](#) et créer une tâche : « Task Definition ».

Cela se fait en deux étapes. D'abord on spécifie le conteneur auquel ECS doit accéder ainsi que quelques paramètres supplémentaires comme le port.

**Task definition configuration**

Task definition family

[Info](#)

Specify a unique task definition family name.

gun-detection-task

Up to 255 letters (uppercase and lowercase), numbers, hyphens, and underscores are allowed.

**Container - 1**

[Info](#)

Essential container

Remove

Container details

Specify a name, container image, and whether the container should be marked as essential. Each task definition must have at least one essential container.

Name

gun-detection-api

Image URI

934857846971.dkr.ecr.us-east-2.amazonaws.com

Essential container

Yes

Private registry

[Info](#)

Store credentials in Secrets Manager, and then use the credentials to reference images in private registries.

☒ Private registry authentication

Port mappings

[Info](#)

Add port mappings to allow the container to access ports on the host to send or receive traffic. Any changes to port mappings configuration impacts the associated service connect settings.

Container port

5000

Protocol

TCP

Port name

gun-detection-api-5000-tcp

App protocol

HTTP

Remove

**Figure 14 Création d'une tâche : Configuration du conteneur**

Ensuite, on définit la façon dont on va héberger le cluster. Ici, on utilise Fargate. Comme on l'a vu dans le choix des technologies, Fargate automatise la gestion des ressources contrairement à EC2 qui nous laisse manuellement gérer les ressources.

Après avoir spécifié les ressources à allouer pour la tâche qui sera hébergée sur Fargate, on spécifie le « Task execution role » qui autorise ECS à faire des appels vers les API AWS. Cela est utile car ECS a besoin d'accéder au conteneur hébergé sur ECR grâce à un pull.

## ▼ Environment

Specify the infrastructure requirements for the task definition.

### App environment [Info](#)

Specify the infrastructure for the task definition.

Add an option ▼

AWS Fargate (serverless) ✕

### Operating system/Architecture [Info](#)

Linux/X86\_64 ▼

### Task size [Info](#)

Specify the amount of CPU and memory to reserve for your task.

CPU

.25 vCPU ▼

Memory

2 GB ▼

### ► Container size - optional [Info](#)

### ▼ Task roles, network mode- *conditional*

#### Task role [Info](#)

A task IAM role allows containers in the task to make API requests to AWS services. You can create a task IAM role from the [IAM console](#) [🔗](#).

- ▼

#### Task execution role [Info](#)

A task execution IAM role is used by the container agent to make AWS API requests on your behalf. If you don't already have a task execution IAM role created, we can create one for you.

ecsTaskExecutionRole ▼

**Figure 15** Création d'une tâche : Environnement et ressources

La tâche a été créée avec succès comme le montre l'image suivante :

Amazon Elastic Container Service > Task definitions > gun-detection-task > Revision 1 > Containers

## gun-detection-task:1

Deploy ▼Actions ▼Create new revision ▼

**Overview** [Info](#)

ARN arn:aws:ecs:us-east-2:9348578 46971:task-definition/gun-detection-task:1	Status ✔ <b>ACTIVE</b>	Time created 4/23/2023, 04:59:58 UTC	App environment FARGATE
Task role -	Task execution role <a href="#">ecsTaskExecutionRole</a>	Operating system/Architecture Linux/X86_64	Network mode awsvpc

[Containers](#) | [JSON](#) | [Storage](#) | [Tags](#)

**Task size**

Task CPU .25 vCPU	Task memory 2 GB
----------------------	---------------------

**Containers** [Info](#)

Container na...	Image	Private regis...	Essential	CPU	Memory	GPU
gun-detectio...	9348578469...	-	true	0	-	-

Figure 16 Tâche créée

### *Création du service*

Ensuite on crée un service. Le service nous permet gérer les tâches (démarrage, auto=scaling, etc.).

Nous laisserons les options par défaut pour la partie environnement :

## Environment

AWS Fargate

Existing cluster

Select an existing cluster. To create a new cluster, go to [Clusters](#).

gun-detection-cluster

▼ Compute configuration (advanced)

Compute options | [Info](#)

To ensure task distribution across your compute types, use appropriate compute options.

☒ Capacity provider strategy  
Specify a launch strategy to distribute your tasks across one or more capacity providers.

☐ Launch type  
Launch tasks directly without the use of a capacity provider strategy.

Capacity provider strategy | [Info](#)

Select either your cluster default capacity provider strategy or select the custom option to configure a different strategy.

☐ Use cluster default

☒ Use custom (Advanced)

Capacity provider	Base	Weight
FARGATE ▼	0	1

Add more

Platform version | [Info](#)

Specify the platform version on which to run your service.

