Système MLOps de Détection de Fraude Bancaire - Rapport de Restitution

Identifiant du Groupe :

G2-MG04

1. Informations du Groupe

Information	Détail
Identifiant du groupe	G2-MG04
Membre(s)	• Boukasba Fares • BERAMI EL IDRISSI Abdelmoughit • JEBBARI Fatima • GUERMOUCH Kenza • PASCOTO Idriss
Titre du projet	Système MLOps de Détection de Fraude Bancaire en Production
Région de déploiement	eu-west-3 (Paris)
Compte AWS	073184925698

2. Description du Projet

Objectif Principal

Développer et déployer un **système MLOps complet et industriel de détection de fraude bancaire**, capable de :

-Entraîner automatiquement des modèles ML multi-algorithmes
-Exposer des prédictions via une **API REST haute performance**
-Déployer automatiquement sur AWS via un pipeline CI/CD robuste
-Assurer **reproductibilité complète** de l'infrastructure
-Monitorer et logger l'activité en production

Problématique Métier

La fraude bancaire représente un enjeu majeur :

- **Détection rapide** : les transactions frauduleuses doivent être identifiées en temps réel
- **Scalabilité** : volume de millions de transactions/jour
- **Fiabilité** : zéro downtime dans l'infrastructure

- **Transparence** : modèles auditables et traçables

3. Stack Technique Complet

Machine Learning
- **Framework** : Scikit-learn, XGBoost
- **Modèles** : Isolation Forest + XGBoost Ensemble (Voting)
- **Features** : 28 composants PCA + 7 features engineered
- **Performance** : F1-score optimisé sur seuil adaptatif

Backend & API
- **Framework** : FastAPI 0.104.1
- **Python** : 3.11 (officiellement supporté)
- **Serveur** : Uvicorn ASGI
- **Validation** : Pydantic schemas

Frontend
- **Technologie** : HTML5 + Vanilla JavaScript
- **Design** : Responsive, Gradient UI moderne
- **Fonctionnalités** : Single prediction, Batch prediction, Real-time API calls

Conteneurisation
- **Docker** : Multi-stage builds optimisés
- **Base image** : python:3.11-slim (23MB)
- **Health checks** : Intégrés

CI/CD
- **Plateforme** : GitHub Actions
- **Workflows** : 3 pipelines automatisés
- **Tests** : Python 3.9, 3.10, 3.11
- **Auth** : AWS OIDC (zero long-lived secrets)

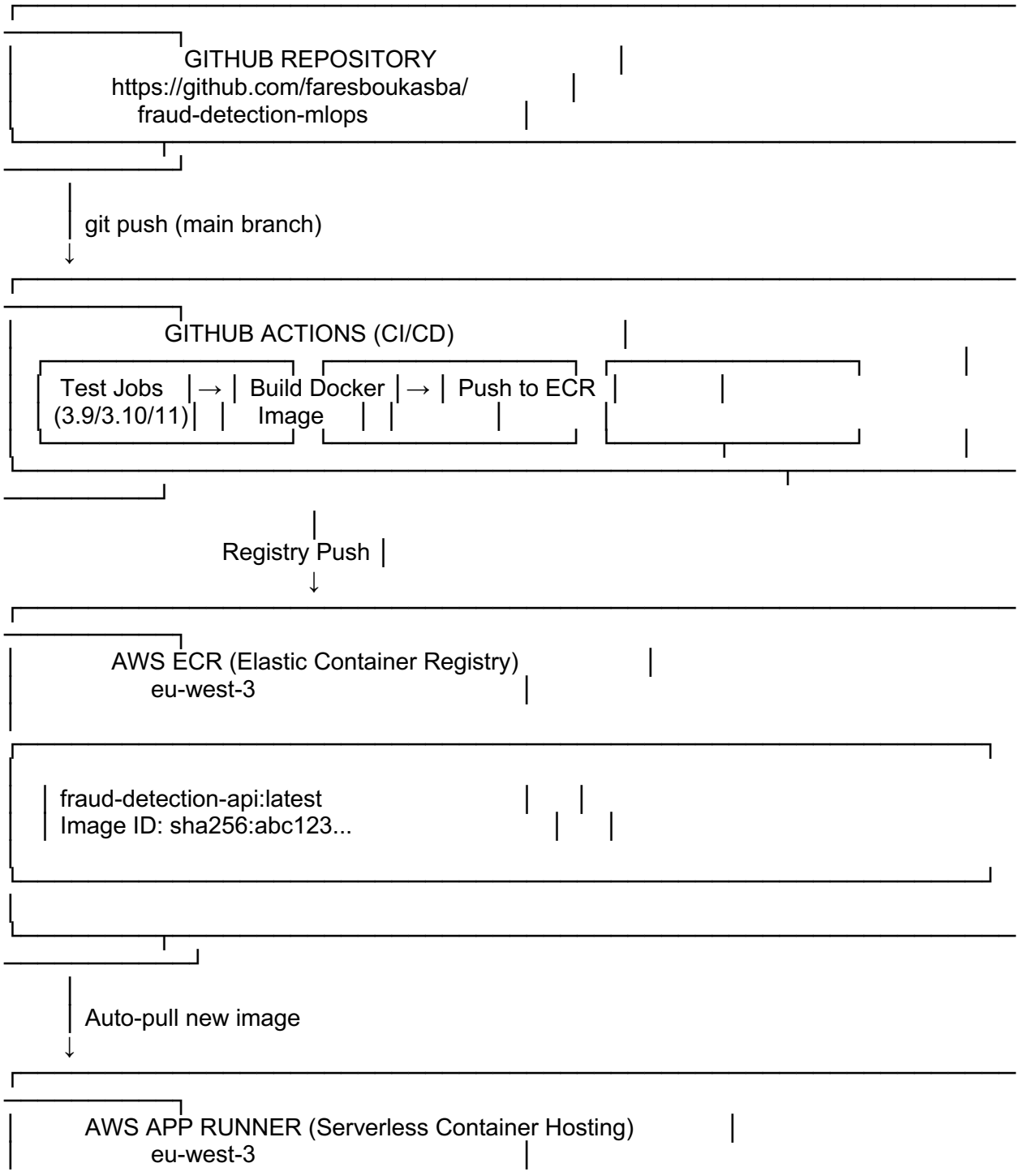
Cloud AWS
- **Conteneurs** : ECR (Elastic Container Registry)
- **Compute** : App Runner (serverless, auto-scaling)
- **Logs & Monitoring** : CloudWatch Logs + Metrics
- **IAM** : Rôles OIDC sécurisés
- **Région** : eu-west-3 (Paris)

