.ipssi

RAPPORT DE PROJET FINAL

INGÉNIERIE DEVOPS & CLOUD COMPUTING

Déploiement Automatisé d'une Application Node.js

(CI/CD Professionnel - Projet 2048)

Auteurs :
Mohamed KOUBAA
Ala BOUSSARSAR

Technologies :

`AWS (EC2, ECR)`

`Ansible (Provisioning)`

`Jenkins (Pipeline CI/CD)`

`Docker (Conteneurisation)`

Dépôt GitHub :
https://github.com/Akatsuki1995/2048-devops-project

Date du rendu : 30 Novembre 2025

Année universitaire : 2025 / 2026

Table des matières

1 Introduction Générale

1.1 Contexte et Objectifs

Résumé Exécutif

Ce projet a pour objectif la mise en œuvre complète d'une chaîne DevOps moderne, depuis le code source jusqu'au déploiement en production sur Amazon Web Services (AWS). Il répond à une problématique d'entreprise : comment transformer une application legacy (le jeu 2048) en un service cloud natif, résilient et mis à jour automatiquement ?

Le travail couvre l'intégralité du cycle :

- **Conteneurisation** de l'application Node.js.
- **Infrastructure as Code (IaC)** avec Ansible pour provisionner l'environnement AWS.
- **Pipeline CI/CD** avec Jenkins pour l'automatisation du build et du déploiement.
- **Observabilité** via des webhooks GitHub pour le déclenchement automatique.

1.2 Organisation et Travail en Groupe

Ce projet a été réalisé en binôme. La répartition des tâches a été effectuée comme suit pour simuler un environnement professionnel :

- **Mohamed KOUBAA :** Responsable de la partie CI/CD (Jenkins), de la gestion des secrets et de l'intégration Webhook.
- **Ala BOUSSARSAR :** Responsable de l'Infrastructure as Code (Ansible), de la configuration AWS (EC2/ECR) et de la conteneurisation Docker.

La collaboration s'est faite via un dépôt GitHub commun (lien), avec des revues de code systématiques.

2 Prérequis et Environnement Technique

2.1 Prérequis Logiciels

Le projet s'appuie sur la stack technique suivante, installée et configurée localement et sur le serveur CI :

Outil	**Rôle dans le projet**
Git & GitHub	Gestion du code source (SCM) et déclencheur Webhook
AWS CLI	Interaction en ligne de commande avec l'API AWS
Docker	Construction de l'image de l'application (Build)
Node.js ($\geq$ 16)	Runtime pour l'application serveur Express
Ansible	Automatisation du provisioning de l'infrastructure
Jenkins	Orchestrateur du Pipeline CI/CD

TABLE 1 – Tableau des outils utilisés

2.2 Architecture Cible

L'architecture déployée respecte le schéma suivant : Flux directionnel du développeur vers la production.