LATEST ▼

Figure 17 Création d'un service : Environnement

Dans la section déploiement, on sélectionne la tâche définie précédemment :

### Deployment configuration

Application type [Info](#)

Specify what type of application you want to run.

☒ Service  
Launch a group of tasks handling a long-running computing work that can be stopped and restarted. For example, a web application.

☐ Task  
Launch a standalone task that runs and terminates. For example, a batch job.

Task definition

Select an existing task definition. To create a new task definition, go to [Task definitions](#).

☐ Specify the revision manually  
Manually input the revision instead of choosing from the 100 most recent revisions for the selected task definition family.

Family

gun-detection-task ▼

Revision

1 (LATEST) ▼

Service name

Assign a unique name for this service.

gun-detection-service

Service type [Info](#)

Specify the service type that the service scheduler will follow.

☒ Replica  
Place and maintain a desired number of tasks across your cluster.

☐ Daemon  
Place and maintain one copy of your task on each container instance.

Desired tasks

Specify the number of tasks to launch.

1

► Deployment options

Figure 18 Création d'un service : Configuration du déploiement

Dans la section « Networking » où on va spécifier un groupe de sécurité « gun-detection-security-group ». Ce groupe contient des règles de sécurité personnalisées pour accéder au cluster à partir de n'importe quelle IP.

**▼ Networking**

**VPC** [Info](#)  
Choose the Virtual Private Cloud to use.

vpc-0f3b898fa4df90c02  
default

**Subnets**  
Choose the subnets within the VPC that the task scheduler should consider for placement.

Choose subnets

subnet-06280905c43cb10a7 X  
us-east-2c

subnet-09b97d119372e7501 X  
us-east-2b

subnet-0afa577cfbda08e6f X  
us-east-2a

**Security group** [Info](#)  
Choose an existing security group or create a new security group.

☒ Use an existing security group

☐ Create a new security group

**Security group name**  
Choose an existing security group.

sg-0377fce53b90f6be9 X  
gun-detection-security-group

**Public IP** [Info](#)  
Choose whether to auto-assign a public IP to the task's elastic network interface (ENI).

☒ Turned on

**Figure 19** Création du service : Networking



Ce groupe autorise l'accès au cluster :

Security group rule ID	Type <a href="#">Info</a>	Protocol <a href="#">Info</a>	Port range <a href="#">Info</a>	Source <a href="#">Info</a>	Description - optional <a href="#">Info</a>		
sgr-060e04262d79384e4	All traffic ▼	All	All	Custom ▼	<div><div>Q</div><div>0.0.0.0 /0 ✕</div></div>		<div>De let e</div>

Figure 20 Groupe de sécurité

Après avoir créé le service, on peut voir qu'il a été déployé :

