



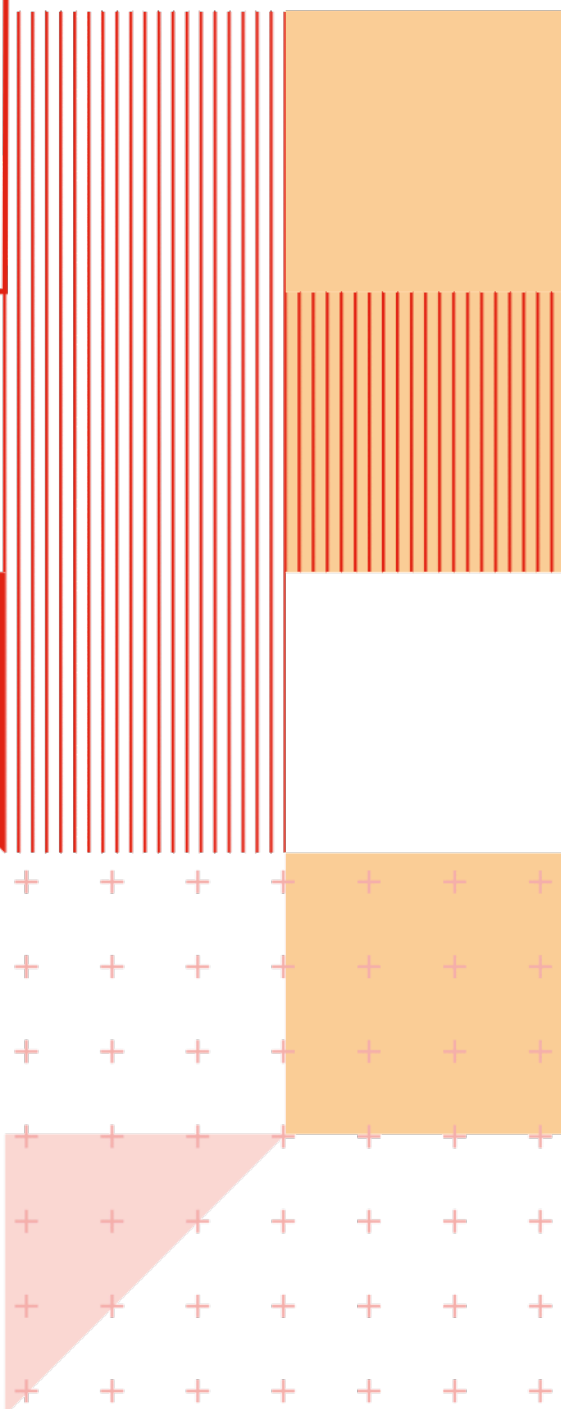
INSTITUT NATIONAL  
DES SCIENCES  
APPLIQUÉES  
CENTRE VAL DE LOIRE

## Rapport de Projet : Déploiement & Infrastructure

**BARBIER Ted**

Département STI

4ème Année



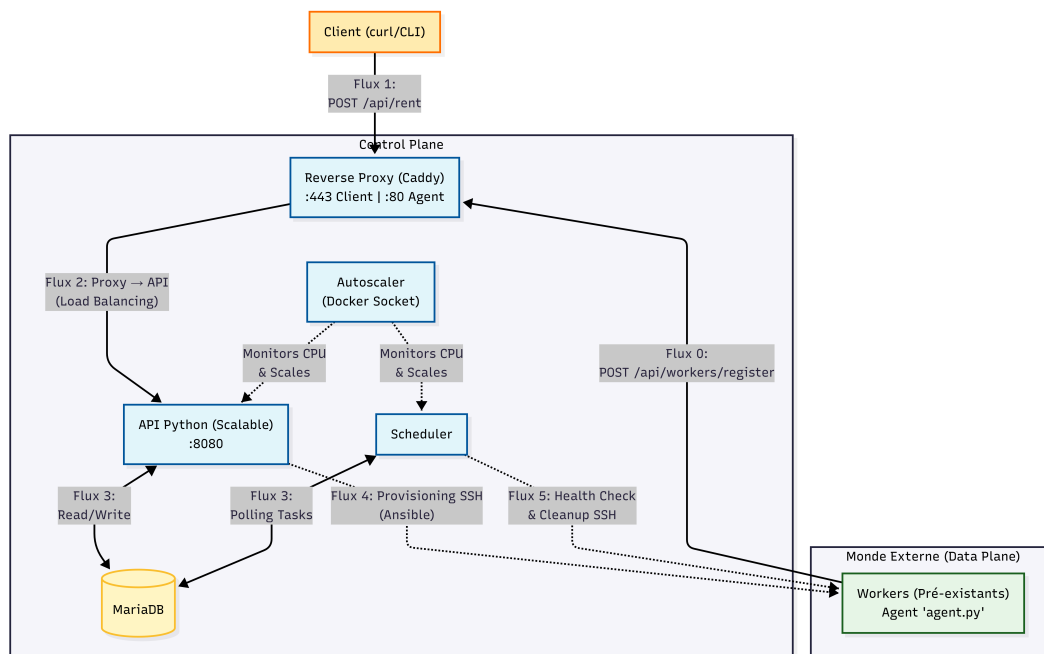
## Table des matières

1.	Introduction .....	2
2.	Vue d'Ensemble & Philosophie .....	2
2.1.	Pourquoi cette Architecture ? .....	2
3.	Architecture Système : Philosophie Micro-services .....	2
3.1.	Le Data Plane : Les Workers « Opportunistes » .....	3
3.2.	Le Control Plane : L'intelligence orchestrée .....	3
4.	Choix Techniques & Justifications Critiques .....	3
4.1.	Pourquoi Python et l'API REST ? .....	3
4.2.	MariaDB : Le choix de l'intégrité transactionnelle .....	3
4.3.	Sécurité, Isolation et Monétisation via Ansible .....	3
5.	Analyse de la Stratégie de Développement : Le Cas Vagrant .....	4
5.1.	L'intention : Un environnement universel .....	4
5.2.	L'impasse technique : Conflit d'architecture (ARM vs x86) .....	4
6.	Implémentation de la Résilience .....	4
6.1.	Gestion de la Concurrency massive : SKIP LOCKED .....	4
6.2.	Cycle de vie d'une ressource (Workflow) .....	4
7.	Stratégie de Test et Validation .....	4