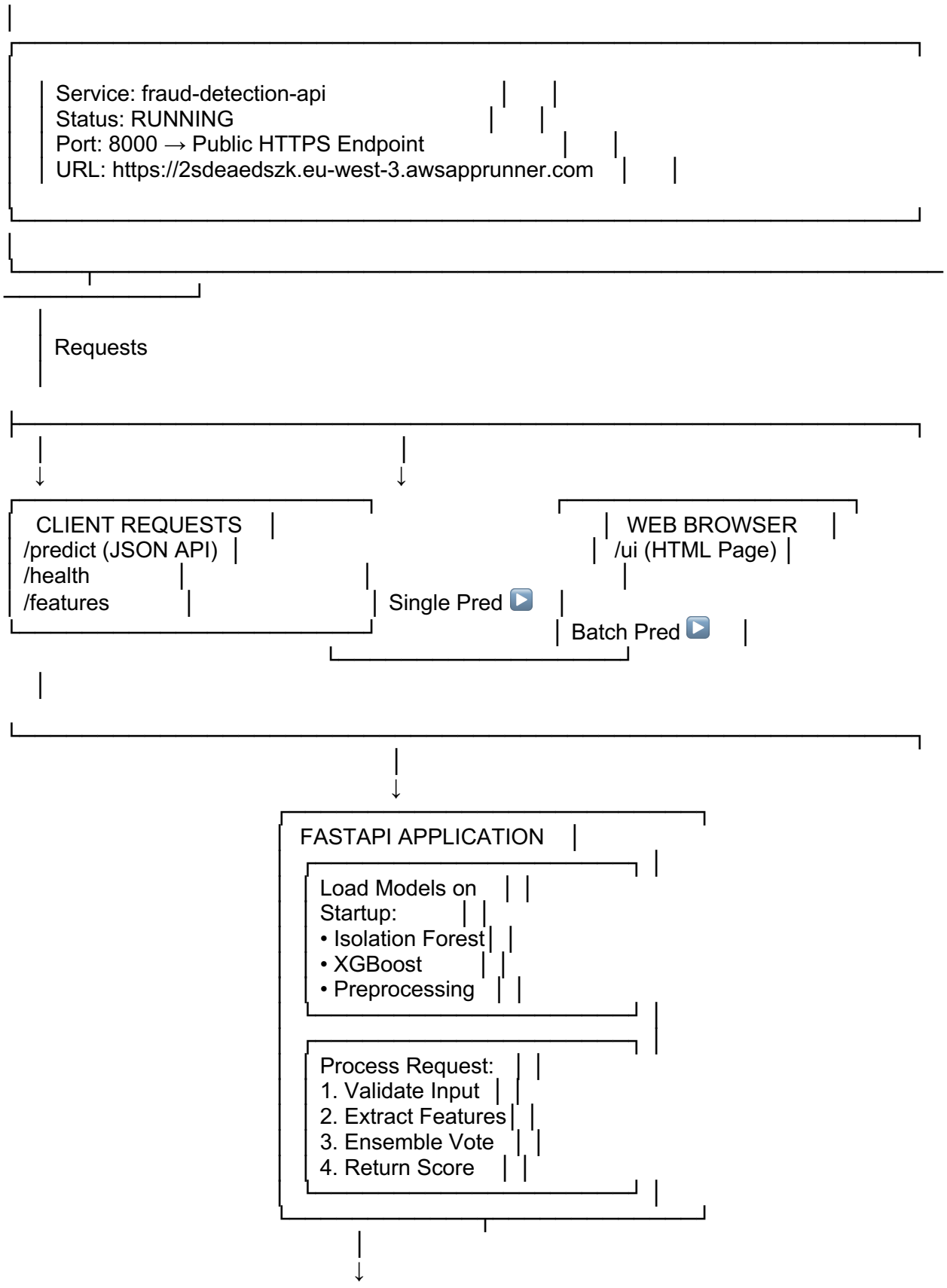
Infrastructure as Code
- **IaC** : Terraform 1.x
- **Ressources** : ECR repos, App Runner services, IAM roles, CloudWatch alarms
- **Reproductibilité** : 100% déclarative

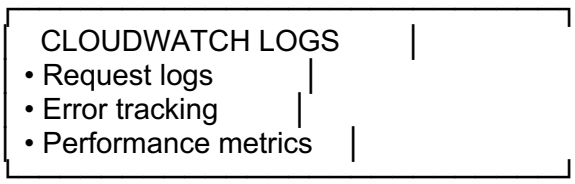
Objectif de la section : Démontrer la **compréhension complète du stack MLOps moderne**.