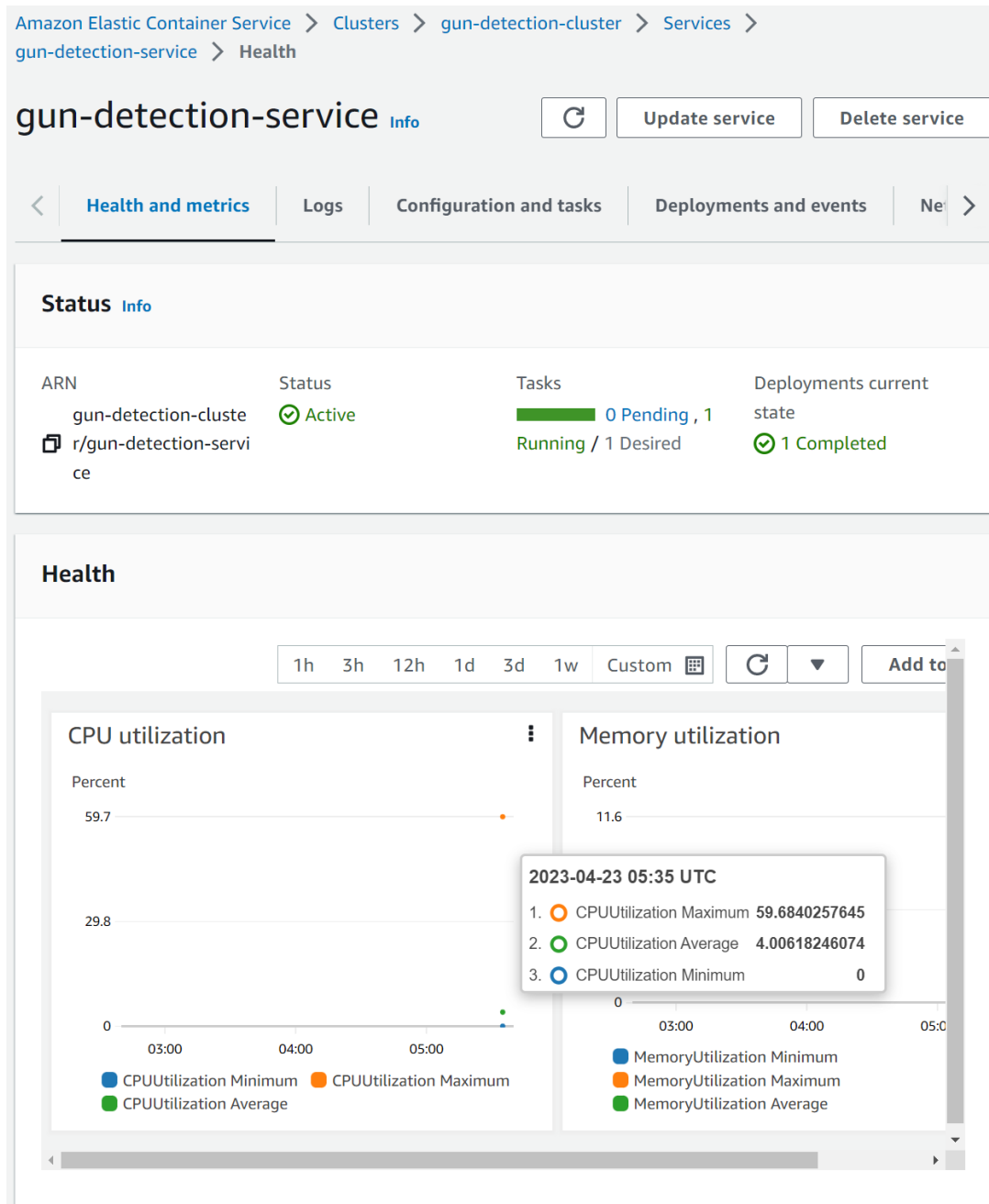


Figure 21 Service créé

L'on peut voir que le déploiement fonctionne parfaitement en accédant à l'URL <http://18.188.109.184:5000/api> :

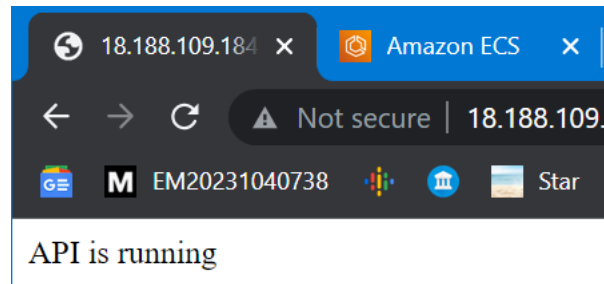


Figure 22 Test de l'API déployée

## Création et déploiement de l'interface

L'interface est simplement un mélange de HTML, CSS et JavaScript. Lorsqu'on télécharge l'image, elle envoie une requête POST à l'API et affiche le résultat.

