École Nationale Supérieure
d'Informatique et d'Analyse des Systèmes
- Rabat

Projet de Fin d'Année

Implementation and Optimization of collaborative Strategies in Microservices Architecture using Game Theory and Agent-Based Models

Élèves :
MOHAMED CHADI TAQI
SAAD ALAOUI SOSSE
CHAYMAE BOUAZZA
IMANE BENABBOU

Encadrant :
DR HATIM GUERMAH

Jury :
Dr HATIM GUERMAH
Dr YOUNES TABII

Année Universitaire 2024-2025

Remerciements

C'est avec une profonde reconnaissance que nous dédions ces premières pages à tous ceux qui ont contribué, de près ou de loin, à l'élaboration et à la réussite de ce projet fédérateur. Nous tenons avant tout à exprimer notre infinie gratitude à notre encadrant,

Monsieur Hatim Guermah. Sa disponibilité constante, sa rigueur scientifique et ses directives pédagogiques ont été d'une valeur inestimable tout au long de ce travail. Ses conseils avisés nous ont non seulement guidés dans la résolution des défis techniques liés aux systèmes distribués et à l'algorithmique complexe de ce projet, mais nous ont également poussés à approfondir notre réflexion et à viser l'excellence. Nous sommes

également très honorés de présenter ce travail devant un jury éminent. Nous adressons nos vifs remerciements à **Monsieur Tabii Younes** ainsi qu'à **Monsieur Guermah** pour l'intérêt qu'ils ont bien voulu porter à notre projet en acceptant de l'examiner. Nous sommes impatients de bénéficier de leurs critiques constructives et de leur expertise pour perfectionner davantage notre approche. Enfin, nous souhaitons témoigner notre

reconnaissance envers l'ensemble du corps professoral de la filière **Génie Logiciel** pour la qualité de l'enseignement dispensé et l'encadrement offert durant ces années de formation. Nous remercions également le cadre administratif de l'**ENSIAS** pour avoir assuré un environnement d'apprentissage stimulant et propice à l'innovation, atout majeur dans notre parcours de futurs ingénieurs.

Résumé

La gestion efficace des ressources dans les environnements de conteneurisation, tels que Kubernetes, représente un défi majeur pour les infrastructures cloud modernes. Les approches traditionnelles reposent souvent sur des configurations statiques (*requests* et *limits*) qui peinent à s'adapter à la variabilité des charges de travail, entraînant soit un gaspillage de ressources (sur-allocation), soit une dégradation des performances due au phénomène de *CPU throttling*. Ce projet fédérateur introduit **MBCAS** (Market-Based CPU Allocation System), une solution innovante combinant la modélisation multi-agents et la théorie des jeux pour optimiser l'allocation dynamique du processeur. Notre approche

modélise chaque pod comme un agent autonome (*PodAgent*) utilisant l'apprentissage par renforcement, spécifiquement l'algorithme **Q-Learning**, pour apprendre des stratégies d'enchères optimales basées sur ses métriques d'utilisation réelles et ses objectifs de niveau de service (SLO). Ces agents interagissent au sein d'un nœud via un mécanisme de négociation coopérative résolu par la **Solution de Négociation de Nash**. Cette méthode garantit une distribution des ressources qui est à la fois équitable, proportionnelle aux priorités des services, et Pareto-efficace, assurant qu'aucune capacité CPU n'est gaspillée tant qu'une demande existe. D'un point de vue technique, le système est implé-

menté en Go et s'intègre nativement à l'écosystème Kubernetes via des *Custom Resource Definitions* (CRDs). Il exploite les métriques fines du noyau Linux (Cgroups v2) et utilise la fonctionnalité récente d'*In-Place Pod Vertical Scaling* pour appliquer les nouvelles allocations en temps réel sans nécessiter le redémarrage des conteneurs. Les résultats démontrent que MBCAS réduit significativement le throttling tout en maintenant une utilisation élevée du cluster, offrant ainsi une alternative robuste aux auto-scaleurs verticaux traditionnels (VPA). **Mots-clés :** Kubernetes, Allocation de Ressources, Système

Multi-Agents, Théorie des Jeux, Négociation de Nash, Q-Learning, Cgroups v2.

Abstract

Efficient resource management in containerized environments, such as Kubernetes, poses a significant challenge for modern cloud infrastructures. Traditional approaches often rely on static configurations (*requests* and *limits*) that struggle to adapt to workload variability, resulting in either resource wastage (over-provisioning) or performance degradation due to *CPU throttling*. This capstone project introduces **MBCAS** (Market-Based CPU Allocation System), an innovative solution combining multi-agent modeling and game theory to optimize dynamic CPU allocation. Our approach models each pod as an autonomous agent (*PodAgent*) using reinforcement learning—specifically the **Q-Learning** algorithm—to learn optimal bidding strategies based on real-time usage metrics and Service Level Objectives (SLOs). These agents interact within a node through a cooperative bargaining mechanism solved by the **Nash Bargaining Solution**. This method guarantees a resource distribution that is simultaneously fair, proportional to service priorities, and Pareto-efficient, ensuring that no CPU capacity is wasted as long as demand exists. From a technical standpoint, the system is implemented in Go and integrates natively with the Kubernetes ecosystem via *Custom Resource Definitions* (CRDs). It leverages fine-grained Linux kernel metrics (Cgroups v2) and utilizes the recent *In-Place Pod Vertical Scaling* feature to apply new allocations in real-time without requiring container restarts. The results demonstrate that MBCAS significantly reduces throttling while maintaining high cluster utilization, thereby offering a robust alternative to traditional Vertical Pod Autoscalers (VPA). **Keywords :** Kubernetes, Resource Allocation, Multi-Agent System, Game Theory, Nash Bargaining, Q-Learning, Cgroups v2.

Introduction Générale

Le cursus d'ingénieur en Génie Logiciel à l'ENSIAS a pour vocation de former des profils capables d'appréhender la complexité croissante des systèmes distribués modernes. Dans cette perspective, le **Projet Fédérateur** constitue une étape charnière, offrant l'opportunité de synthétiser les compétences acquises de l'architecture logicielle à l'intelligence artificielle, en passant par l'administration système autour d'une problématique concrète et innovante. C'est dans ce cadre que s'inscrit notre projet : **MBCAS (Market-Based CPU Allocation System)**.

L'industrie du logiciel a vu, ces dernières années, une adoption massive des technologies de conteneurisation, avec Kubernetes s'imposant comme le standard de facto pour l'orchestration. Cependant, malgré sa robustesse, la gestion dynamique des ressources (CPU et mémoire) reste un défi ouvert. Les pratiques actuelles reposent majoritairement sur des configurations statiques (*requests* et *limits*) définies par les développeurs. Cette approche mène invariablement à deux écueils : le sur-dimensionnement (*over-provisioning*), coûteux et inefficace, ou le sous-dimensionnement, entraînant du *CPU throttling* et dégradant la qualité de service (QoS).

Face à ces limitations, ce projet propose une rupture paradigmatique en traitant l'allocation de ressources non plus comme un problème d'ordonnancement statique, mais comme un **marché économique** régulé par des mécanismes de la théorie des jeux. MBCAS introduit une architecture décentralisée où chaque service (Pod) est piloté par un agent autonome intelligent. Grâce à l'apprentissage par renforcement (**Q-Learning**), ces agents apprennent à exprimer leurs besoins réels en "enchérissant" pour des ressources.

L'originalité de notre approche réside dans l'utilisation de la **Solution de Négociation de Nash** pour arbitrer ces demandes. Contrairement aux heuristiques gloutonnes, cet algorithme mathématique garantit une allocation *Pareto-efficace* et équitable, assurant qu'aucun cycle CPU n'est gaspillé tant qu'un besoin existe dans le cluster.

Sur le plan technique, la réalisation de ce projet a nécessité la maîtrise de technologies de pointe. Développé en **Go**, MBCAS s'interface avec les couches basses du noyau Linux via **Cgroups v2** et exploite les fonctionnalités expérimentales de Kubernetes, notamment l'*In-Place Pod Vertical Scaling*, permettant de redimensionner les conteneurs à chaud sans interruption de service.