4. Architecture Globale du Système

Schéma d'Architecture




Service: fraud-detection-api
Status: RUNNING
Port: 8000 → Public HTTPS Endpoint
URL: https://2sdeaedszk.eu-west-3.awsapprunner.com
Requests
CLIENT REQUESTS
/predict (JSON API)
/health
/features
WEB BROWSER
/ui (HTML Page)
Single Pred ▶️
Batch Pred ▶️
FASTAPI APPLICATION
Load Models on Startup:
• Isolation Forest
• XGBoost
• Preprocessing
Process Request:
1. Validate Input
2. Extract Features
3. Ensemble Vote
4. Return Score



5. Description des Composants

1. Versioning & CI/CD

Le code source est hébergé sur **GitHub** dans le repository fraud-detection-mlops. À chaque git push sur la branche main :

1. **GitHub Actions** déclenche le workflow CI/CD automatiquement
2. **Tests unitaires** s'exécutent en parallèle pour Python 3.9, 3.10, et 3.11
3. **Lint & Code Quality** sont vérifiés (flake8, pytest)
4. **Docker build** crée une image optimisée multi-stage
5. **AWS ECR** reçoit l'image pushée avec tags latest et sha256

2. Infrastructure Cloud (AWS)

L'application est déployée entièrement sur **AWS eu-west-3** :

- **ECR** : Stockage centralisé des images Docker versionnées
- **App Runner** : Service serverless qui tire les images ECR et expose l'API publiquement
- **CloudWatch** : Centralisation des logs applicatifs et métriques système
- **IAM** : Rôles avec authentification OIDC (GitHub → AWS)

3. API FastAPI

L'API expose plusieurs endpoints :

- GET /health : Vérification de l'état du service
- POST /predict : Prédiction unitaire avec 35 features requises
- POST /predict-batch : Prédictions en lot
- GET /features : Liste des features requises
- GET /ui : Interface HTML interactive (NOUVEAUTÉ)
- GET /docs : Swagger auto-généré

4. Frontend Web

Interface HTML5 moderne servie directement par FastAPI :

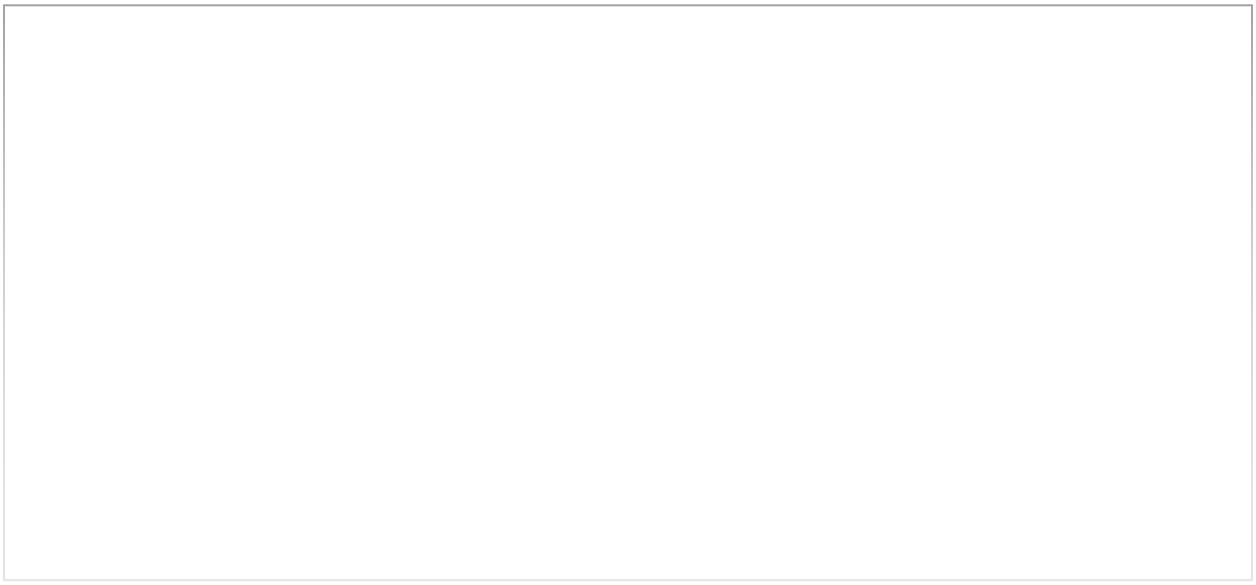
- **Single Prediction Tab** : Formulaire pour analyser une transaction
- **Batch Prediction Tab** : Upload CSV pour prédictions massives
- **Real-time Status** : Indicateur de connexion API
- **Result Display** : Scores de fraude avec seuil et décision

5. ML Pipeline

Modèles entraînés et sérialisés :

- **Isolation Forest** : Détection d'anomalies non supervisée
- **XGBoost** : Classification supervisée haute précision
- **Ensemble Voting** : Combinaison pondérée des deux modèles
- **Feature Engineering** : 7 features calculées à l'inférence

Objectif de la section : Démontrer une **vision MLOps production-grade**.