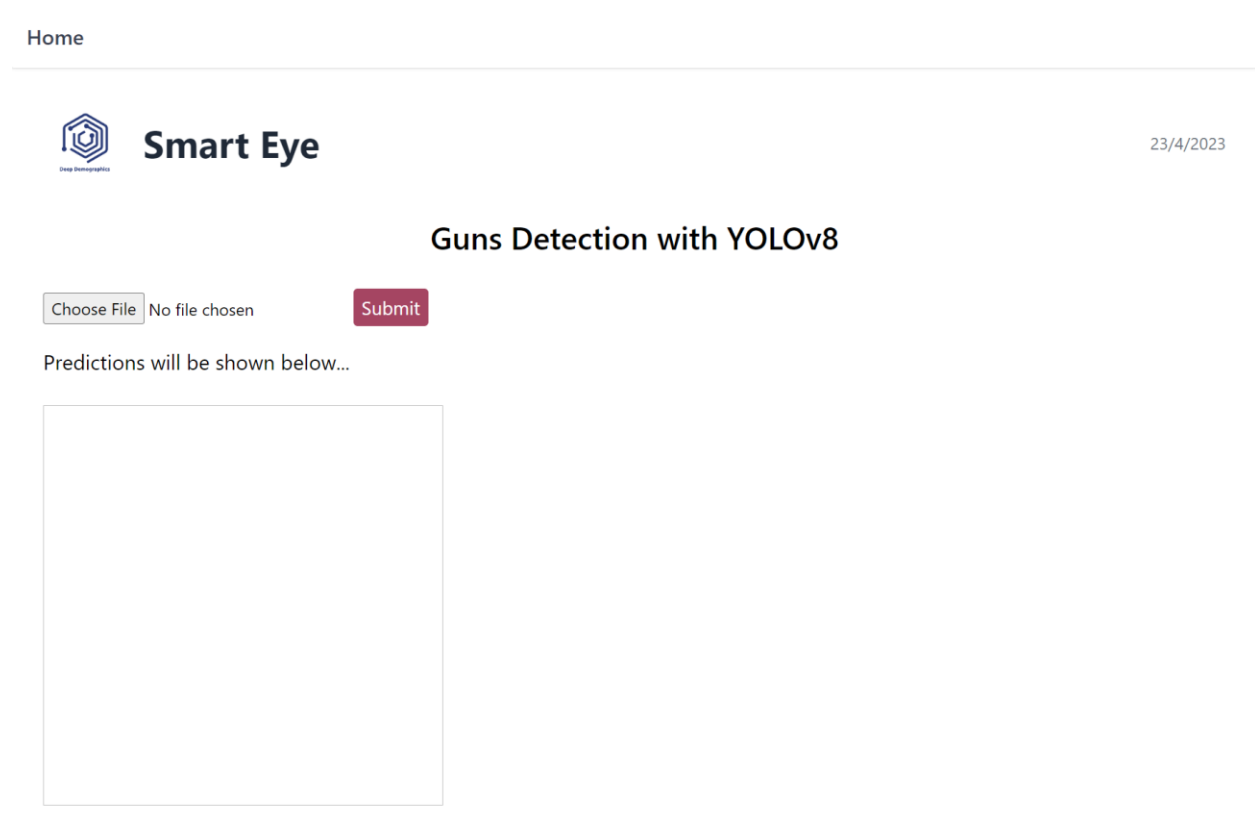


Figure 23 Interface de l'application

Maintenant, il ne reste plus qu'à la déployer sur Elastic Beanstalk. Cela se fait en deux étapes :

- Création d'un environnement Beanstalk.
- Création d'une application Beanstalk

La première fois qu'on tente de créer l'environnement, Elastic Beanstalk crée un bucket automatiquement, nommé « elasticbeanstalk-<region>-id ». Au départ, ce bucket n'est pas accessible publiquement. La création de l'environnement va donc échouer.

Pour régler ce problème, il faut donc modifier les permissions.


### *Modification des permissions*

Il y a trois permissions à mettre à jour :

1. Public Access : l'autorisation de l'accès au bucket de façon publique.
2. Object Ownership : détermine qui peut avoir accès aux objets.
3. Access Control List (ACL) : définit qui peut lire et écrire dans le bucket.

« Block all public access » doit être désactivé afin que l'accès au public soit débloqué :

#### **Block public access (bucket settings)**

Public access is granted to buckets and objects through access control lists (ACLs), bucket policies, access point policies, or all. In order to ensure that public access to all your S3 buckets and objects is blocked, turn on Block all public access. These settings apply only to this bucket and its access points. AWS recommends that you turn on Block all public access, but before applying any of these settings, ensure that your applications will work correctly without public access. If you require some level of public access to your buckets or objects within, you can customize the individual settings below to suit your specific storage use cases. [Learn more](#) 

Edit

#### **Block all public access**

 Off

► Individual Block Public Access settings for this bucket

**Figure 24 Autorisation de l'accès au public**

L'object ownership doit être défini sur « Bucket Owner Preferred ». Afin que le propriétaire du bucket S3 puisse spécifier les règles d'accès.

**Object Ownership** [Info](#)

Control ownership of objects written to this bucket from other AWS accounts and the use of access control lists (ACLs). Object ownership determines who can specify access to objects.

Edit

Object Ownership

Bucket owner preferred


ACLs are enabled and can be used to grant access to this bucket and its objects. If new objects written to this bucket specify the bucket-owner-full-control canned ACL, they are owned by the bucket owner. Otherwise, they are owned by the object writer.


Figure 25 L'object ownership

Une fois que l'object ownership a été modifié, nous pouvons spécifier les ACL (Access Control List). On autorise la lecture (Read) à tout le monde (Everyone) :

**Access control list (ACL)** [Edit](#)

Grant basic read/write permissions to other AWS accounts. [Learn more](#)

 **The console displays combined access grants for duplicate grantees**  
To see the full list of ACLs, use the Amazon S3 REST API, AWS CLI, or AWS SDKs.