Ce rapport détaille le cheminement de notre projet, depuis l'analyse théorique des modèles multi-agents jusqu'à l'implémentation d'un opérateur Kubernetes fonctionnel, démontrant ainsi la capacité de l'ingénierie logicielle moderne à résoudre des problèmes d'infrastructure complexes par l'intelligence artificielle.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Contexte et Problématique

1.1 Introduction : L'ère de l'Orchestration et des Microservices

L'industrie du développement logiciel a opéré, au cours de la dernière décennie, une transition majeure des architectures monolithiques vers les architectures orientées microservices. Cette décomposition applicative, bien qu'offrant une agilité et une résilience accrues, a introduit une complexité opérationnelle exponentielle : la nécessité de gérer des milliers de conteneurs distribués sur des grappes de serveurs hétérogènes.

Dans ce paysage, **Kubernetes** s'est imposé comme le standard industriel (*de facto*) pour l'orchestration de conteneurs. Il abstraie la complexité de l'infrastructure sous-jacente en proposant un modèle déclaratif pour le déploiement et la gestion du cycle de vie des applications. Cependant, si Kubernetes excelle dans l'orchestration, la gestion fine et dynamique des ressources computationnelles (CPU et Mémoire) demeure un défi ouvert et critique pour les ingénieurs en fiabilité de site (SRE) et les architectes logiciels.

1.2 Mécanismes d'Allocation de Ressources dans Kubernetes

Au cœur du modèle de ressources de Kubernetes se trouve le planificateur (*Scheduler*), qui prend des décisions de placement des *Pods* (unités atomiques de déploiement) en se basant sur deux directives statiques définies dans les manifestes de déploiement :

- **Requests (Garantie) :** La quantité minimale de ressources (CPU/RAM) garantie à un conteneur. Le planificateur utilise cette valeur pour décider sur quel nœud placer le pod.
- **Limits (Plafond) :** La quantité maximale de ressources qu'un conteneur est autorisé à consommer. Au-delà de ce seuil, le noyau Linux intervient pour restreindre le processus.

Ce modèle statique pose un problème fondamental : il suppose que la consommation de ressources d'une application est prédictible et constante. Or, la réalité des charges de travail modernes est intrinsèquement variable, stochastique et sujette à des pics imprévisibles (*bursty workloads*).

1.3 Stratégies de Mise à l'Échelle (Scaling)

Pour répondre à cette variabilité, Kubernetes propose nativement deux vecteurs d'élasticité : le scaling horizontal et le scaling vertical.

1.3.1 Scaling Horizontal (HPA) - Hors Périmètre

Le *Horizontal Pod Autoscaler* (HPA) répond à la charge en augmentant le nombre de répliques d'un service (*scale-out*). Bien qu'efficace pour les applications sans état (*stateless*), cette approche présente des limites structurelles :

1. **Latence de démarrage :** L'instanciation de nouveaux pods et leur initialisation (Cold Start) prend du temps, rendant le HPA inefficace pour absorber des pics de charge soudains de l'ordre de la milliseconde ou de la seconde.
2. **Complexité pour le Stateful :** Pour les bases de données ou les applications avec état, la multiplication des instances est souvent complexe voire impossible sans une logique de réplication lourde.

Ce projet se concentre donc exclusivement sur l'optimisation des ressources d'un conteneur existant, un domaine où les gains d'efficacité sont les plus prometteurs.

1.3.2 Scaling Vertical (VPA) et ses Limitations

Le *Vertical Pod Autoscaler* (VPA) vise à ajuster dynamiquement les *requests* et *limits* d'un pod existant pour s'adapter à sa consommation réelle (*scale-up/down*). Théoriquement idéal, le VPA souffre dans sa version standard de limitations rédhibitoires pour les environnements de production critiques :

1. La contrainte de disruption (Le problème du redémarrage) Historiquement, la modification des ressources allouées à un conteneur dans Kubernetes est une opération immuable qui nécessite la destruction puis la recréation du pod.

- **Impact :** Cette opération entraîne une interruption de service, la perte des caches en mémoire (RAM warm-up), et perturbe les connexions actives.
- **Conséquence :** Le VPA est rarement utilisé sur des charges de travail sensibles à la latence ou critiques.

2. Réactivité et Granularité Les algorithmes de VPA traditionnels se basent sur des métriques historiques lissées sur plusieurs minutes. Ils manquent de la granularité nécessaire pour réagir aux micro-bursts, laissant les applications subir du *throttling* (bridage CPU) entre deux cycles de recommandation.

1.4 Évolution Technologique : In-Place Pod Vertical Scaling

Une avancée récente dans l'écosystème Kubernetes (introduite progressivement depuis la version 1.27) est la fonctionnalité d'**In-Place Pod Vertical Scaling**. Cette capacité permet de modifier les limites `cgroups` d'un processus en cours d'exécution sans redémarrer le conteneur associé.

C'est sur cette innovation technologique majeure ("Bleeding Edge") que s'appuie notre projet. Elle lève le verrou technique du redémarrage, ouvrant la voie à des stratégies d'allocation en temps réel et à haute fréquence que nous explorons avec MBCAS.

1.5 Problématique : Le Dilemme de l'Allocation Statique

Malgré ces outils, les administrateurs systèmes sont aujourd'hui confrontés à un dilemme binaire et inefficace :

— **Sur-allocation (Over-provisioning) :** Pour garantir la performance et éviter le risque, les ressources sont sur-dimensionnées. Cela conduit à un gaspillage massif de capacité CPU (souvent $> 30\%$ de ressources inactives) et à une facture Cloud inutilement élevée.

— **Sous-allocation (Under-provisioning) :** Pour optimiser les coûts, les limites sont serrées. Au moindre pic de charge, le mécanisme de protection du noyau Linux (CFS Bandwidth Control) s'active, provoquant du *CPU Throttling*. L'application est alors ralentie artificiellement, dégradant l'expérience utilisateur et violant les SLAs.

La problématique centrale de ce projet est donc :

Comment concevoir un système autonome capable d'allouer les ressources CPU de manière dynamique, équitable et sans interruption, afin de maximiser l'utilisation du cluster tout en garantissant les performances individuelles des services ?

1.6 Contribution et Approche Proposée : Une Architecture Hybride

Pour résoudre ce dilemme structurel et pallier les limitations des outils existants, ce projet fédérateur introduit **MBCAS (Market-Based CPU Allocation System)**. Notre solution dépasse le simple réglage paramétrique pour proposer une réponse systémique fondée sur la convergence de deux disciplines :

1. **Réponse à la Rigidité (Modélisation Multi-Agents) :** Afin de s'affranchir d'une planification centralisée souvent latente, nous décentralisons la prise de décision. Chaque Pod est transformé en un agent autonome (*PodAgent*) capable d'apprendre ses besoins réels en temps réel. Grâce à l'apprentissage par renforcement (**Q-Learning**), l'agent anticipe les variations de charge et formule des demandes précises, éliminant ainsi le besoin de définir statiquement des *requests* et *limits* arbitraires.
2. **Réponse à la Congestion (Théorie des Jeux Coopératifs) :** Pour gérer les situations de pénurie où la demande excède la capacité du nœud, MBCAS substitue la compétition chaotique par une coopération rationnelle. Nous implémentons la **Solution de Négociation de Nash**, un mécanisme mathématique qui garantit une allocation *Pareto-Efficace*. Cela assure mathématiquement qu'aucune ressource n'est gaspillée (résolvant la sur-allocation) et que la pénurie est partagée équitablement (minimisant l'impact du throttling).

En couplant cette intelligence décisionnelle avec la capacité d'*In-Place Scaling*, MBCAS offre une alternative viable, capable d'adapter l'infrastructure à la charge instantanée sans interruption de service.