7.1.	Couverture de Code .....	4
7.2.	Méthodologie : Mocking .....	5
8.	Démonstration et Scénario Réaliste .....	5
8.1.	Scénario de la Démonstration .....	5
9.	Conclusion .....	5
9.1.	Perspectives d'évolution .....	5
10.	Annexes Techniques .....	6
10.1.	Détails des Endpoints API .....	6
10.2.	Structure du Projet .....	6
10.3.	Commandes Utiles .....	6

## 1. Introduction

Dans le cadre de l'unité d'enseignement « Déploiement & Infrastructure as Code », ce projet vise à concevoir un prototype de **Platform as a Service** (PaaS) baptisé **Orion-Dynamic**.

L'objectif est de gérer dynamiquement un parc de machines (les « Workers »), de les louer à des clients pour une durée déterminée, et d'assurer la continuité de service même en cas de panne de matériel. Ce projet explore le concept d'**infrastructure opportuniste** : les ressources de calcul ne sont pas connues à l'avance par le serveur, mais s'enregistrent spontanément.



**Figure 1** : Architecture Globale du projet Orion-Dynamic

## 2. Vue d'Ensemble & Philosophie

L'architecture d'Orion-Dynamic n'est pas fortuite ; elle répond à des contraintes précises de scalabilité et de résilience typiques des environnements Cloud modernes.