 **AWS doesn't recommend granting access to the Everyone grantee**  
Anyone in the world can access the objects in this bucket.  
[Learn more](#)






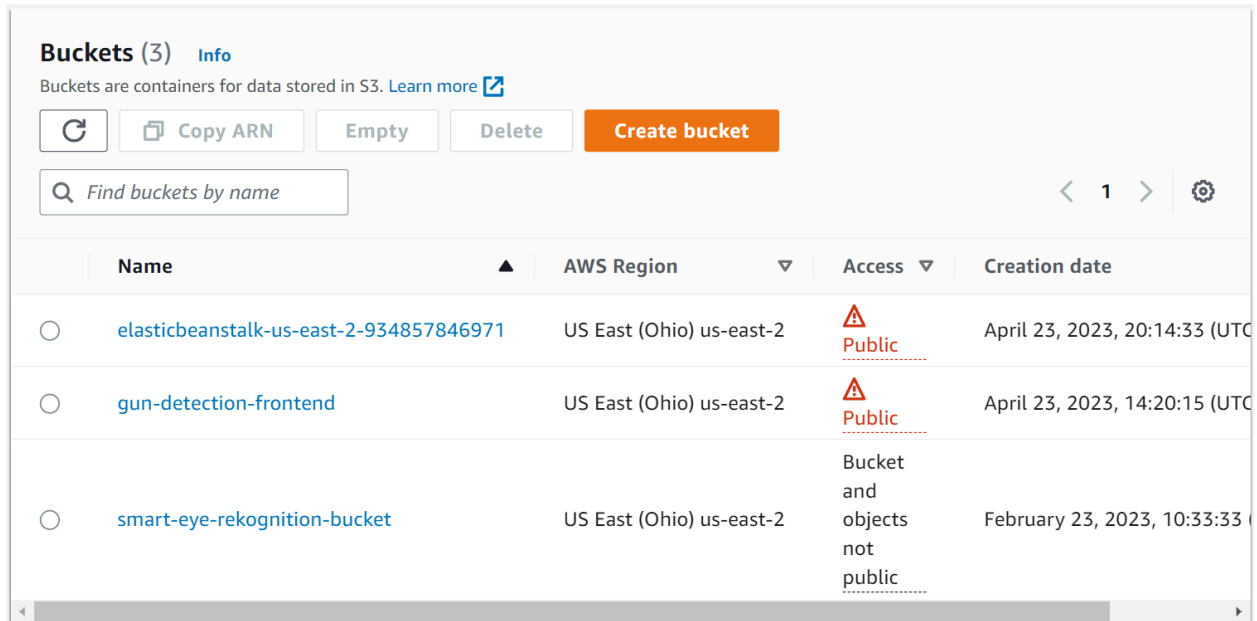
Grantee	Objects	Bucket ACL
Bucket owner (your AWS account) Canonical ID:  7e5842d536d363a4fb27e261ca255cce2ad290810f5c5642dfe424aba477df31	List, Write	Read, Write
Everyone (public access) Group:  <a href="http://acs.amazonaws.com/groups/global/AllUsers">http://acs.amazonaws.com/groups/global/AllUsers</a>	-	 Read
Authenticated users group (anyone with an AWS account) Group:  <a href="http://acs.amazonaws.com/groups/global/AuthenticatedUsers">http://acs.amazonaws.com/groups/global/AuthenticatedUsers</a>	-	-
S3 log delivery group Group:  <a href="http://acs.amazonaws.com/groups/s3/LogDelivery">http://acs.amazonaws.com/groups/s3/LogDelivery</a>	-	-

Figure 26 Modification des ACL pour autoriser la lecture

Au final, notre bucket ressemble à ceci :



**Buckets (3)** [Info](#)

Buckets are containers for data stored in S3. [Learn more](#)