Chapitre 2

Fondements Théoriques : Théorie des Jeux et Intelligence Artificielle

2.1 Introduction

La résolution du problème d'allocation de ressources dans les systèmes distribués nécessite de modéliser des interactions complexes entre de multiples entités. Pour ce faire, ce projet s'appuie sur deux piliers théoriques majeurs : la Théorie des Jeux, qui fournit le cadre mathématique pour analyser les décisions stratégiques, et l'Intelligence Artificielle, spécifiquement l'Apprentissage par Renforcement, qui confère aux entités la capacité d'adaptation.

2.2 Introduction Générale à la Théorie des Jeux

La Théorie des Jeux est une branche des mathématiques appliquées et de l'économie qui étudie les situations où le succès d'un individu dépend non seulement de ses propres choix, mais aussi de ceux des autres participants.

2.2.1 Concepts Fondamentaux

Un jeu est formellement défini par un triplet (N, S, U) où :
- $N = \{1, ..., n\}$ est l'ensemble fini des **joueurs** (agents décideurs).
- S_i est l'ensemble des **stratégies** disponibles pour le joueur i. L'espace des stratégies du jeu est $S = S_1 \times ... \times S_n$.
- $U_i : S \to R$ est la **fonction d'utilité** (ou *payoff*) du joueur i, qui quantifie son gain pour chaque issue possible du jeu.

L'hypothèse centrale est la **rationalité** : chaque joueur cherche à maximiser son espérance d'utilité, en anticipant que les autres joueurs feront de même.

2.2.2 Taxonomie des Jeux

Il existe plusieurs distinctions cruciales pour classifier les jeux :

Jeux Coopératifs vs Non-Coopératifs

- **Non-Coopératifs :** Les joueurs ne peuvent pas former d'alliances contraignantes. C'est le domaine de l'*Équilibre de Nash* classique, où l'on étudie la compétition pure.
- **Coopératifs :** Les joueurs peuvent communiquer et former des coalitions pour maximiser un gain commun, qu'ils se partagent ensuite. C'est ce cadre qui est pertinent pour l'allocation de ressources partagées (comme un CPU).

Jeux à Somme Nulle vs Somme Non-Nulle

- **Somme Nulle :** Le gain d'un joueur est exactement égal à la perte de l'autre ($\sum U_i = 0$). C'est un conflit total.
- **Somme Non-Nulle :** Il existe des issues où tous les joueurs peuvent gagner (ou perdre) simultanément. L'allocation de ressources est typiquement à somme non-nulle : une bonne allocation améliore la performance globale du cluster (tout le monde gagne).

2.2.3 Concepts d'Équilibre et d'Efficacité

Il est impératif de distinguer deux concepts souvent confondus, bien que tous deux développés par John Nash, car ils appartiennent à deux branches distinctes de la théorie des jeux.

L'Équilibre de Nash (Jeux Non-Coopératifs)

L'**Équilibre de Nash** (1951) est le concept central des jeux *non-coopératifs*. Il décrit une situation de compétition pure où les agents ne peuvent ni communiquer ni prendre d'engagements contraignants.

- **Définition :** Un ensemble de stratégies est un Équilibre de Nash si aucun joueur n'a intérêt à changer unilatéralement sa stratégie, les stratégies des autres étant fixées. C'est un état de stabilité stratégique "figé".
- **Limitation (Le "Prix de l'Anarchie") :** L'Équilibre de Nash est souvent *sous-optimal* pour le groupe. L'exemple célèbre est le *Dilemme du Prisonnier* : les deux joueurs choisissent rationnellement de se trahir (équilibre de Nash), alors qu'ils auraient tous deux gagné à coopérer.
- **Dans le contexte Kubernetes :** Si nous utilisions cette approche, chaque Pod tenterait égoïstement de consommer le maximum de CPU pour saturer le nœud avant les autres. Cela mènerait à une "tragédie des communs" où le cluster serait instable et inefficace.

La Solution de Négociation de Nash (Jeux Coopératifs)

La **Solution de Négociation de Nash** (1950), ou *Nash Bargaining Solution (NBS)*, relève des jeux *coopératifs*. Ici, on suppose que les agents peuvent s'accorder sur une répartition conjointe du surplus.

- **Définition :** C'est une solution axiomatique qui détermine comment partager un gain commun de manière unique, satisfaisante et juste. Elle ne cherche pas un point de stabilité stratégique, mais un point d'*optimisation sociale*.
- **Avantage (Efficacité de Pareto) :** Contrairement à l'Équilibre de Nash, la Solution de Négociation garantit l'efficacité de Pareto. Il n'y a aucun gaspillage : tout le CPU disponible est alloué.
- **Dans le contexte Kubernetes (Projet MBCAS) :** C'est l'approche retenue. Les Pods "s'accordent" (via l'algorithme central) pour se partager le CPU de manière à maximiser le produit de leurs utilités. Cela assure que les ressources sont utilisées au maximum tout en respectant une équité proportionnelle.

Tableau Comparatif

Critère	Équilibre de Nash	Négociation de Nash (NBS)
Branche	Jeux Non-Coopératifs (Compétition)	Jeux Coopératifs (Collaboration)
Objectif	Stabilité individuelle (Best Response)	Équité et Efficacité collective
Résultat	Souvent inefficace (Gaspillage possible)	Toujours Pareto-Efficace (Zéro gaspillage)
Mécanisme	Chaque agent agit seul contre les autres	Un arbitre (mathématique) répartit le gain
Usage MBCAS	**Rejeté** (Risque d'instabilité)	**Adopté** (Garantie de performance)

TABLE 2.1 – Comparaison entre Équilibre de Nash et Négociation de Nash

L'Efficacité de Pareto

Une allocation de ressources x est dite **Pareto-Efficace** (ou Pareto-Optimale) s'il n'existe aucune autre allocation y telle que :

$$\forall i, U_i(y) \geq U_i(x) \quad \text{et} \quad \exists j, U_j(y) > U_j(x) \tag{2.1}$$

En termes simples : on ne peut pas améliorer la situation de quelqu'un sans détériorer celle de quelqu'un d'autre. C'est le "Saint Graal" de l'ingénierie système, signifiant qu'il n'y a **aucun gaspillage**.

2.3 Application au Projet : Théorie des Jeux Coopératifs

Pour le projet MBCAS, nous avons rejeté l'approche non-coopérative (qui mènerait à une guerre des ressources et à l'instabilité) au profit de l'approche coopérative. Le problème est modélisé comme un **Problème de Négociation** (*Bargaining Problem*).

2.3.1 Le Problème de Négociation

Soit un ensemble de Pods N se disputant une capacité CPU totale C. Le problème est défini par la paire (F, d) :

- $F \subset R^n$ est l'ensemble des allocations faisables ($\sum x_i \leq C$).
- $d = (d_1, ..., d_n)$ est le **point de désaccord** (*disagreement point*), correspondant au minimum vital garanti à chaque Pod (ex : 100m CPU). Si la négociation échoue, chaque Pod i reçoit d_i.

2.3.2 La Solution de Négociation de Nash (NBS)

Parmi toutes les allocations Pareto-Efficaces possibles, laquelle est la plus "juste" ? John Nash (1950) a proposé une solution axiomatique unique qui satisfait :

1. **Invariance aux transformations affines** (Indépendance des unités de mesure).
2. **Efficacité de Pareto** (Pas de gaspillage).
3. **Indépendance des alternatives non pertinentes (IIA)**.
4. **Symétrie** (À besoins égaux, allocations égales).