### 2.1. Pourquoi cette Architecture ?

Le système a été pensé pour résoudre trois problèmes fondamentaux :

1. **L'hétérogénéité des ressources** : Les workers peuvent être n'importe où (Cloud, Edge, On-premise) et derrière n'importe quel réseau (NAT). D'où le choix d'un modèle **Push** où l'agent initie la connexion.
2. **La panne comme norme** : Dans un parc volatil, la panne n'est pas une exception mais un état attendu. Tout le Control Plane (Scheduler, Autoscaler) est conçu pour **réagir** à la panne plutôt que de tenter de l'empêcher (Self-Healing).
3. **L'isolation stricte** : La sécurité des clients est garantie par une séparation totale des environnements via des comptes UNIX éphémères, orchestrés par Ansible.

## 3. Architecture Système : Philosophie Micro-services

Le système repose sur une séparation stricte entre le plan de contrôle (**Control Plane**) et le plan de données (**Data Plane**).

### 3.1. Le Data Plane : Les Workers « Opportunistes »

Le **Data Plane** est constitué des ressources de calcul. Dans notre simulation, ces machines sont représentées par des conteneurs légers (Alpine Linux) exécutant un serveur SSH.

Chaque Worker embarque un **Agent** (`agent.py`). Au démarrage de la machine, cet agent a pour unique responsabilité de contacter l'API centrale pour signaler sa présence.

- **Choix du mode Push** : Contrairement à une configuration statique où le serveur doit scanner le réseau (Pull), c'est ici le nœud qui « pousse » ses informations. Cela permet d'outrepasser les contraintes de NAT/Firewall et d'ajouter des capacités de calcul instantanément sans reconfigurer le serveur central.

### 3.2. Le Control Plane : L'intelligence orchestrée

Le **Control Plane** est composé de plusieurs micro-services conteneurisés, chacun ayant une responsabilité unique :

**Reverse Proxy (Caddy)** Sert d'API Gateway unique. Il assure l'équilibrage de charge (**Load Balancing**) vers les réplicas de l'API.

**API (Python/FastAPI)** Le cœur réactif du système. Elle gère l'enregistrement des workers et les baux clients. Elle est strictement « Stateless ».

**Scheduler** Assure la cohérence de l'infrastructure. Il effectue les « Health Checks », les migrations et le nettoyage des baux expirés.

**Autoscaler** Un service de régulation en boucle fermée qui ajuste le nombre de réplicas de l'API et du **Scheduler** en fonction de la charge CPU réelle détectée sur le socket Docker.

**Base de Données (MariaDB)** Stocke l'état du parc, les utilisateurs et les contrats de location (baux).

## 4. Choix Techniques & Justifications Critiques

### 4.1. Pourquoi Python et l'API REST ?

Le choix de Python s'est imposé pour sa rapidité de prototypage et sa compatibilité native avec les outils d'automatisation comme Ansible.

- **Justification REST** : Nous avons privilégié REST plutôt que gRPC ou des brokers de messages. Dans un contexte de PaaS ouvert, REST est le standard universel : n'importe quel terminal peut s'enregistrer via une simple requête HTTP JSON, garantissant une interopérabilité maximale.

### 4.2. MariaDB : Le choix de l'intégrité transactionnelle

Le choix d'une base SQL (MariaDB) plutôt que NoSQL est dicté par le besoin de **cohérence forte**.

- **Problématique de concurrence** : Deux clients ne doivent jamais pouvoir louer la même machine simultanément.
- **Solution** : Les transactions ACID de MariaDB permettent de verrouiller l'état d'un worker lors de son allocation, rendant l'opération atomique.

### 4.3. Sécurité, Isolation et Monétisation via Ansible

Le but du service est de louer une ressource tout en garantissant qu'elle reste sous notre contrôle technique (condition sine qua non pour la facturation).

- **Stratégie** : Nous utilisons Ansible pour créer un utilisateur temporaire avec des droits restreints.
- **Justification** : Plutôt que de donner un accès root (risqué), nous isolons le client. Ansible assure l'idempotence : si la création échoue, le système peut retenter sans corrompre la machine. À l'expiration du bail, Ansible supprime l'utilisateur, garantissant que la ressource redevient disponible pour un nouveau cycle de facturation.