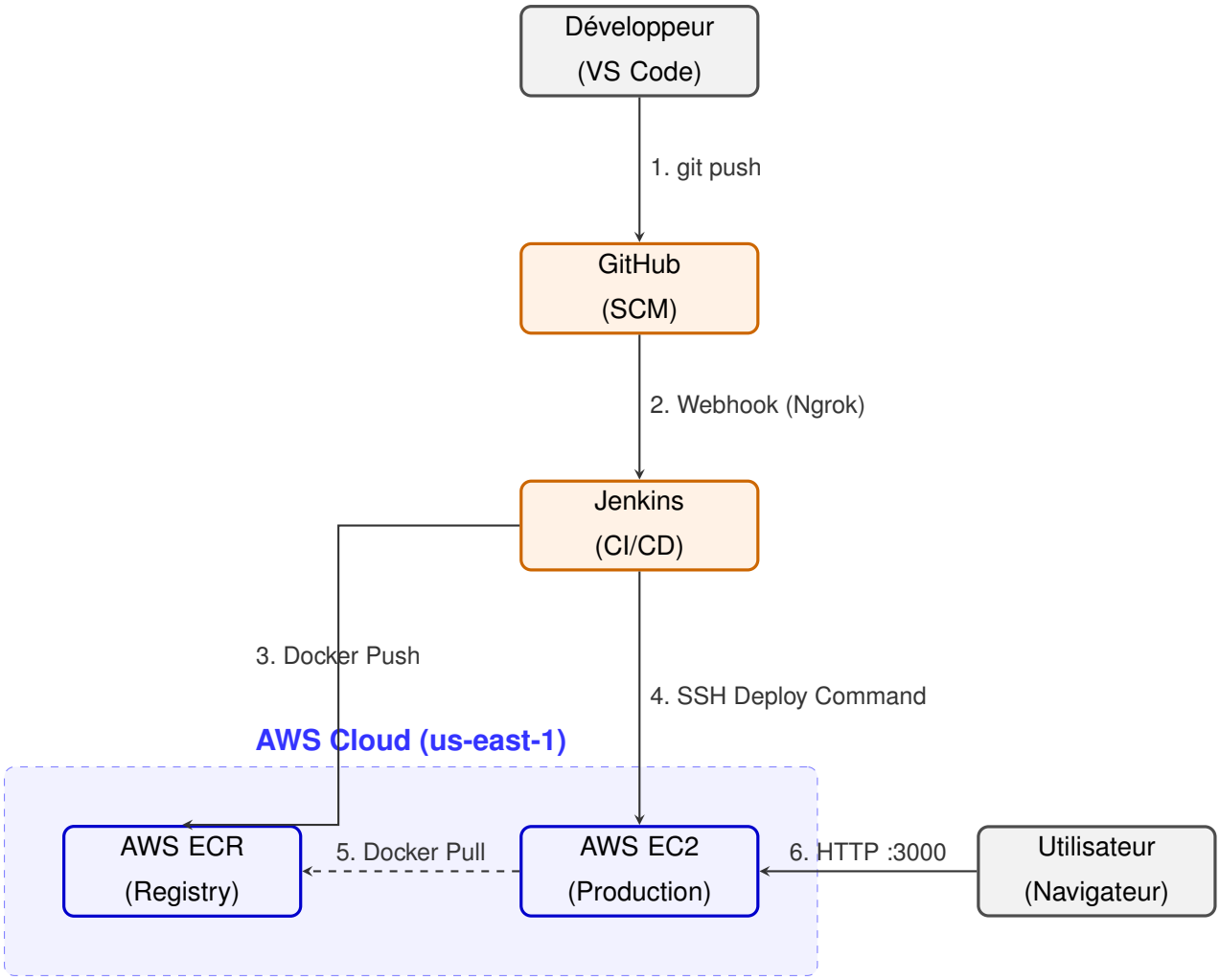

FIGURE 1 – Architecture du Pipeline CI/CD

3 Préparation de l'Application (Node.js)

3.1 Choix et Fork de l'Application

Nous avons sélectionné le projet open-source **2048**, disponible sur GitHub. Conformément aux consignes, nous avons "Forké" le projet dans notre espace personnel pour en avoir le contrôle total.

3.2 Transformation en Application Node.js

L'application d'origine étant statique, nous avons créé un serveur web léger avec **Express.js** pour la servir.

Fichier `server.js` créé :

```
GNU nano 8.4                                                server.js
const express = require('express');
const path = require('path');
const app = express();
const PORT = 3000;

// Serve static files (css, js, images)
app.use(express.static(path.join(__dirname, '/')));

// Serve the game
app.get('/', (req, res) => {
    res.sendFile(path.join(__dirname, 'index.html'));
});

app.listen(PORT, () => {
    console.log(`2048 App running on port ${PORT}`);
});
```

FIGURE 2 – Code du serveur Node.js (Express)

3.3 Conteneurisation (Dockerfile)

Le `Dockerfile` a été conçu pour être léger et sécurisé, basé sur l'image `node:16-alpine`. Il installe les dépendances et expose le port 3000.