Cette solution est obtenue en maximisant le **Produit de Nash** :

$$\text{Maximize} \prod_{i=1}^{n}(x_i - d_i) \quad \text{sous contrainte } x \in F, x_i \geq d_i \tag{2.2}$$

2.3.3 Généralisation : La Solution de Nash Pondérée

Dans un cluster Kubernetes, tous les Pods ne sont pas égaux (ex : un service de paiement est plus critique qu'un job de log). Nous utilisons donc la forme asymétrique (pondérée) de la NBS :

$$\max_{x} \prod_{i=1}^{n}(x_i - d_i)^{w_i} \tag{2.3}$$

Où w_i est le poids de priorité du Pod i. La solution analytique pour l'allocation x_i de chaque agent est donnée par :

$$x_i = d_i + \frac{w_i}{\sum_{j=1}^{n} w_j} \times (C - \sum_{j=1}^{n} d_j) \tag{2.4}$$

Cette formule, implémentée dans notre algorithme, garantit que chaque Pod reçoit son minimum vital d_i, et que le surplus de capacité $(C - \sum d_j)$ est distribué proportionnellement à l'importance w_i.

2.4 Modélisation Basée sur les Agents (ABM)

Pour mettre en œuvre cette théorie, nous adoptons une architecture décentralisée.

2.4.1 Le Concept d'Agent

Un Système Multi-Agents (SMA) est composé d'entités autonomes situées dans un environnement. Dans MBCAS, chaque Pod est piloté par un **PodAgent** qui possède :

— Un **État interne** : Sa connaissance actuelle de la charge.
— Des **Capteurs** : Pour lire les métriques Cgroups v2 (usage, throttling).
— Des **Effecteurs** : Pour soumettre des enchères au mécanisme de Nash.

2.5 Apprentissage par Renforcement : Q-Learning

L'agent doit déterminer quelle quantité de ressource demander (d_i) et avec quelle urgence (w_i). Comme la charge de travail est stochastique, il utilise le **Q-Learning** pour apprendre la politique optimale $\pi : S \to A$.

2.5.1 L'Équation de Bellman

L'agent maintient une table $Q(s, a)$ représentant la qualité d'une action a dans l'état s. La mise à jour suit l'équation de Bellman temporelle :

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{2.5}$$

Avec les hyperparamètres du projet : $\alpha = 0.1$ (apprentissage) et $\gamma = 0.9$ (vision long terme).

2.5.2 Définition du MDP (Processus de Décision Markovien)

Espace d'États (27 États)

L'état s_t est un vecteur discret $(Usage, Throttling, Allocation)$:
— $Usage \in \{Low, Medium, High\}$
— $Throttling \in \{None, Some, High\}$
— $Allocation \in \{Low, Adequate, Excess\}$

Fonction de Récompense (R)

La fonction de récompense guide l'apprentissage. Elle est définie mathématiquement comme suit dans notre implémentation :

$$R(s, a) = \begin{cases} +10.0 & \text{si } Alloc \geq Usage \text{ (Demande satisfaite)} \\ -30.0 \times \tau & \text{où } \tau \text{ est le ratio de throttling (Pénalité forte)} \\ -100.0 & \text{si } Latence > Target_{SLO} \text{ (Violation critique)} \\ -5.0 \times \delta & \text{où } \delta \text{ est le surplus inutile (Pénalité de gaspillage)} \end{cases} \tag{2.6}$$

Cette structure de récompense force l'agent à trouver un équilibre subtil : demander assez pour éviter le throttling (pénalité -30), mais pas trop pour éviter le gaspillage (pénalité -5).

Chapitre 3

Analyse et Conception du Système MB-CAS

3.1 Introduction

Après avoir établi les fondements théoriques basés sur les systèmes multi-agents et la théorie des jeux, ce chapitre détaille la concrétisation de ces concepts en une architecture logicielle robuste. La conception de **MBCAS** (Market-Based CPU Allocation System) suit une approche modulaire et distribuée, nativement intégrée à l'écosystème Kubernetes via le pattern *Operator*.

Nous présenterons d'abord l'architecture globale, puis la décomposition détaillée des composants (Agent et Contrôleur), la modélisation des données (CRD), et enfin la conception des processus d'intelligence artificielle.

3.2 Architecture Globale : Une Approche Hybride

Pour répondre aux exigences de scalabilité et de résilience, MBCAS adopte une architecture décentralisée qui sépare le plan de décision (Intelligence) du plan de contrôle (Exécution).

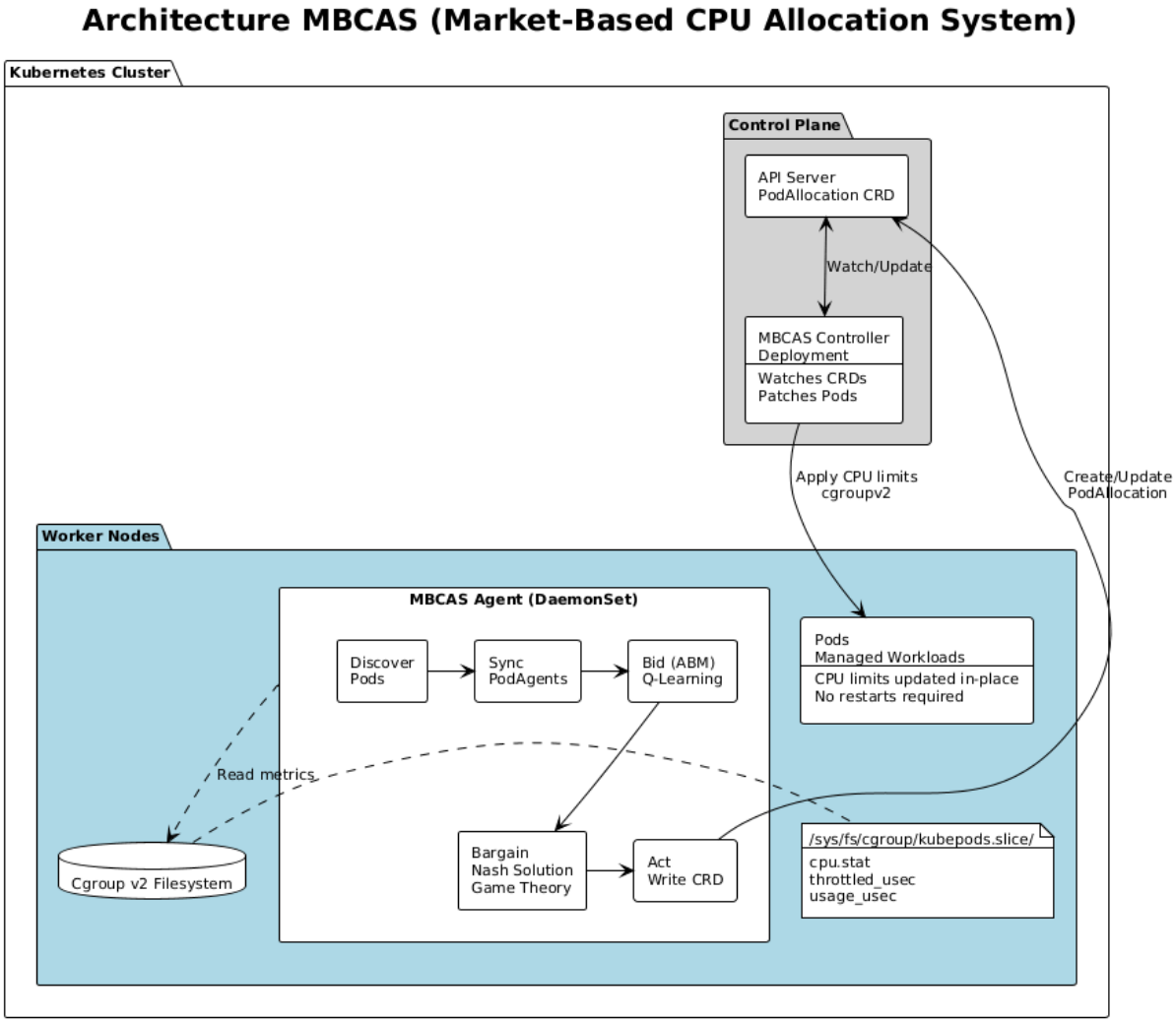

FIGURE 3.1 – Architecture Distribuée MBCAS : Interaction Agent-Contrôleur

Comme illustré dans la Figure 3.1, le système repose sur deux composants majeurs communiquant de manière asynchrone :

3.2.1 Le MBCAS Agent (Plan de Données & Intelligence)

Déployé sous forme de *DaemonSet*, cet agent est le cerveau local présent sur chaque nœud.