## 5. Analyse de la Stratégie de Développement : Le Cas Vagrant

Une phase importante du projet, **imposée par le cahier des charges**, visait à mettre en place un **environnement de développement** via Vagrant.

### 5.1. L'intention : Un environnement universel

L'objectif était d'utiliser Vagrant pour créer une machine virtuelle (VM) « bac à sable » standardisée. Cette VM devait héberger l'intégralité du projet afin de garantir un environnement de dev identique pour tous. Cependant, cette solution a dû être abandonnée en raison de contraintes matérielles insurmontables sur macOS.

### 5.2. L'impasse technique : Conflit d'architecture (ARM vs x86)

La tentative a été abandonnée après environ deux heures de tests infructueux.

- **Cause du blocage** : Le développement s'est déroulé sur une architecture **Apple Silicon (ARM64)**. La virtualisation d'images Linux standard (souvent optimisées pour x86\_64) au sein de Vagrant a provoqué des instabilités majeures, des plantages du processus de provisionnement et des timeouts I/O lors de l'installation de Docker.
- **Décision d'ingénierie** : Le choix a été fait de pivoter vers une approche **Native Docker Compose**.
- **Leçon retenue** : Si la virtualisation (Vagrant) offre une isolation supérieure, elle introduit une couche de complexité (virtualisation imbriquée) parfois incompatible avec les nouvelles architectures matérielles. La conteneurisation native s'est révélée être un compromis plus pragmatique et performant.

## 6. Implémentation de la Résilience

### 6.1. Gestion de la Concurrence massive : SKIP LOCKED

Un défi majeur est d'éviter que deux instances du Scheduler ne traitent le même worker simultanément (ex: deux migrations pour une même panne).

- **Solution technique** : Utilisation de la clause SQL `SELECT ... FOR UPDATE SKIP LOCKED`.
- **Analyse** : Cela permet de paralléliser le Control Plane. Chaque instance « pioche » une tâche libre sans bloquer les autres instances. C'est ce qui permet au système de supporter des milliers de nœuds sans ralentissement.

### 6.2. Cycle de vie d'une ressource (WorkFlow)

1. **Découverte** : Le Worker s'annonce via `POST /api/workers/register`.
2. **Allocation** : Le client appelle `POST /api/rent`. L'API sélectionne un nœud, exécute le playbook Ansible et renvoie les accès.
3. **Surveillance** : Le Scheduler effectue un Health Check (Ping SSH) toutes les 30s. Après 3 échecs, la **Migration** est déclenchée : le client est déplacé sur un nœud sain de manière transparente.
4. **Libération/Extension** : Le client peut prolonger son bail via `/api/extend` ou laisser le Scheduler nettoyer la machine à l'échéance.

## 7. Stratégie de Test et Validation

La fiabilité du système est garantie par une suite de tests unitaires rigoureuse exécutée via pytest.

### 7.1. Couverture de Code

Le projet atteint une couverture globale d'environ **82%** sur les 58 tests implémentés :

- **Autoscaler (90%)** : Validation des algorithmes de scaling et des commandes Docker.
- **API (86%)** : Test exhaustif des endpoints, de l'authentification JWT et de la gestion des erreurs.

- **Scheduler (76%)** : Vérification des boucles de contrôle, des Health Checks et de la logique d'expiration.

## 7.2. Méthodologie : Mocking

Afin d'assurer des tests rapides et isolés, nous utilisons le **Mocking** pour les composants externes :

- **Base de Données** : Les appels SQL sont interceptés pour simuler des états de données sans nécessiter de serveur MariaDB réel.
- **Ansible** : L'exécution des playbooks est simulée pour vérifier la logique d'appel sans provisionner de vraies machines.