6. Data Pipeline

Pipeline Complet de Données

Le pipeline de préparation des données est **entièrement automatisé et reproductible** :

```
Kaggle Dataset
    ↓
  Download (download_data.py)
    ↓
  Clean & Transform (clean_transform.py)
    ├─ Handle missing values
    ├─ Remove duplicates
    ├─ Feature engineering
    └─ Normalization/Scaling
    ↓
  Data Pipeline (data_pipeline.py)
    ├─ Train/Test split
    ├─ Feature standardization
```

└─ Pickle serialization
↓
models/ directory
└─ Saved datasets + scalers

Structure des Fichiers

```
src/data/
├── download_data.py        # Kaggle API integration
├── clean_transform.py      # Data cleaning & feature engineering
├── data_pipeline.py        # Full preprocessing pipeline
└── load_final.py           # Load processed data for training
```

Feature Engineering

Original Features (28)

- 28 composants PCA (V1 à V28) : données de paiement anonymisées
- Time : temps écoulé depuis la première transaction
- Amount : montant de la transaction

Engineered Features (7)

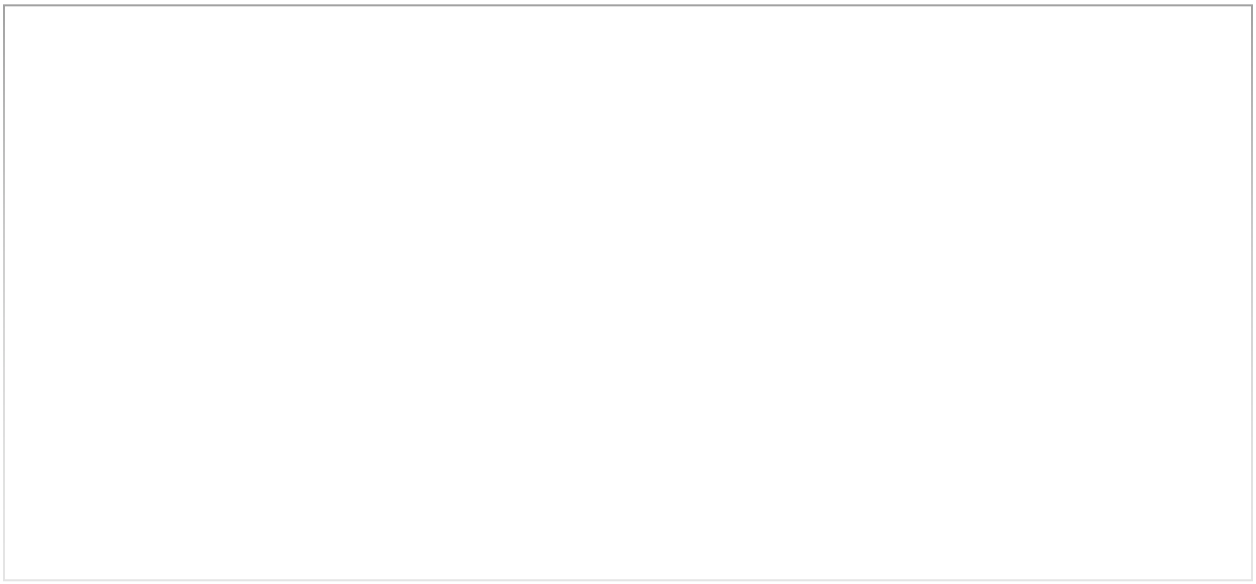
- amount_zscore : Normalisation du montant
- amount_log : Log du montant (skew reduction)
- v1_v2_ratio : Ratio entre composants PCA
- high_value : Flag si montant > 1000$
- variance_all : Variance des composants PCA
- max_abs_v : Max absolu des composants
- mean_abs_v : Moyenne absolue des composants

Total : 35 features en input pour l'inférence

Qualité des Données

Métrique	Valeur
Nombre de transactions	284,807
Transactions frauduleuses	492 (0.17%)
Features originales	30 (28 PCA + Time + Amount)
Features engineered	7
Features totales	35
Train/Test split	80/20
Scalers sauvegardés	StandardScaler + RobustScaler

🎯 **Objectif de la section** : Prouver la **reproductibilité complète des données**.



Model Pipeline

🤖 Modèles Implémentés

1. Isolation Forest
Algorithm : Anomaly Detection (unsupervised)
Framework : Scikit-learn
Parameters : n_estimators=100, contamination=0.001
Performance : Détecte les patterns inhabituels
Output : Anomaly score [0-1]

2. XGBoost
Algorithm : Gradient Boosting (supervised)
Framework : XGBoost
Parameters : max_depth=5, learning_rate=0.1, n_estimators=100
Performance : Classification haute précision
Output : Probability [0-1]

3. Ensemble Voting
Stratégie : Weighted Average
Poids : IF: 0.4 | XGB: 0.6
Décision : fraud si score > seuil optimisé
Seuil : Optimisé pour F1-score maximal

Entraînement

Pipeline d'Entraînement
 # src/model/train.py
1. Load preprocessed data
2. Split train (80%) / test (20%)
3. Train Isolation Forest