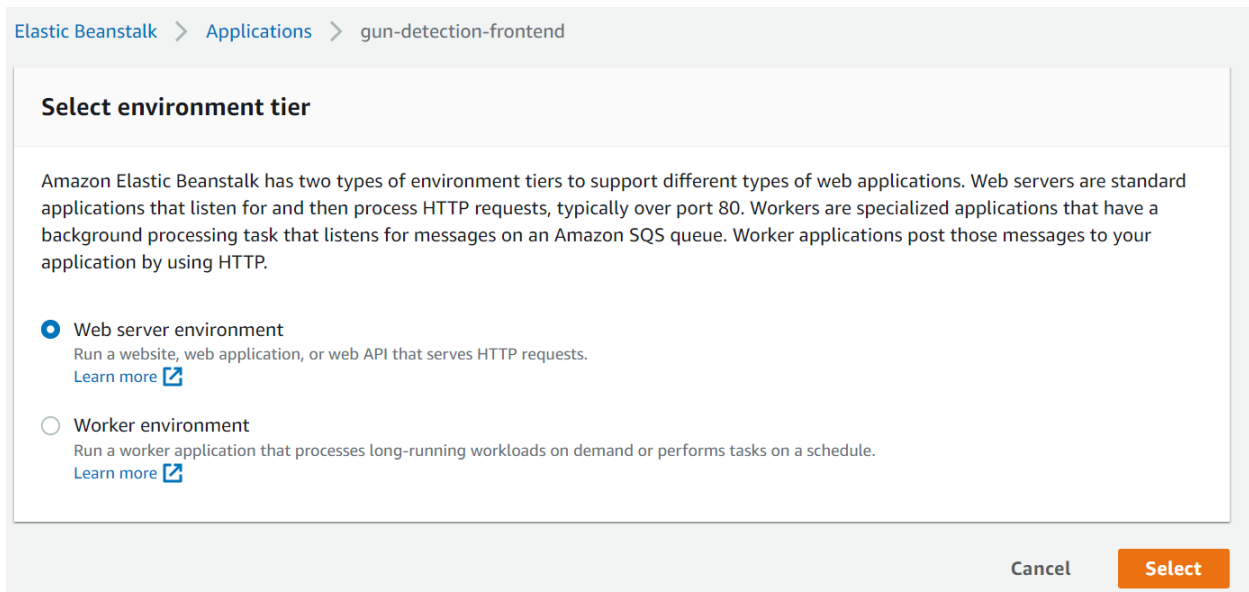
[Refresh](#) [Copy ARN](#) [Empty](#) [Delete](#) [Create bucket](#)

	Name ▲	AWS Region ▼	Access ▼	Creation date
<input type="radio"/>	elasticbeanstalk-us-east-2-934857846971	US East (Ohio) us-east-2	<span>Public</span>	April 23, 2023, 20:14:33 (UTC)
<input type="radio"/>	gun-detection-frontend	US East (Ohio) us-east-2	<span>Public</span>	April 23, 2023, 14:20:15 (UTC)
<input type="radio"/>	smart-eye-rekognition-bucket	US East (Ohio) us-east-2	Bucket and objects not public	February 23, 2023, 10:33:33 (UTC)

Figure 26 Bucket elasticbeanstalk

## Création de l'environnement Beanstalk

Finalement, on crée l'environnement Beanstalk de type serveur web, on spécifie Node.js comme plateforme puisqu'on a une application JavaScript, et on téléverse le code.



Elastic Beanstalk > Applications > gun-detection-frontend

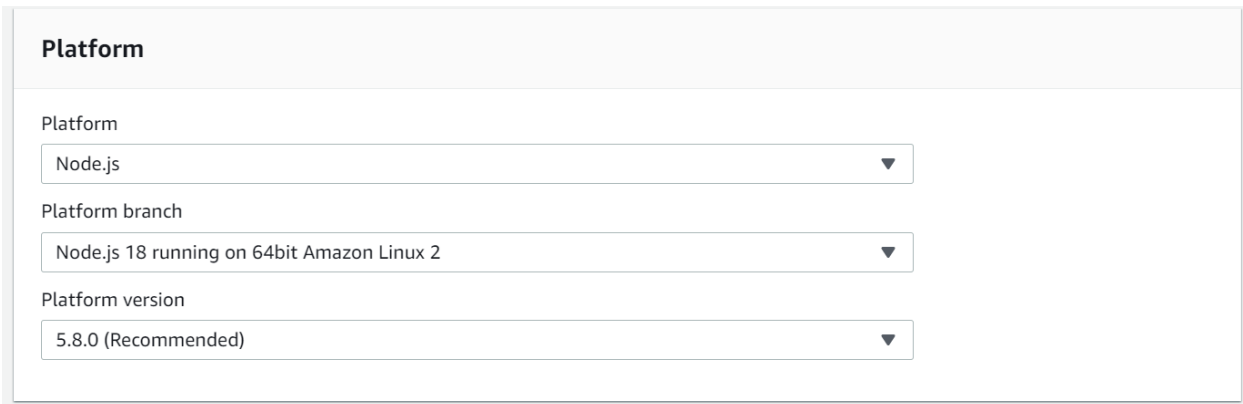
### Select environment tier

Amazon Elastic Beanstalk has two types of environment tiers to support different types of web applications. Web servers are standard applications that listen for and then process HTTP requests, typically over port 80. Workers are specialized applications that have a background processing task that listens for messages on an Amazon SQS queue. Worker applications post those messages to your application by using HTTP.