- **Rôle :** Il interface directement avec le noyau Linux via `cgroup v2` pour extraire des métriques haute fréquence (usage, throttling). Il héberge les modèles Q-Learning individuels des Pods et résout les conflits locaux via l'algorithme de Nash.
- **Autonomie :** En cas de coupure réseau avec le plan de contrôle, l'agent continue d'optimiser les ressources localement, garantissant la robustesse du nœud.

3.2.2 Le MBCAS Controller (Plan de Contrôle)

Ce composant centralisé (Singleton Deployment) assure la cohérence globale.

— **Rôle :** Il surveille les décisions émises par les agents (via les CRD `PodAllocation`) et orchestre leur application.
— **Mécanisme Critique :** Il exploite l'API *In-Place Pod Vertical Scaling* pour modifier les limites de ressources à chaud, sans redémarrer les conteneurs.

3.3 Le Pipeline Décisionnel MBCAS

Le cœur du système est une boucle de contrôle stricte (*Control Loop*) exécutée par l'agent. Ce pipeline transforme les données brutes en actions d'allocation. La Figure 3.2 détaille les cinq étapes séquentielles du processus, cadencées par un intervalle de 5 secondes :

3.3.1 Étape 1 : Discover (Découverte)

L'agent scanne le nœud pour identifier les charges de travail. Un filtrage intelligent est appliqué pour ignorer les Pods systèmes (namespace `kube-system`) et ne cibler que les applications annotées `mbcas.io/managed`, évitant ainsi de perturber les composants critiques de l'infrastructure.

3.3.2 Étape 2 : Sync (Synchronisation)

Cette étape de maintenance aligne les modèles d'IA avec la réalité. Les agents Q-Learning des Pods terminés sont détruits pour libérer la mémoire, tandis que les nouveaux Pods se voient instancier un agent vierge, prêt à apprendre.

3.3.3 Étape 3 : Bid (Enchère Intelligente)

C'est l'étape d'intelligence artificielle locale. Chaque *PodAgent* analyse son état (Usage vs Throttling) et consulte sa Q-Table. Il formule ensuite une enchère (*Bid*) composée de sa demande minimale (survie) et de sa demande idéale, pondérée par l'urgence de la situation (ex : forte pondération si violation de SLO imminente).

3.3.4 Étape 4 : Bargain (Négociation de Nash)

Le solveur central du nœud collecte toutes les enchères. Il exécute l'algorithme de la *Solution de Négociation de Nash* pour répartir la capacité CPU disponible. Cette méthode mathématique garantit qu'aucune ressource n'est gaspillée (Efficacité de Pareto) et que la pénurie est partagée équitablement selon les poids des enchères.

3.3.5 Étape 5 : Act (Action et Persistance)

Les résultats de l'optimisation sont écrits dans l'API Kubernetes sous forme d'objets `PodAllocation`. C'est le point de découplage où l'Agent passe le relais au Contrôleur pour l'application effective.

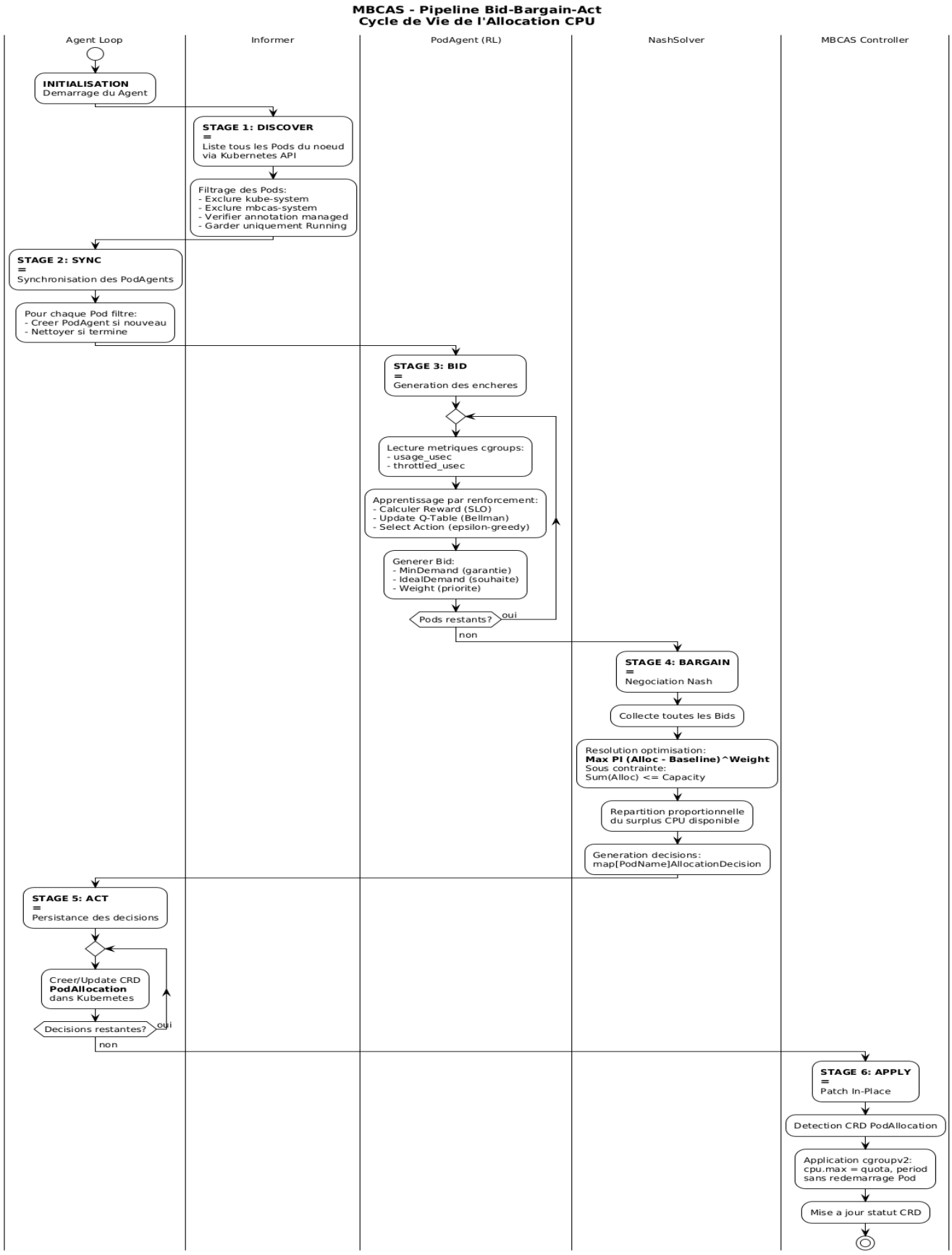

FIGURE 3.2 – Cycle de Vie de l'Allocation : Le Pipeline Bid-Bargain-Act

3.4 Modélisation des Données (CRD)

L'interface entre les composants est strictement typée via la ressource **PodAllocation**. Elle encapsule non seulement la décision (Request/Limit) mais aussi le contexte décisionnel (Poids, Priorité) pour permettre l'auditabilité du système IA.