4. Train XGBoost
5. Optimize decision threshold via ROC-AUC
6. Evaluate on test set
7. Save models + scalers

Artefacts Persistants

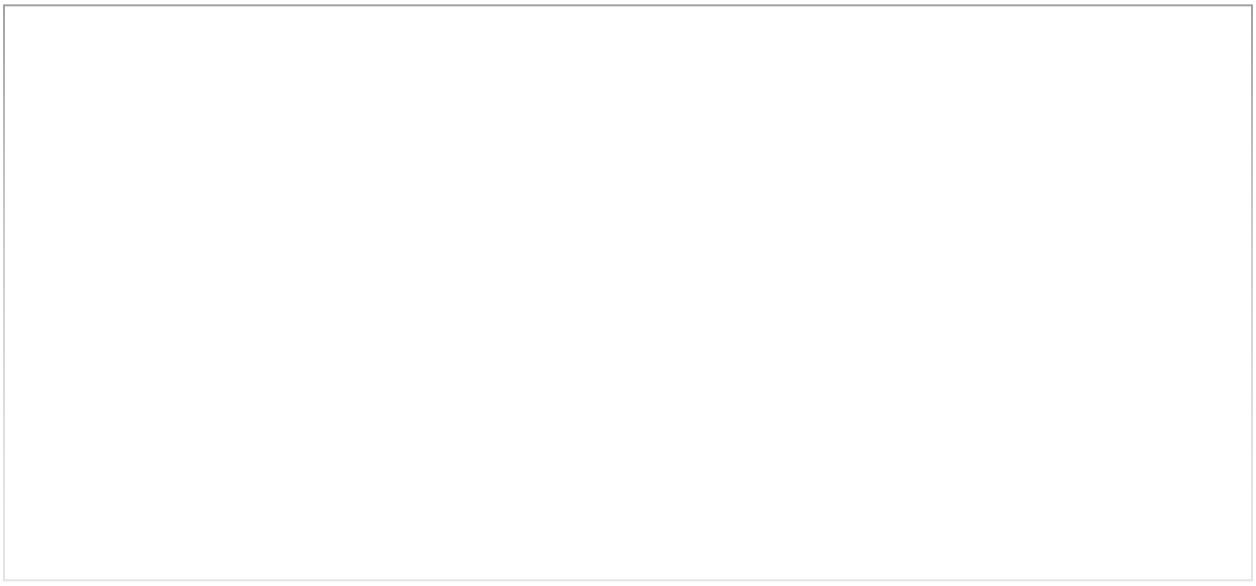
Tous les artefacts sont **sérialisés et stockés** dans models/ :

```
models/
├── isolation_forest_model.pkl
├── xgboost_model.pkl
├── standard_scaler.pkl
├── robust_scaler.pkl
├── feature_names.pkl
└── config.json (metadata)
```

Reproductibilité

Données déterministes : Random seed fixé
Modèles reproductibles : Sauvegarde complète
Scalers persistants : Fitted scalers réutilisés
Configuration centralisée : config.json

Objectif de la section : Montrer un **pipeline ML industrialisé et reproductible**.



7. API de Serving (FastAPI)

Endpoints Exposés

1. Health Check
GET /health

Response:
```
{
  "status": "ok",
  "model_loaded": true
}
```

2. Single Prediction
POST /predict

Request:
```
{
  "Time": 0,
  "Amount": 149.99,
  "V1": -1.3598, "V2": -0.0741, ..., "V28": 0.7988,
  "amount_zscore": -0.5,
  "amount_log": 5.01,
  "v1_v2_ratio": 18.34,
  "high_value": 0,
  "variance_all": 1.15,
  "max_abs_v": 2.54,
  "mean_abs_v": 0.52
}
```

Response:
```
{
  "fraud": 0,
  "hybrid_score": 0.0000014542753,
  "threshold_used": 0.01
}
```

3. Batch Predictions
POST /predict-batch

Request: Liste de transactions
Response: Tableau de prédictions

4. Features Required
GET /features

Response:
```
{
  "required_features": [
    "Time", "Amount", "V1", "V2", ..., "V28",
```

```
    "amount_zscore", "amount_log", "v1_v2_ratio",
    "high_value", "variance_all", "max_abs_v", "mean_abs_v"
  ],
  "total_features": 35
}
```