- ☒ **Web server environment**  
Run a website, web application, or web API that serves HTTP requests.  
[Learn more](#)
- ☐ **Worker environment**  
Run a worker application that processes long-running workloads on demand or performs tasks on a schedule.  
[Learn more](#)

[Cancel](#) [Select](#)

Figure 27 création d'un environnement server web beanstalk



**Platform**

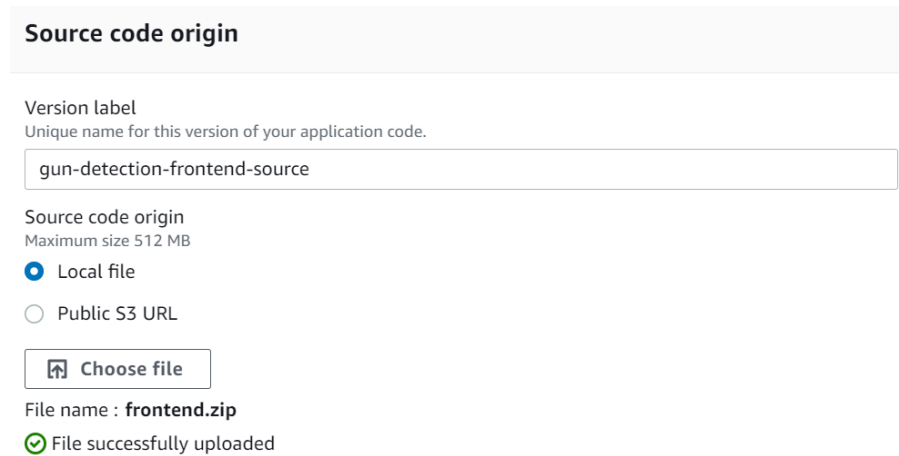
Platform  
Node.js ▼

Platform branch  
Node.js 18 running on 64bit Amazon Linux 2 ▼

Platform version  
5.8.0 (Recommended) ▼

**Figure 28 Création d'un environnement Beanstalk : Choix de la plateforme**

Puis, on téléverse le code :




**Source code origin**

Version label  
Unique name for this version of your application code.  
gun-detection-frontend-source

Source code origin  
Maximum size 512 MB

☒ Local file  
☐ Public S3 URL

 Choose file

File name : **frontend.zip**

✔ File successfully uploaded

**Figure 29 Création d'un environnement Beanstalk : Téléversement du code**

Finalement, l'environnement est créé :



### Creating Gundetectionfrontend-env

This will take a few minutes. ...

```
1:24am Create environment operation is complete, but with errors. For more information, see troubleshooting documentation.
1:24am Added instance [i-0d87e049e750de149] to your environment.
1:23am Command execution completed on all instances. Summary: [Successful: 0, Failed: 1].
1:23am [Instance: i-0d87e049e750de149] Command failed on instance. Return code: 1 Output: Engine execution has encountered an error..
1:23am Instance deployment failed to generate a 'Procfile' for Node.js. Provide one of these files: 'package.json', 'server.js', or 'app.js'. The deployment failed.
1:23am Instance deployment: You didn't include a 'package.json' file in your source bundle. The deployment didn't install a specific Node.js version.
1:23am Instance deployment failed. For details, see 'eb-engine.log'.
1:23am Created Load Balancer listener named:
arn:aws:elasticloadbalancing:us-east-2:934857846971:listener/app/awseb-AWSEB-HTAU6NDQT6SG/bad43238eaddde25/a667cedf7f72254b
1:23am Created load balancer named:
arn:aws:elasticloadbalancing:us-east-2:934857846971:loadbalancer/app/awseb-AWSEB-HTAU6NDQT6SG/bad43238eaddde25
1:22am Created CloudWatch alarm named:
awseb-e-hk4ayisuui-stack-AWSEBCloudwatchAlarmHigh-1RL3YK80TCIRK
1:22am Created CloudWatch alarm named:
awseb-e-hk4ayisuui-stack-AWSEBCloudwatchAlarmLow-8ZE2MHNJ8YPQ
```

Figure 30 Création de l'environnement beanstalk : Fin

## 3.2 Tests

Lorsqu'on lance l'application, et qu'on envoie une image, on peut voir que le modèle détecte une arme.





## Guns Detection with YOLOv8

Choose File 4529570706...f70f6fc\_o.jpg

Submit



## Conclusion

Dans cette partie, nous avons eu un aperçu général de l'implémentation du système. Cependant, afin de garder ce rapport concis, certaines étapes mineurs ont été omises.

Nous avons vu la création du modèle de détection d'armes à feu, son intégration à l'API et sa conteneurisation avec Docker. Puis nous avons déployé le conteneur en utilisant les services d'Amazon notamment ECS qui permet l'orchestration et la gestion des conteneurs.