## 8. Démonstration et Scénario Réaliste

Le script `full_demo.sh` permet de jouer un scénario complet démontrant les capacités d'auto-guérison (**Self-Healing**) de la plateforme.

### 8.1. Scénario de la Démonstration

1. **Infrastructure & Scaling** : Démarrage du Control Plane et scaling des services (API, Scheduler) pour prouver le Load Balancing.
2. **Authentification** : Création de comptes et validation des jetons de sécurité.
3. **Location & Panne** : Un utilisateur loue un worker. Une panne brutale est simulée (arrêt du conteneur worker).
4. **Migration Automatique** : Le Scheduler détecte la perte du nœud et migre instantanément le client vers un nouveau worker sain.
5. **Résurrection & Sécurité** : Le nœud défaillant est redémarré. Le système détecte son retour et déclenche un **nettoyage préventif** (suppression du compte utilisateur) pour éviter tout accès résiduel non autorisé.

## 9. Conclusion

Le projet **Orion-Dynamic** valide la possibilité de construire un service PaaS robuste sur un inventaire volatil. Les choix techniques effectués privilégient la résilience (Migration auto), la scalabilité (Autoscaling) et l'intégrité (SQL/Ansible). Le pivot stratégique de Vagrant vers Docker illustre la capacité d'adaptation nécessaire à la gestion de projets complexes sur des architectures hétérogènes.

### 9.1. Perspectives d'évolution

Bien que fonctionnel, le projet pourrait bénéficier de plusieurs améliorations majeures :

- **Infrastructure as Code (IaC)** : Migration vers **Terraform** pour un provisionnement plus granulaire et déclaratif de l'infrastructure cloud.
- **Observabilité** : Ajout d'une stack de monitoring (Prometheus/Grafana) pour visualiser en temps réel la santé du cluster et les métriques de performance.
- **Marketplace de Services** : Évolution du modèle de location de « Workers nus » vers des « Services Managés ». L'idée serait de proposer un catalogue (ex: Serveur Web, Simulateur DDoS, Cache Redis) et d'adapter l'API, la BDD et le Frontend pour permettre au client de louer non plus une machine, mais une fonctionnalité précise pré-configurée.