La Figure 3.3 illustre le modèle de données implémenté :

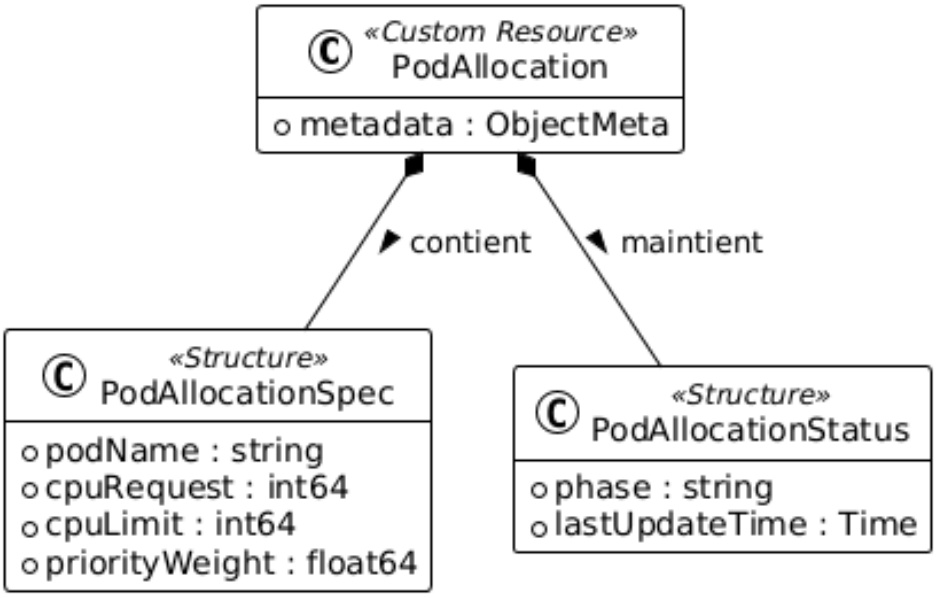

FIGURE 3.3 – Diagramme de Classes UML de la Ressource PodAllocation

Comme modélisé ci-dessus, la structure se compose de deux parties essentielles :

— **Spec (Intention) :** Contient les directives d'allocation calculées par l'agent (ex : `CPURequest`, `CPULimit`) ainsi que le contexte de décision IA (`PriorityWeight`).

— **Status (Observation) :** Permet au contrôleur de signaler l'avancement de l'application des changements (ex : `Phase: Applied`).

3.5 Conclusion

Cette conception par pipeline garantit que chaque décision d'allocation est le fruit d'un processus rigoureux : observation précise, apprentissage individuel, et négociation collective. L'architecture découplée assure que cette complexité n'impacte pas la stabilité du cluster hôte.

Chapitre 4

Implémentation du système

Introduction

Ce chapitre présente la réalisation concrète du système **MBCAS (Market-Based CPU Allocation System)**. Après avoir détaillé les fondements théoriques et la conception architecturale, l'objectif est de décrire comment ces concepts ont été traduits en une solution logicielle fonctionnelle, intégrée nativement à l'écosystème Kubernetes.

L'implémentation repose sur une architecture *cloud-native* combinant un opérateur Kubernetes, des agents autonomes basés sur l'apprentissage par renforcement et un mécanisme de négociation coopérative issu de la théorie des jeux.

4.1 Choix technologiques et environnement

4.1.1 Langage et outils

Le système MBCAS est développé en **Go (Golang)**, un langage particulièrement adapté au développement de systèmes distribués et d'opérateurs Kubernetes.

FIGURE 4.1 – Langage Go (Golang)

Avantages du langage Go

[leftmargin=*]**Support natif de la concurrence** : goroutines et channels permettent une gestion efficace des tâches parallèles **Bibliothèques officielles Kubernetes** : intégration native via *client-go* et *controller-runtime* **Performances élevées** : exécution rapide avec une faible empreinte mémoire

4.1.2 Intégration Kubernetes

L'implémentation adopte le *pattern Operator*, permettant d'étendre Kubernetes à l'aide de ressources personnalisées (*Custom Resource Definitions – CRD*) tout en respectant son modèle déclaratif.

FIGURE 4.2 – Plateforme Kubernetes

Composant	Description	Type de déploiement
MBCAS Agent	Agent autonome responsable de la prise de décision locale sur chaque nœud	*DaemonSet*
MBCAS Controller	Contrôleur centralisé pour la gestion globale du système	*Deployment* (singleton)

TABLE 4.1 – Composants principaux du système MBCAS

4.2 Implémentation de l'Agent MBCAS

4.2.1 Vue générale de l'agent

L'Agent MBCAS constitue le composant central du système au niveau du nœud Kubernetes. Il est responsable de la prise de décision locale concernant l'allocation dynamique du CPU pour l'ensemble des pods exécutés sur ce nœud.

Caractéristiques clés de l'agent

[leftmargin=*]**Déploiement décentralisé** : un agent par nœud via *DaemonSet* **Autonomie complète** : prise de décision locale sans coordination inter-agents **Communication API-only** : interactions exclusives via l'API Kubernetes **Scalabilité optimale** : latence réduite grâce à la décentralisation

4.2.2 Architecture interne de l'agent

L'agent MBCAS est implémenté comme un service Go modulaire, structuré autour d'une architecture en couches facilitant la maintenabilité et l'extensibilité.

Module	Responsabilité
Module de découverte	Identification dynamique des pods présents sur le nœud
Module d'observation	Collecte des métriques CPU bas niveau via Cgroups v2
Gestionnaire de PodAgents	Maintien du cycle de vie des agents autonomes associés à chaque pod
Moteur de décision	Implémentation de la logique d'apprentissage par renforcement et génération des enchères
Solveur de négociation	Application de la Solution de Négociation de Nash pour le calcul des allocations finales
Module de publication	Persistance des décisions sous forme de ressources *PodAllocation*

TABLE 4.2 – Architecture modulaire de l'agent MBCAS

4.2.3 Découverte et synchronisation des pods

La découverte des pods est réalisée à l'aide des *informers Kubernetes*, permettant une surveillance efficace et événementielle des changements d'état.

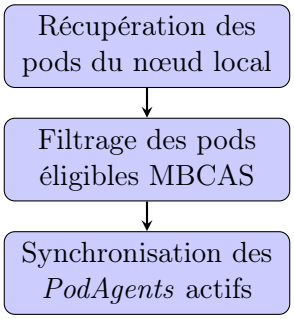

FIGURE 4.3 – Cycle de découverte et synchronisation des pods

Gestion du cycle de vie

Nouveau pod détecté : création et initialisation d'une instance *PodAgent*
Pod supprimé : destruction de l'agent associé pour éviter les fuites de ressources

4.2.4 Collecte des métriques via Cgroups v2

Contrairement aux métriques agrégées fournies par Kubernetes, MBCAS exploite des métriques bas niveau issues de **Cgroups v2**. L'agent lit directement les fichiers du système

de fichiers virtuel `/sys/fs/cgroup/`.

Métrique brute	Fichier Cgroups	Description
Temps CPU total	`usage_usec`	Temps CPU consommé par le pod (en microsecondes)
Temps de throttling	`throttled_usec`	Durée pendant laquelle le pod a été bridé

TABLE 4.3 – Métriques Cgroups v2 collectées

Métrique calculée	Utilité
Taux d'utilisation CPU	Évaluation de la charge actuelle du pod
Ratio de throttling	Détection de situations de contention CPU
Évolution de la charge	Tendance de consommation entre cycles consécutifs

TABLE 4.4 – Métriques dérivées pour la prise de décision

Avantage de l'approche bas niveau

Cette approche permet une **détection fine et réactive** des situations de contention CPU, souvent invisibles aux mécanismes standards d'auto-scaling de Kubernetes.

4.2.5 Gestion des PodAgents

Chaque pod est associé à un *PodAgent*, représentant une entité autonome d'apprentissage et de décision. Le gestionnaire de PodAgents maintient une correspondance un-à-un entre pods et agents.

Élément persisté	Description
État courant discret	Représentation de l'état du pod dans l'espace d'états
Table Q	Valeurs d'utilité pour chaque paire état-action
Paramètres d'apprentissage	Taux d'apprentissage α, facteur d'exploration ϵ
Historique d'observations	Données récentes pour l'analyse de tendances

TABLE 4.5 – Données persistées par chaque PodAgent

Apprentissage par expérience

Cette persistance locale permet à l'agent d'exploiter l'expérience passée afin d'améliorer progressivement la qualité des décisions.