## CHAPITRE 4 : CONCLUSION

Ainsi s'achève la réalisation du projet « Gun-detection » qui fut un succès.

### Leçons apprises

Qu'aie-je appris dans ce projet ? Énormément de chose. J'ai notamment appris à :

- Servir un modèle d'Intelligence Artificielle (plus précisément de Deep Learning) via une API avec Python
- Déployer une API sous forme de conteneur afin de la rendre accessible au public
- Gérer l'orchestration des conteneurs avec Amazon ECS et Fargate.

Je suis partie d'une simple expérimentation en local pour finir en production en utilisant les technologies les plus avancées qui permettent de gérer des challenges comme l'auto-scaling, l'allocation des ressources, etc.

### Problèmes rencontrés

Évidemment, comme dans la plupart des projets en informatique, j'ai rencontré des challenges à chacune des étapes du projet :

- L'intégration du modèle à l'API : utiliser des algorithmes de Deep Learning pour créer des modèles dans un simple projet Python est une chose, intégrer ces modèles à une API en est une autre. Le format, la taille du modèle, le problème de conversion en d'autres formats sont parmi les challenges que j'ai rencontrés.
- La conteneurisation de l'API : plusieurs problèmes de dépendances se sont posés. Il fallait donc trouver des moyens de spécifier au Dockerfile d'installer ces dépendances manquantes.
- Le déploiement du conteneur : sans doute l'un des plus grands défis de ce projet a été pour de déployer le conteneur sur Amazon ECS. Ce concept m'étant étranger, j'ai dû commettre des erreurs, apprendre à les corriger.

J'ai réussi à dépasser tous ces défis.

### Perspectives

Ce projet a nécessité une grande quantité de travail. Il fallait s'arrêter à un certain point. Cependant, il n'est pas terminé et plusieurs fonctionnalités peuvent être implémentées :

- La prise en charge de vidéos. C'est d'ailleurs le principal objectif : détecter des armes en temps à travers des vidéos ou caméras de surveillance.
- L'amélioration de l'interface

# CONCLUSION

Dans ce projet, nous avons développé et déployé une application complète de bout en bout pour la détection d'objets en utilisant un modèle YOLO.

Nous avons commencé par entraîner le modèle sur un ensemble de données personnalisé, puis nous avons déployé une API Flask pour servir les prédictions. Nous avons également créé une interface utilisateur conviviale en utilisant JavaScript, qui envoie des demandes à l'API et affiche les prédictions sur une image. Nous avons utilisé Docker pour conteneuriser l'API et l'interface utilisateur, puis nous les avons déployés sur AWS Fargate. Nous avons également configuré un équilibrage de charge et une surveillance des performances à l'aide d'Amazon ECS.

En fin de compte, nous avons créé une application de détection d'objets entièrement fonctionnelle, qui peut être utilisée pour de nombreuses applications, telles que la surveillance vidéo, la sécurité et la reconnaissance d'objets.

Ce projet pour moi une des expériences les plus enrichissantes car il regroupe une très grande variété d'outils, de concepts et d'architectures.

# APPENDICE

Le code de ce projet est disponible sur GitHub : <https://github.com/BecayeSoft/Guns-Detections-Project>. Ce repository contient trois dossiers :

- API : contient l'API
- frontend : contient l'application web
- desktop-app : contient une application de bureau en Python pour tester le modèle

L'API hébergée dans le cloud est restreinte à mon usage personnel pour des raisons de coût. Cependant il est possible de cloner l'API, de lancer pour tester le modèle local. C'est pareil pour le front-end.

Un fichier README sera inclus dans le repository GitHub dans chaque sous-dossier afin de détailler un peu plus l'utilisation du code.

# WEBOGRAPHIE

- <https://aws.amazon.com>
- <https://aws.amazon.com/fargate>
- <https://aws.amazon.com/ecr>
- <https://aws.amazon.com/ecs>
- <https://aws.amazon.com/elasticbeanstalk>
- <https://flask-cors.readthedocs.io/en/latest/>
- <https://flask.palletsprojects.com/en/2.3.x/>
- <https://github.com/ultralytics/ultralytics/tree/main/docs>
- <https://www.youtube.com/watch?v=MJ1vWb1rGwM&t=8986s&pp=ygUjZW5klHRvIGVuZCBtYWNoaW5lIGxlyXJuaW5nIHByb2pY3Q%3D>
- <https://stackoverflow.com/questions/71080354/getting-the-bucket-does-not-allow-acls-error>
- <https://roboflow.com/>