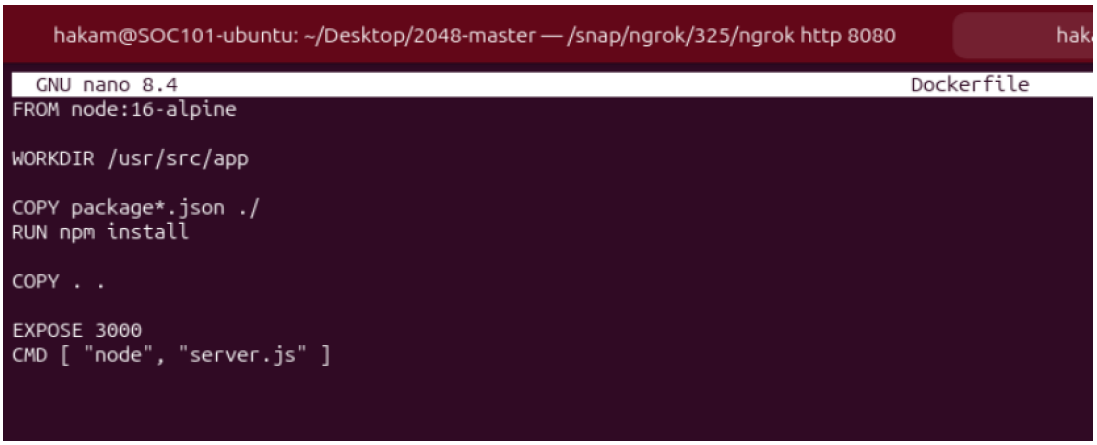

FIGURE 3 – Dockerfile Optimisé pour la production

4 Provisionnement AWS avec Ansible

Le provisionnement de l'infrastructure a été entièrement automatisé via Ansible, garantissant un environnement reproductible.

4.1 Configuration d'Ansible

Nous avons utilisé les collections `community.aws` et `amazon.aws`.

— Installation : `ansible-galaxy collection install community.aws amazon.aws`

— Configuration : Un fichier `ansible.cfg` a été créé pour définir l'inventaire local.

4.2 Structure du Playbook

Conformément aux bonnes pratiques exigées, nous avons structuré le projet Ansible de manière modulaire :

```
ansible/
  playbook.yml       # Point d'entrée
  roles/             # Rôles (common, aws_infra)
  group_vars/        # Variables (région, ami, type instance)
```

4.3 Exécution et Résultats

Le playbook exécute les tâches suivantes :

1. Création du dépôt **ECR** (`2048-game-repo`).
2. Création du **Security Group** (`2048-sg`) ouvrant les ports 22 (SSH) et 3000 (App).
3. Lancement de l'instance **EC2** Ubuntu.