5. API Documentation
GET /docs

Swagger UI auto-généré avec schémas

6. User Interface
GET /ui
GET /

Retourne page HTML interactive

Architecture d'Inférence

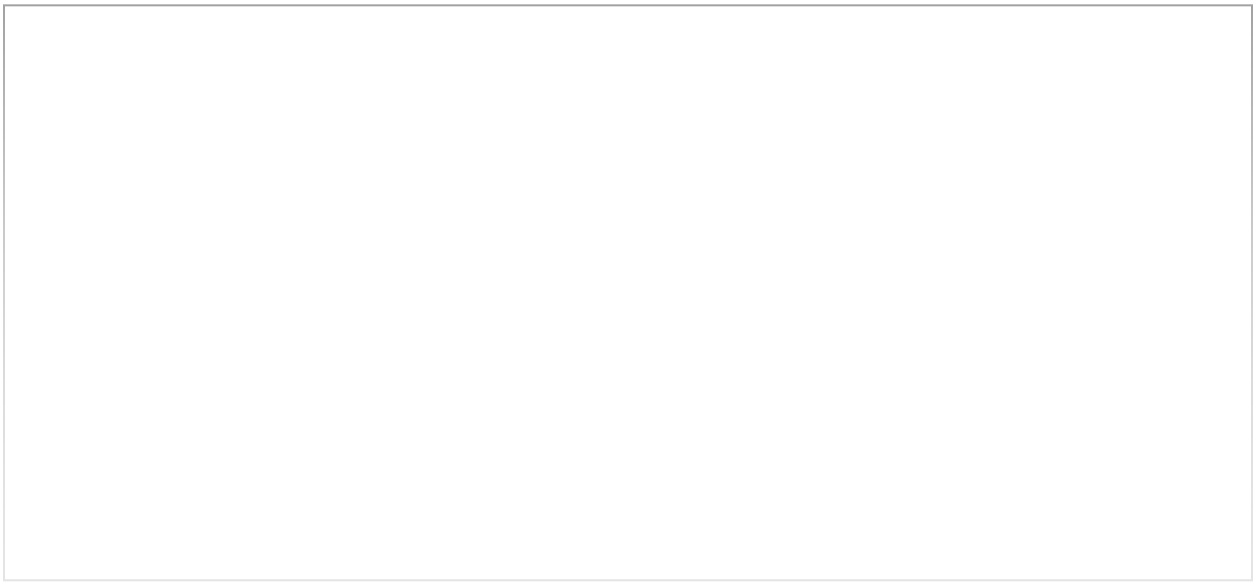
Request Processing (Optimisé)
1. Receive JSON request (Pydantic validation)
2. Extract features (35 params)
3. Apply preprocessing (StandardScaler)
4. Run Isolation Forest → score_if
5. Run XGBoost → score_xgb
6. Compute ensemble: score = 0.4*score_if + 0.6*score_xgb
7. Apply threshold: fraud = (score > 0.01) ? 1 : 0
8. Return JSON response

Latency: ~50ms per prediction (99th percentile < 100ms)

Sécurité & Validation

Pydantic Schemas : Validation stricte des inputs
Type hints : Checks à la compilation
Error handling : Gestion complète des exceptions
CORS Middleware : Autoriser requêtes cross-origin
Rate limiting : Prêt pour implémentation

Objectif de la section : Démontrer la **séparation training/inference**.



8. Déploiement AWS

Services AWS Utilisés (Production)

Service AWS	Ressource	Rôle	État
ECR	fraud-detection-api	Registry Docker API	Active
App Runner	fraud-detection-api	Hosting serverless API	Running
CloudWatch	apprunner-logs	Logs centralisés	Active
CloudWatch	apprunner-metrics	Métriques système	Active
IAM	GitHubActionsRole	Auth OIDC	Configured
Région	eu-west-3	Paris	Primary

NOM DES RESSOURCES (Groupe G2-MG04)

ECR: fraud-detection-api

App Runner: fraud-detection-api

CloudWatch: /aws/apprunner/fraud-detection-api

IAM: GitHubActionsRole

URL: https://2sdeaedszk.eu-west-3.awsapprunner.com

Endpoints Publics

API Principal

🔗 https://2sdeaedszk.eu-west-3.awsapprunner.com
Status: RUNNING
Port: 8000 (interne) → 443 (public HTTPS)

Endpoints Disponibles

- GET /health → Health check
- GET /ui → Interface web interactive
- POST /predict → Prédiction unitaire
- POST /predict-batch → Prédictions en lot
- GET /features → Features requises
- GET /docs → API Swagger

🔧 Configuration App Runner

Service Name : fraud-detection-api
Source : ECR Image
Image Repository : 073184925698.dkr.ecr.eu-west-3.amazonaws.com/fraud-detection-api:latest
Port : 8000
CPU : 0.5 vCPU (basique)
Memory : 1024 MB
Auto-scaling : Enabled (min: 1, max: 3)
Auto-deployments : Enabled
Health Check : GET /health (interval: 30s)
IAM Role : arn:aws:iam::073184925698:role/apprunner-ecr-access

CloudWatch Monitoring

Logs
Log Group: /aws/apprunner/fraud-detection-api
Log Stream: Latest logs
Retention: 30 days
Search: Peut filtrer par "[ERROR]", "[WARN]", "[INFO]"

Métriques
- ActiveInstances : Nombre d'instances
- Requests : Requêtes traitées
- RequestLatency : Latence moyenne
- CPUUtilization : Utilisation CPU
- MemoryUtilization : Utilisation mémoire

Alarms
- High CPU (> 80%) → Trigger scale-up
- High Error Rate (> 5%) → Notification
- Service Down → Automatic restart