4.2.6 Génération des enchères (Bid Phase)

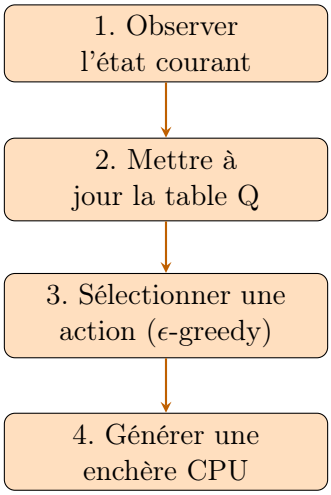

FIGURE 4.4 – Processus de génération des enchères

Composante de l'enchère	Signification
Demande minimale CPU	Quantité minimale nécessaire au fonctionnement du pod
Demande idéale CPU	Quantité optimale pour des performances maximales
Poids de priorité	Urgence du besoin en ressources (reflète l'état du pod)

TABLE 4.6 – Structure d'une enchère CPU

4.2.7 Résolution de la négociation (Bargain Phase)

Une fois toutes les enchères collectées, l'agent exécute le solveur de négociation de Nash pour répartir la capacité CPU disponible sur le nœud.

Propriétés de la Solution de Nash

[leftmargin=*]**Équité** : aucun pod n'est systématiquement favorisé **Proportionnalité aux priorités** : respect des poids de priorité **Efficacité de Pareto** : aucune allocation alternative ne peut améliorer un pod sans en pénaliser un autre

Gestion de la surcharge

En cas de surcharge extrême, un mécanisme de **dégradation contrôlée** est activé afin de réduire proportionnellement les allocations minimales et préserver la stabilité du système.

4.2.8 Publication des décisions (Act Phase)

À l'issue de la négociation, l'agent publie les résultats sous forme de ressources *PodAllocation* dans l'API Kubernetes.

Champ PodAllocation	Contenu
Identifiant du pod	Nom et namespace du pod cible
CPU *request*	Valeur minimale garantie calculée
CPU *limit*	Valeur maximale autorisée calculée
Informations de priorité	Poids et classe de priorité utilisés

TABLE 4.7 – Contenu d'une ressource PodAllocation

Application des allocations

Ces ressources sont prises en charge par le contrôleur Kubernetes, qui assure l'application effective des nouvelles allocations via l'API Kubernetes.

4.2.9 Tolérance aux pannes et robustesse

Scénario de panne	Mécanisme de résilience
Redémarrage de l'agent	Aucun impact sur les pods en cours d'exécution
Perte d'état temporaire	Reprise de l'état via les CRD persistées
Erreur de lecture de métriques	Traitement en mode dégradé avec valeurs par défaut
Défaillance du nœud	Redéploiement automatique de l'agent via DaemonSet

TABLE 4.8 – Mécanismes de tolérance aux pannes

Garantie de robustesse

Cette conception assure un fonctionnement robuste dans des environnements Kubernetes dynamiques et potentiellement instables, typiques des infrastructures de production.

4.2.10 Déploiement et validation du système

Après l'implémentation des différents composants, le système MBCAS a été déployé sur un cluster Kubernetes de test afin de valider son bon fonctionnement. Les commandes `kubectl` ont été utilisées pour vérifier l'état des composants et l'application effective des allocations CPU.

```
bcas-system   mbcas-agent-9swrz                  1/1   Running   0   47m
bcas-system   mbcas-controller-76d674bc6d-vpxj8  1/1   Running   0   47m
```

FIGURE 4.5 – Déploiement des composants MBCAS dans le namespace *mbcas-system*

La Figure 4.5 montre le déploiement réussi des composants principaux du système MBCAS. Le contrôleur MBCAS est exécuté sous forme de *Deployment*, tandis que l'agent MBCAS est déployé sous forme de *DaemonSet*, garantissant la présence d'un agent par nœud du cluster. Tous les pods sont dans l'état *Running*, confirmant le bon démarrage du système.

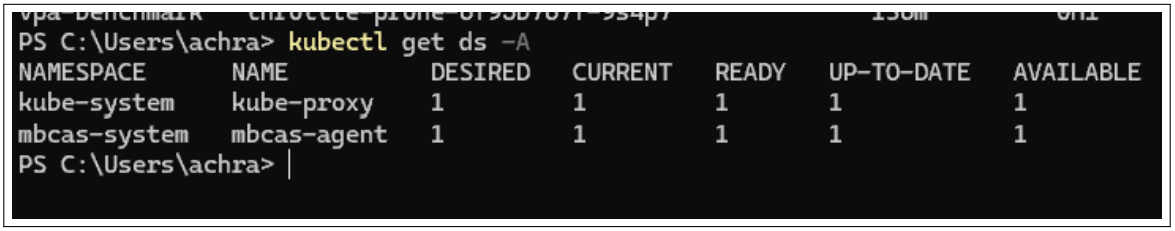

FIGURE 4.6 – État du DaemonSet MBCAS Agent

La Figure 4.6 illustre l'état du *DaemonSet* associé à l'agent MBCAS. Le nombre de pods désirés, disponibles et prêts est identique, ce qui confirme que l'agent est correctement déployé sur l'ensemble des nœuds du cluster. Cette configuration permet une prise de décision locale et décentralisée.

FIGURE 4.7 – Ressources PodAllocation générées par MBCAS

La Figure 4.7 présente les ressources *PodAllocation* créées automatiquement par l'agent MBCAS. Chaque ressource correspond à un pod géré par le système et contient les valeurs de CPU *request* et *limit* calculées à partir des enchères et de la négociation de Nash.

Validation réussie

L'état *Applied* confirme que les décisions ont été correctement prises en compte par le contrôleur Kubernetes et appliquées via la fonctionnalité *In-Place Pod Vertical Scaling*, **sans redémarrage des pods**.

4.3 Évaluation expérimentale et comparaison avec le VPA

4.3.1 Objectifs de l'évaluation

L'objectif principal de cette évaluation est de comparer le système proposé **MBCAS** avec le mécanisme standard d'auto-scaling vertical de Kubernetes, à savoir le **Vertical Pod Autoscaler (VPA)**. La comparaison vise à analyser :

[leftmargin=*]l'efficacité de l'allocation CPU, la capacité à limiter le *throttling*, la stabilité des allocations, la réactivité face aux variations de charge, le coût global en ressources allouées.

L'hypothèse centrale est que MBCAS permet une utilisation plus efficiente du CPU tout en maintenant des performances applicatives comparables, voire supérieures, à celles du VPA.

4.3.2 Scénarios de charge expérimentaux

Afin de couvrir un large éventail de comportements applicatifs, plusieurs profils de charge ont été définis et exécutés sur un cluster Kubernetes de test.

Scénario	Description
Idle	Charge quasi nulle, pod majoritairement inactif
Steady	Charge constante et stable
Bursty	Alternance rapide entre phases calmes et pics
Spiky	Pics CPU très courts et intenses
Ramping	Augmentation progressive de la charge
Overprovisioned	Ressources largement supérieures aux besoins réels

TABLE 4.9 – Scénarios de charge utilisés pour l'évaluation

4.3.3 Métriques de comparaison

Les métriques suivantes ont été collectées à partir des agents MBCAS, des Cgroups v2 et des recommandations VPA :

[leftmargin=*]**CPU alloué** : somme des *requests* et *limits* **Utilisation CPU effective** **Ratio de throttling** **Stabilité des allocations** (variations successives) **Latence de réaction aux changements de charge**

Ces métriques permettent une comparaison à la fois quantitative et qualitative des deux approches.