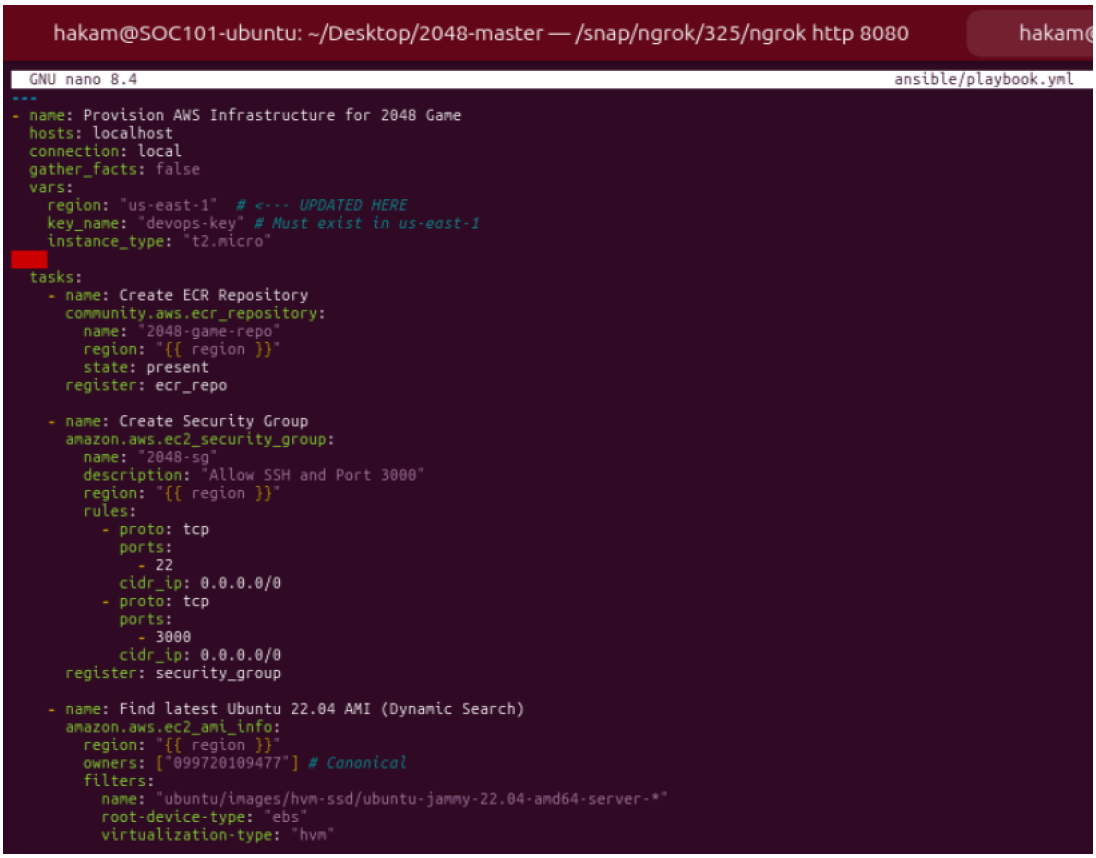

FIGURE 4 – Extrait du Playbook Ansible - Tâches AWS

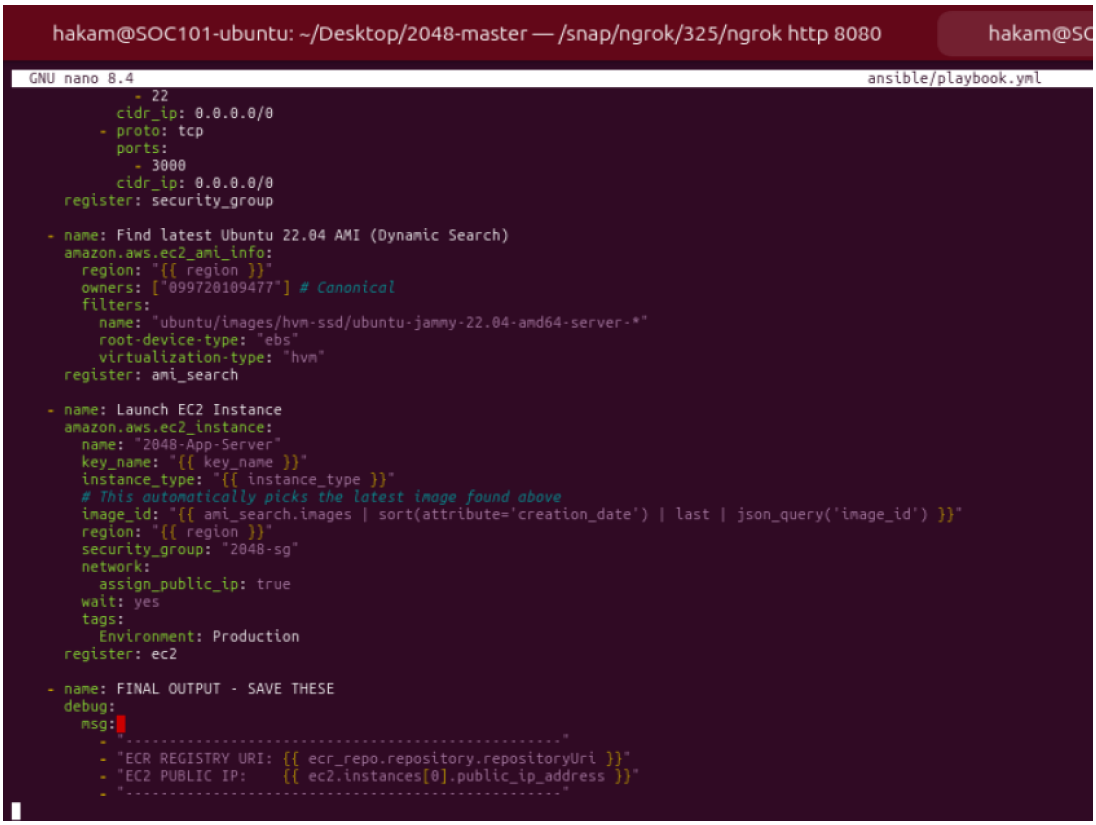

FIGURE 5 – Suite du Playbook - Gestion des instances

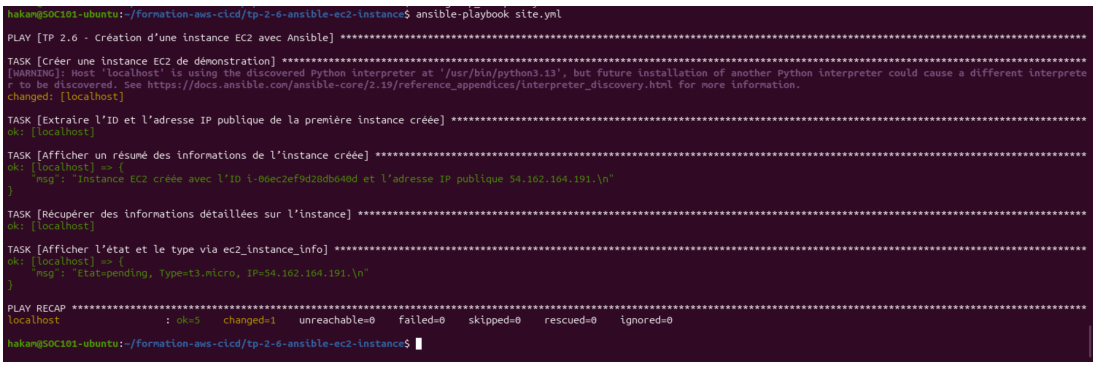

FIGURE 6 – Succès du Provisioning Ansible (Sortie Terminal)

5 Conception du Pipeline Jenkins (CI/CD)

Le pipeline Jenkins automatise le cycle de vie de l'application.

5.1 Préparation de Jenkins

Les plugins suivants ont été installés et configurés :

— **Docker Pipeline :** Pour construire et pousser les images.

— **AWS Credentials :** Pour gérer l'authentification cloud.

— **SSH Agent :** Pour se connecter à l'instance de déploiement.

5.2 Gestion des Secrets

Conformément à la section 10 du cahier des charges ("Gestion des Secrets"), aucun mot de passe n'est stocké dans le code. Nous utilisons le **Jenkins Credentials Store** :

— `aws-standard` : Clés d'accès AWS (Access Key ID / Secret Key).

— `aws-token` : Token de session (Requis pour les comptes AWS Academy).

— `ec2-ssh-key` : Clé privée SSH pour la connexion à l'instance EC2.

5.3 Étapes du Pipeline (Jenkinsfile)

Le pipeline, défini dans le `Jenkinsfile`, suit les 6 étapes recommandées :

1. **Récupération du code :** `checkout scm`.
2. **Installation & Tests :** `npm install` et `npm test`.
3. **Construction Docker :** `docker build`.
4. **Authentification ECR :** Via `aws ecr get-login-password`.
5. **Push vers ECR :** Envoi de l'image taguée.
6. **Déploiement (CD) :** Connexion SSH à l'EC2, arrêt du conteneur actuel, pull de la nouvelle image et redémarrage.