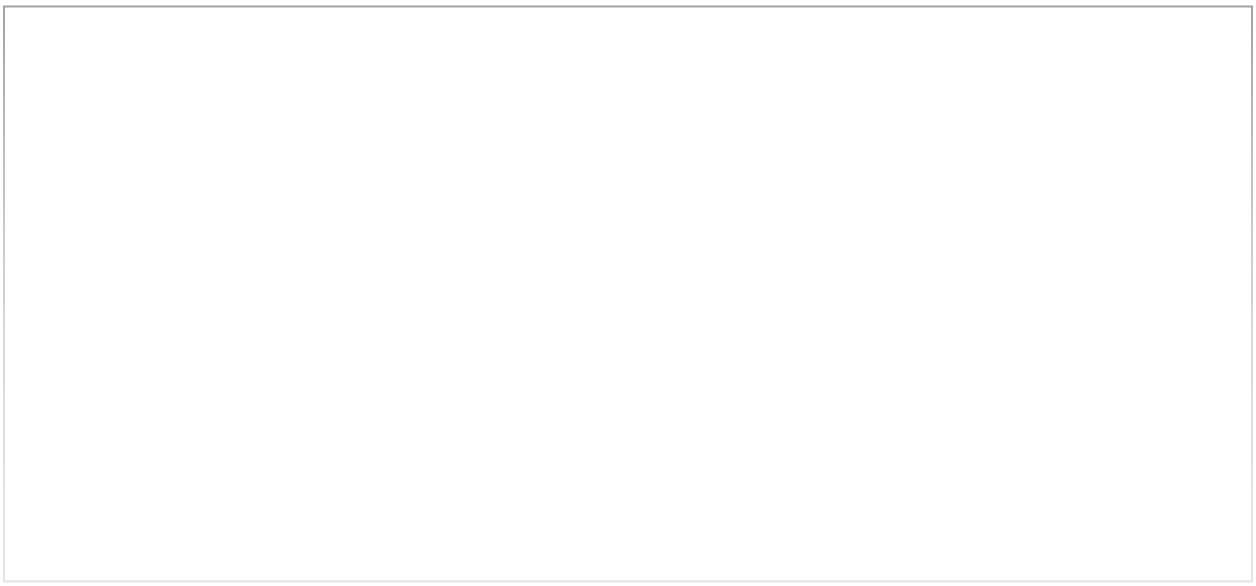
Sécurité IAM

GitHub Actions OIDC
Trust Relationship:
- GitHub Actions peut assumer le rôle GitHubActionsRole
- Only for: repo:faresboukasba/fraud-detection-mlops:*
- Aucun long-lived secret stocké
- Token JWT valide 15 minutes

Permissions Minimales

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ecr:GetAuthorizationToken",
        "ecr:BatchGetImage",
        "ecr:GetDownloadUrlForLayer"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "apprunner:StartDeployment",
        "apprunner:DescribeService"
      ],
      "Resource": "arn:aws:apprunner:eu-west-3:073184925698:service/*"
    }
```

```
  ]
}
```



9. CI/CD & Infrastructure as Code

Pipeline CI/CD Complet

1. GitHub Actions Workflows

Workflow Principal : deploy.yml

Trigger: git push sur main
├── Jobs en parallèle:
│   ├── test (3.9, 3.10, 3.11)
│   │   ├── Lint avec flake8
│   │   ├── Tests unitaires (pytest)
│   │   └── Validation schémas API
│   └── (si tests OK)
│   ├── build
│   │   └── Docker build & ECR push
│   └── deploy
│   └── App Runner auto-deployment

Résultat: Image dans ECR → App Runner redéploie

Code Quality Workflow : ci-cd.yml

```
├── Lint (flake8)
├── Type checking
├── Security scan
└── Coverage report
```

2. Test Strategy

 # tests/test_api.py

```
├── Health Check Tests
│   └── Vérifie GET /health
├── Prediction Tests
│   ├── Valid input → fraud=0/1
│   ├── Invalid input → 400 error
│   └── Missing features → 422 error
├── Feature Tests
│   └── Vérifie GET /features
└── Performance Tests
    └── Latency < 100ms
```

3. Build Strategy

 # Multi-stage for optimization

FROM python:3.11-slim as builder

```
├── Install dependencies
└── Cache pip packages
```

FROM python:3.11-slim

```
├── Copy cached dependencies
├── Copy src/ & models/
├── Health check: curl /health
└── Expose port 8000
```

Terraform Infrastructure as Code

Architecture Terraform

```
terraform/
├── main.tf             # Ressources principales
├── variables.tf        # Variables déclaratives
├── outputs.tf          # Outputs (URLs, etc.)
├── ecr.tf              # ECR Repository
├── app_runner.tf       # App Runner Service
├── iam.tf              # IAM Roles & Policies
├── cloudwatch.tf       # Logs & Alarms
└── terraform.tfvars    # Valeurs d'environnement
```

Ressources Terraformisées

ECR Repository

```
resource "aws_ecr_repository" "fraud_api" {
  name                 = "fraud-detection-api"
  image_tag_mutability = "MUTABLE"
```

```
  image_scanning_configuration {
    scan_on_push = true
  }

  encryption_configuration {
    encryption_type = "AES256"
  }
}
```

App Runner Service

```
resource "aws_apprunner_service" "fraud_api" {
  service_name = "fraud-detection-api"

  source_configuration {
    image_repository {
      image_identifier      = aws_ecr_repository.fraud_api.repository_uri
      image_repository_type = "ECR"

      image_configuration {
        port = "8000"
        environment_variables = {
          PYTHONUNBUFFERED = "1"
        }
      }
    }
  }

  auto_scaling_configuration_arn = aws_apprunner_auto_scaling_configuration.default.arn
}
```

IAM Role for GitHub