## 10. Annexes Techniques

### 10.1. Détails des Endpoints API

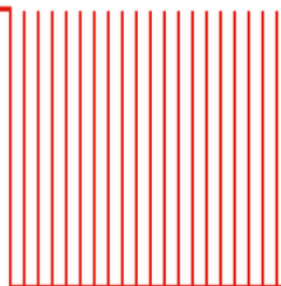
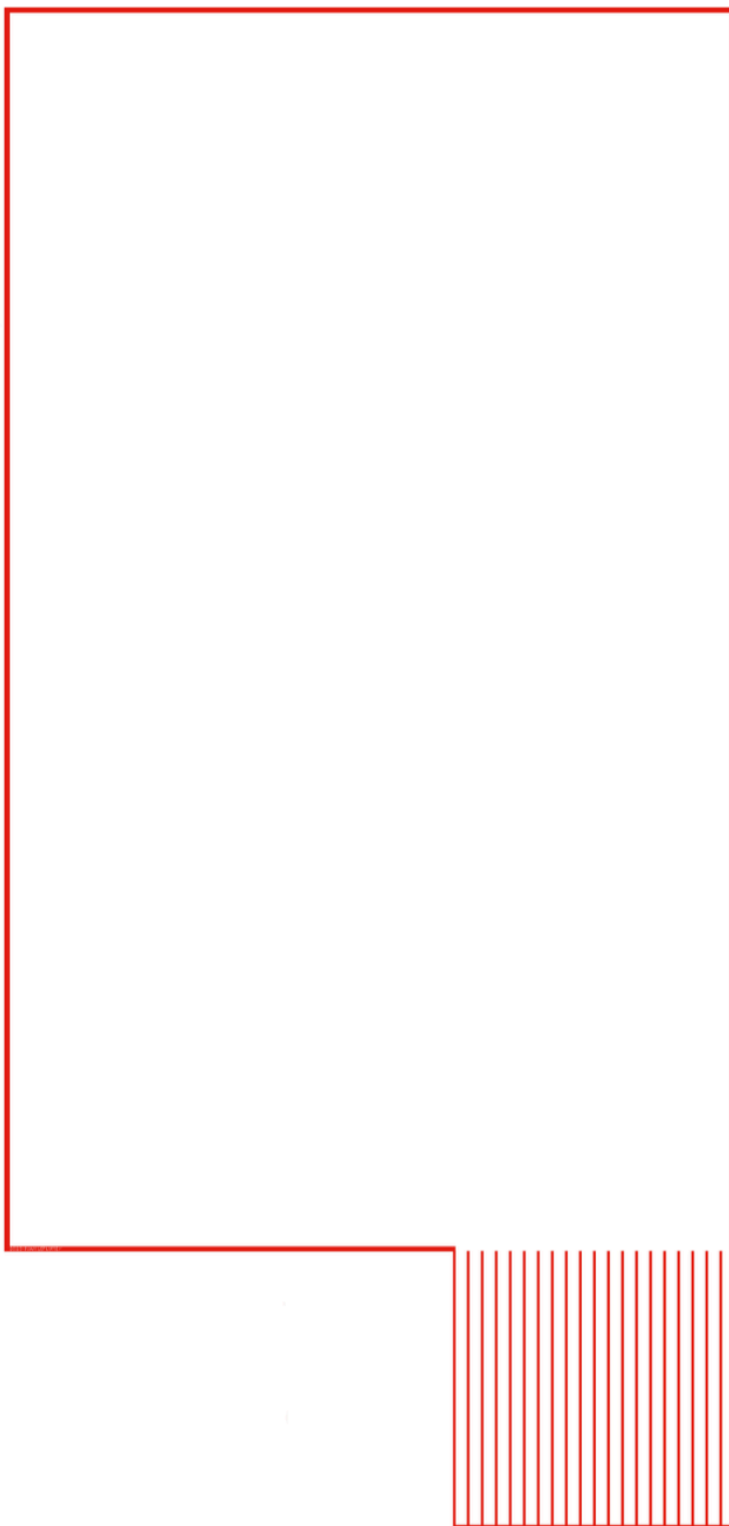
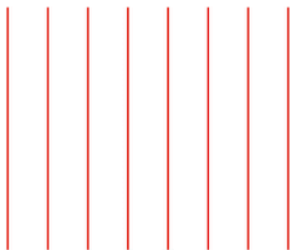
- POST /api/signup : Création de compte utilisateur.
- POST /api/login : Authentification et génération de JWT.
- POST /api/rent : Location de workers (allocation dynamique).
- GET /api/nodes : Monitoring en temps réel du parc.

### 10.2. Structure du Projet

- /control-plane : Micro-services Python et Dockerfiles.
- /playbooks : Logic Ansible d'isolation utilisateur.
- Caddyfile : Configuration du Load Balancer.
- launch\_workers.sh : Script de scalabilité du Data Plane.

### 10.3. Commandes Utiles

- Lancer la démo : `cd orion-dynamic && ./full_demo.sh`
- Lancer les tests : `cd orion-dynamic && ./run_unit_tests.sh`



## **INSA Centre Val de Loire**

88 boulevard Lahitolle

CS 60013

18022 Bourges cedex

Tél : + 33 (0)2 48 48 40 00

**[www.insa-centrevaldeloire.fr](http://www.insa-centrevaldeloire.fr)**



**INSA** INSTITUT NATIONAL  
DES SCIENCES  
APPLIQUÉES  
CENTRE VAL DE LOIRE

  
**MINISTÈRE  
DE L'ENSEIGNEMENT  
SUPÉRIEUR  
ET DE LA RECHERCHE**  
*Liberté  
Égalité  
Fraternité*