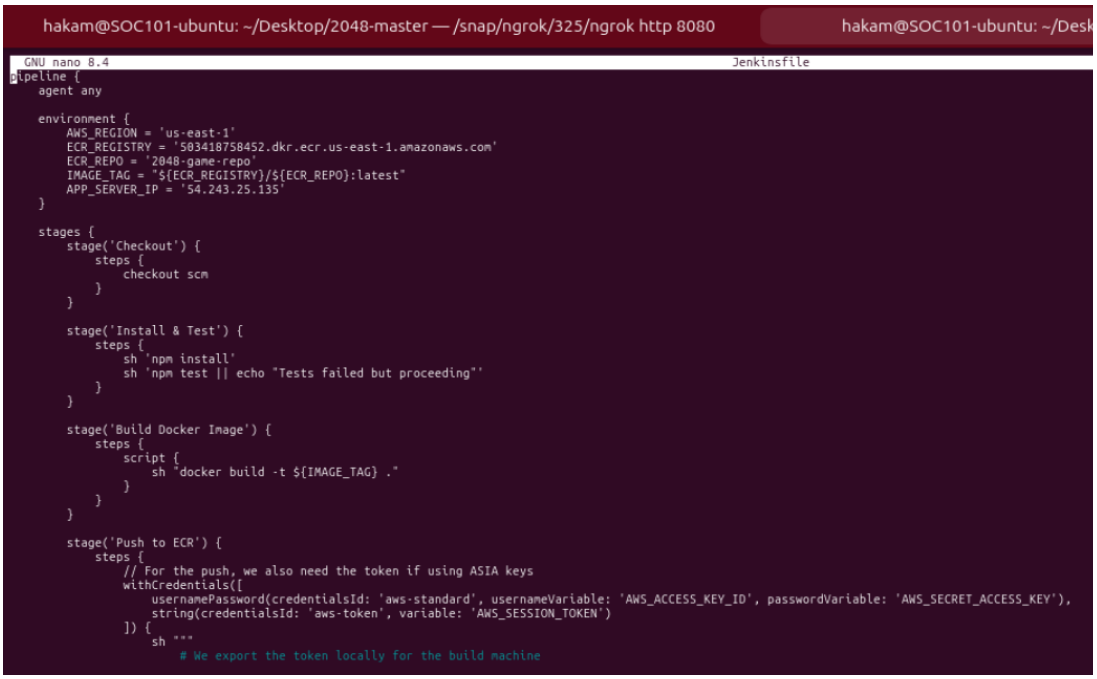

FIGURE 7 – Code du Jenkinsfile - Build et Push

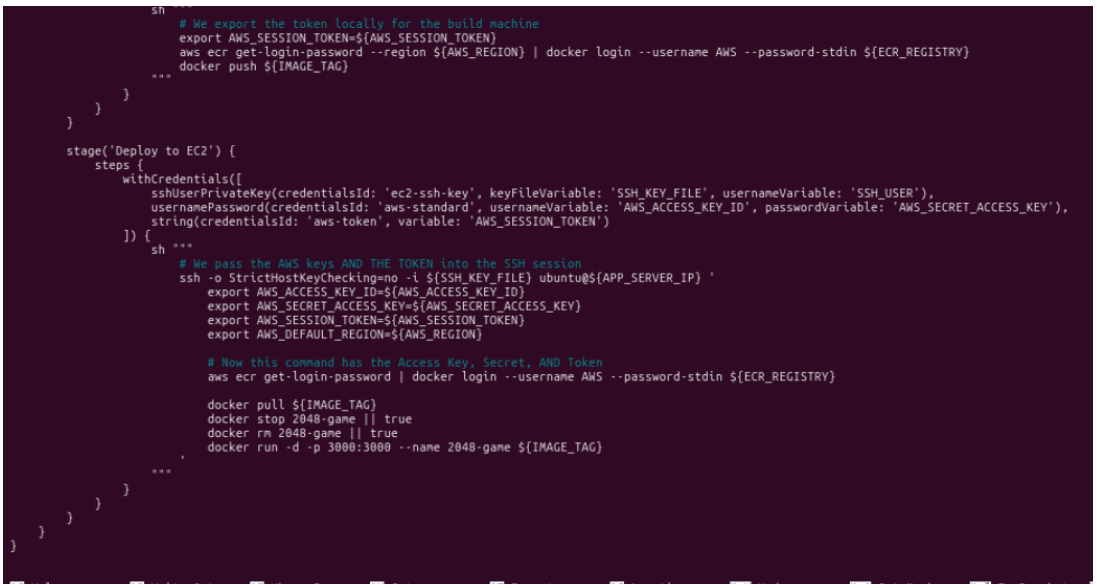

FIGURE 8 – Code du Jenkinsfile - Déploiement SSH

5.4 Configuration du Webhook GitHub

Pour permettre le déclenchement automatique ("Just the push event"), nous avons configuré un Webhook sur GitHub pointant vers notre Jenkins (exposé via Ngrok pour ce projet).

— **Payload URL :** `https://xxxx.ngrok-free.app/github-webhook/`

— **Content type :** `application/json`

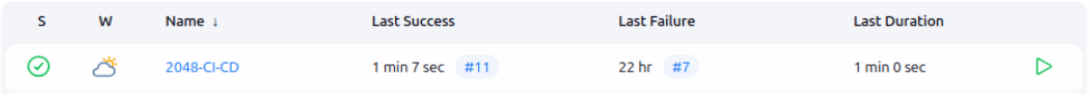

FIGURE 9 – Vue globale du Pipeline Jenkins (Exécution réussie)

6 Vérifications et Validation du Travail

Pour valider le fonctionnement complet de la chaîne, nous avons réalisé un scénario de mise à jour en direct.