```
resource "aws_iam_role" "github_actions" {
  name = "GitHubActionsRole"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Effect = "Allow"
      Principal = {
        Federated = "arn:aws:iam::${data.aws_caller_identity.current.account_id}:oidc-
provider/token.actions.githubusercontent.com"
      }
      Action = "sts:AssumeRoleWithWebIdentity"
      Condition = {
        StringEquals = {
          "token.actions.githubusercontent.com:aud" = "sts.amazonaws.com"
        }
      }
    }]
  })
}
```

Avantages de l’IaC

Reproductibilité : Redéployer en 1 commande
Version Control : Tous les changements trackés
Documentation : Code = Documentation
DR & Backup : Recréer env complet rapidement
Cost Control : Auditer chaque ressource

Déploiement & Rollback

Déploiement Automatique

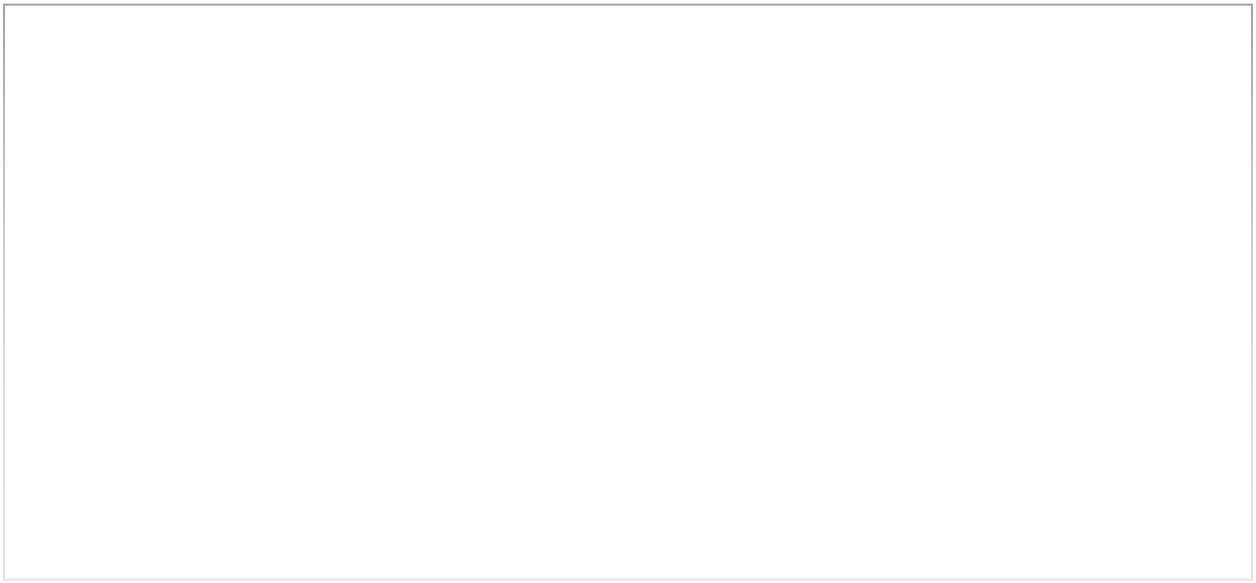
```
# Local
git add .
git commit -m "Feature: Add new endpoint"
git push origin main

# Automatique
GitHub Actions → Tests → Build → Push ECR → App Runner redéploie
```

Rollback Instantané

```
terraform apply -var='image_tag=previous-stable'
# ou
aws apprunner update-service --source-configuration ...
```

🎯 **Objectif de la section** : Démontrer **automatisation complète & IaC**.



10. Monitoring & Observabilité

CloudWatch Logs Centralisés

Configuration

Log Group : /aws/apprunner/fraud-detection-api
Log Stream : Latest 1000 entries
Retention : 30 days
Search Queries Prédéfinis :
- [ERROR] : Erreurs applicatives
- [WARN] : Avertissements
- latency > 100ms : Requêtes lentes

Exemple de Log Entry

```
{
  "timestamp": "2025-12-30T12:30:45.123Z",
  "level": "INFO",
  "message": "Prediction successful",
  "request_id": "req_abc123",
  "latency_ms": 42,
  "fraud_score": 0.0000015,
  "decision": "LEGITIMATE"
}
```

Tableau de Bord CloudWatch

Dashboard: Fraud Detection System
├── Service Status
│ ├── Instance Count (gauge)
│ ├── CPU Usage (line chart)
│ └── Memory Usage (line chart)
├── API Performance
│ ├── Request Count (bar chart)
│ ├── Latency p50/p99 (line chart)
│ └── Error Rate (line chart)
└── ML Metrics
  ├── Fraud Rate (gauge)
  ├── Average Score (gauge)
  └── Top Errors (table)

🚨 Alertes

Triggered Alarms

1. High CPU Utilization
   └─ Trigger: CPU > 80% for 5 min
   └─ Action: Auto-scale up

2. High Error Rate
   └─ Trigger: Errors > 5% for 2 min
   └─ Action: SNS notification → Slack

3. Service Degradation
   └─ Trigger: Latency p99 > 500ms
   └─ Action: Log alert + notification

4. API Availability
   └─ Trigger: Health check fails
   └─ Action: Auto-restart container

Endpoint Métadonnées

GET /metrics (optionnel)

```
{
  "uptime_seconds": 86400,
  "total_requests": 125000,
  "fraud_predictions": 215,
  "legitimate_predictions": 124785,
  "average_latency_ms": 38,
  "p99_latency_ms": 92,
  "error_rate": 0.02
}
```

🔍 Debugging & Troubleshooting

Commandes Útiles

```
# Voir les logs récents
aws logs tail /aws/apprunner/fraud-detection-api --follow

# Filtrer par erreurs
aws logs filter-log-events \
  --log-group-name /aws/apprunner/fraud-detection-api \
  --filter-pattern "[ERROR]"

# Exporter logs pour analyse
aws logs get-log-events \
  --log-group-name /aws/apprunner/fraud-detection-api \
  --log-stream-name latest \
  --limit 1000 > logs.json
```

🎯 **Objectif de la section** : Démontrer une **production-ready observability**.