6.1 Déploiement Initial

L'application est déployée dans son état d'origine (Fond Beige). L'accès via l'IP publique de l'EC2 sur le port 3000 est fonctionnel.

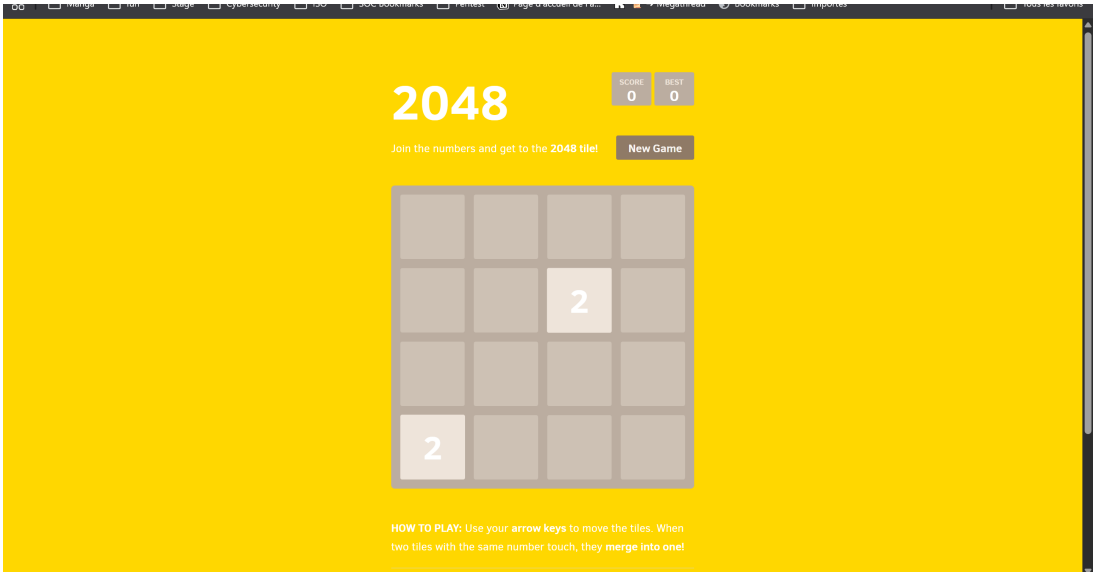

FIGURE 10 – Application avant mise à jour (Version Beige)

6.2 Publication sur ECR

L'image Docker a été correctement construite, taguée et poussée sur le registre privé ECR.

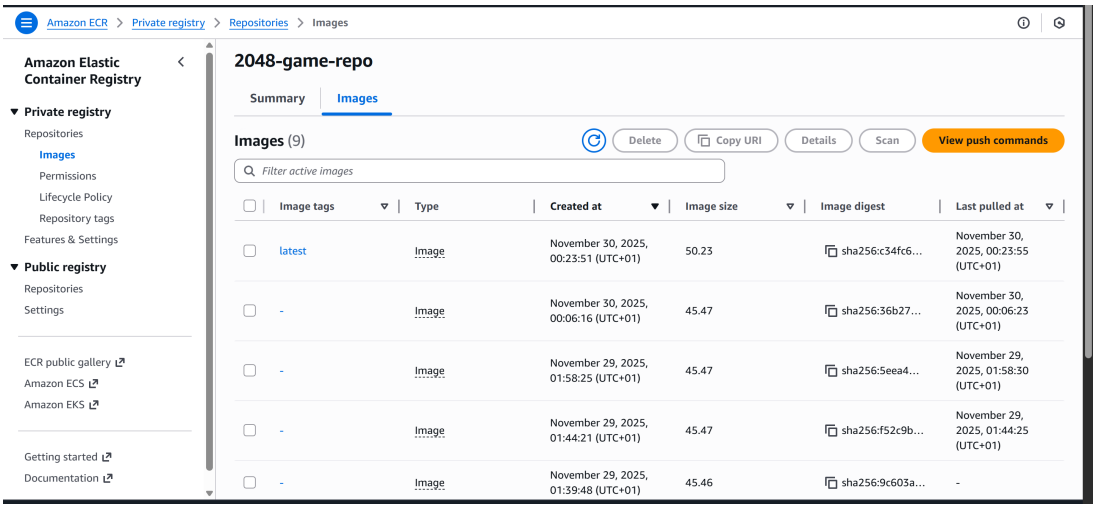

FIGURE 11 – Validation de l'image Docker sur Amazon ECR

6.3 Déploiement via Commit (Le Test "Blue/Gold")

Nous avons modifié le fichier CSS pour changer la couleur de fond en **Bleu Nuit**.

— **Action :** `git commit -m "Update background"` puis `git push`.

— **Réaction :** Jenkins a détecté le push, lancé le build, et mis à jour le conteneur sur l'EC2.

— **Résultat :** Sans intervention manuelle sur le serveur, l'application a changé de couleur.

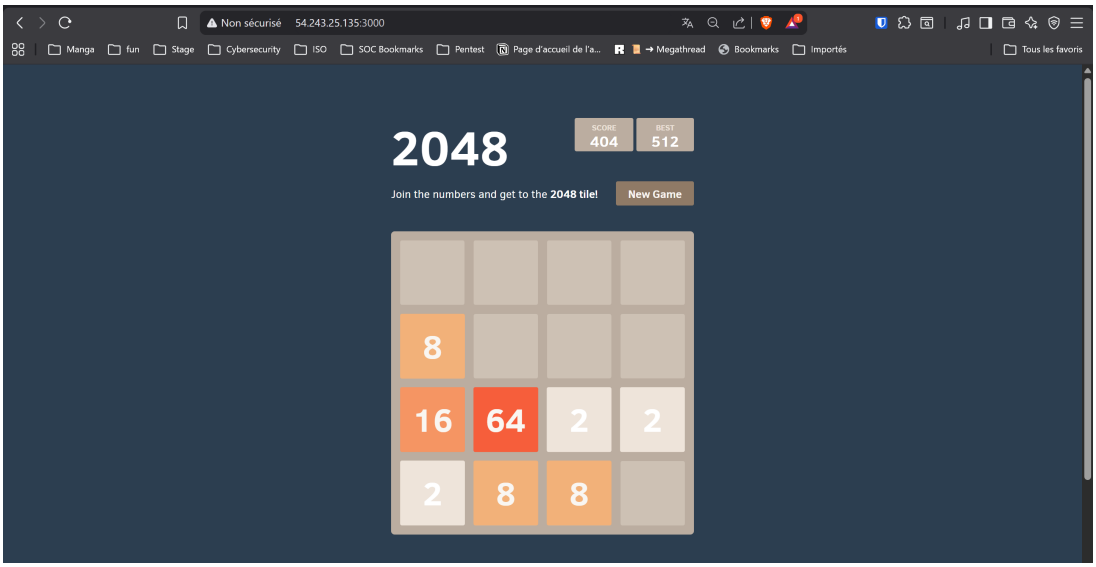

FIGURE 12 – Application mise à jour automatiquement (Version Bleu Nuit)

7 Analyse des Problèmes Rencontrés

7.1 1. Conflit Ansible / Python

Problème Rencontré & Résolution

Problème : Ansible ne parvenait pas à charger la collection `community.aws` en raison d'un conflit de version Python sur l'environnement local.

Solution : Nous avons forcé l'installation locale des collections et utilisé un environnement virtuel Python dédié pour isoler les dépendances `boto3`.

7.2 2. Identifiants AWS Temporaires (ASIA)

Problème Rencontré & Résolution

Problème : Le `docker push` échouait car les comptes AWS Academy utilisent des clés temporaires qui nécessitent un `session_token`.

Solution : Nous avons ajouté une étape dans le Jenkinsfile pour injecter explicitement la variable `AWS_SESSION_TOKEN` lors des commandes AWS CLI.

8 Conclusion

Ce projet final a permis de mettre en pratique l'ensemble des concepts DevOps vus en cours. Nous avons réussi à construire une infrastructure cloud reproductible et un pipeline de déploiement robuste.

Les compétences acquises sont directement applicables en entreprise :

— Maîtrise de l'automatisation via **Ansible**.

— Gestion professionnelle des pipelines **Jenkins**.

— Compréhension profonde de l'écosystème **AWS** et de la conteneurisation.

Le projet est disponible en open-source sur GitHub :
https://github.com/Akatsuki1995/2048-devops-project

Le projet est fonctionnel, documenté et prêt pour une évolution future vers des architectures plus complexes (Kubernetes, Terraform).