4.3.4 Comparaison de l'utilisation CPU

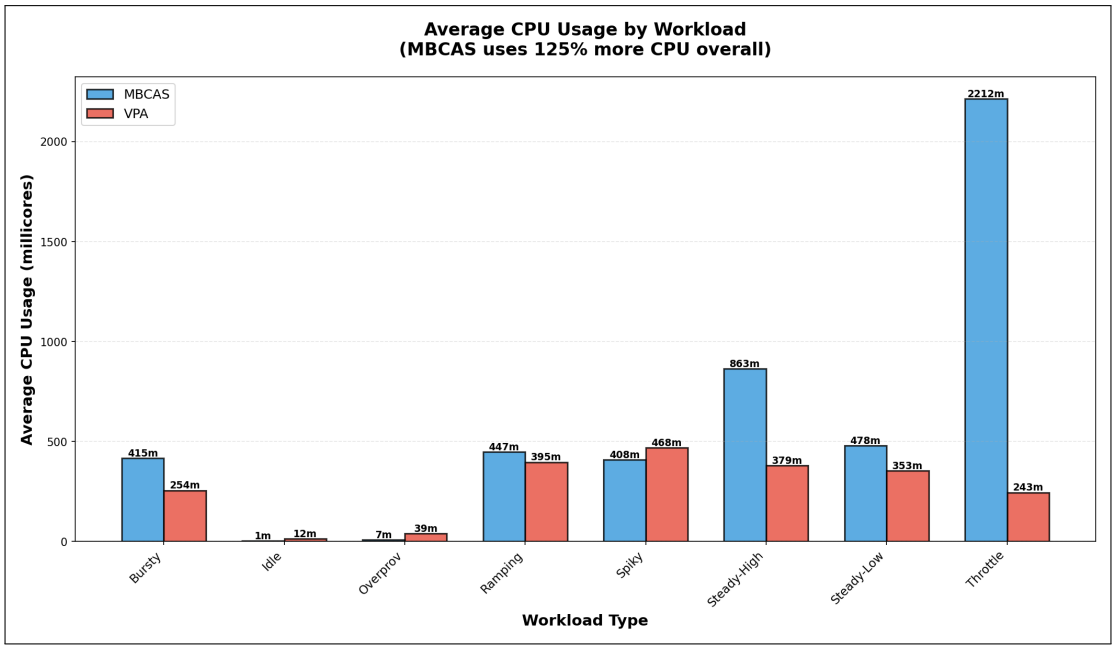

FIGURE 4.8 – Utilisation CPU effective : MBCAS vs VPA

La Figure 4.8 montre que MBCAS ajuste plus finement les allocations CPU à la consommation réelle. Contrairement au VPA, qui tend à sur-allouer afin de se prémunir contre les pics, MBCAS adopte une stratégie adaptative basée sur l'observation directe du throttling et de l'évolution de la charge.

4.3.5 Efficacité et stabilité des allocations

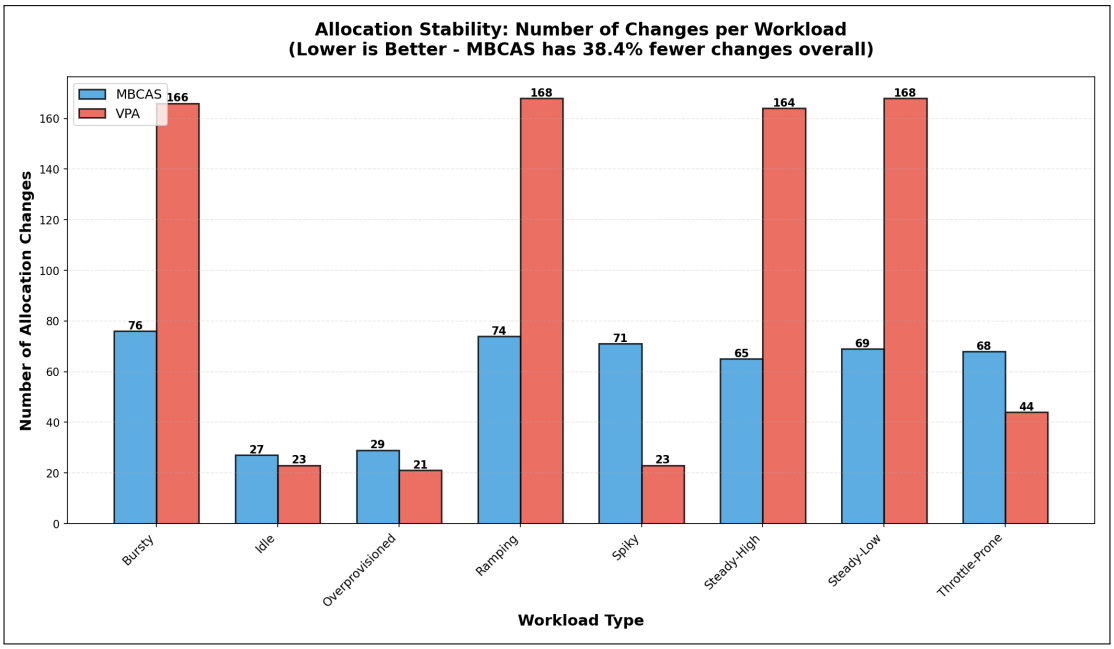

FIGURE 4.9 – Stabilité des allocations CPU

La Figure 4.9 met en évidence une plus grande stabilité des allocations avec MBCAS. Le VPA introduit des variations plus abruptes, souvent liées à des estimations statistiques retardées, tandis que MBCAS privilégie des ajustements progressifs guidés par l'apprentissage.

4.3.6 Gestion du throttling CPU

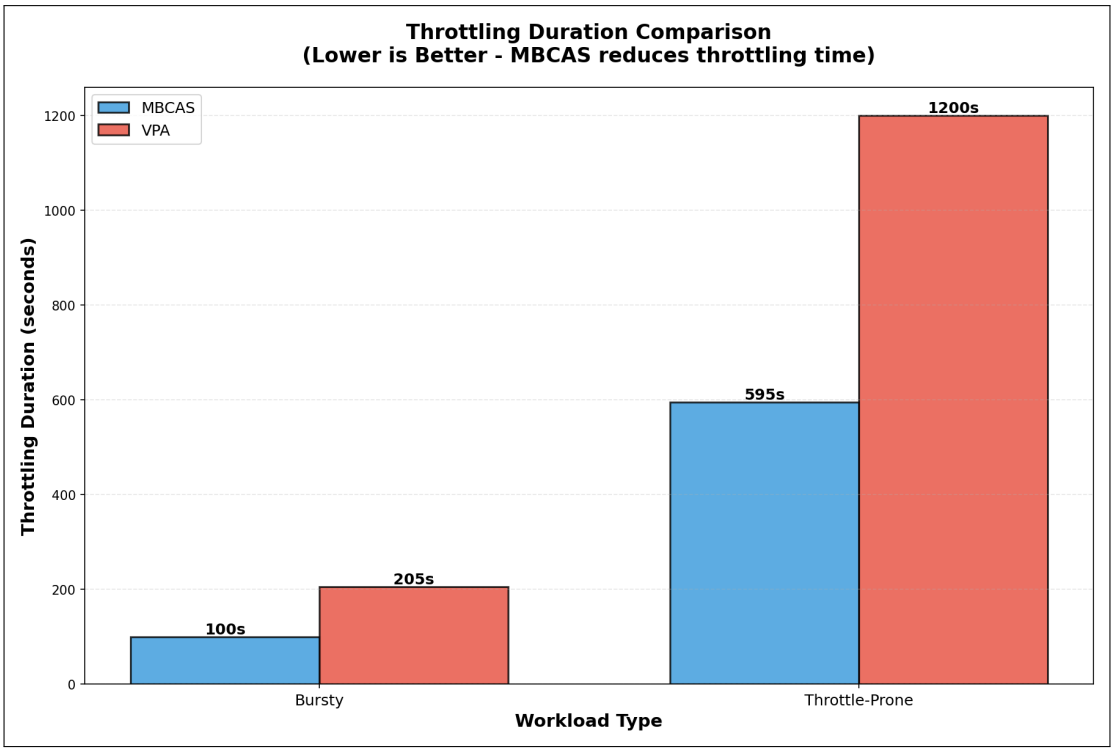

FIGURE 4.10 – Durée et fréquence du throttling CPU

La Figure 4.10 montre que MBCAS réduit significativement les périodes de throttling prolongé. Lorsque des situations de contention apparaissent, le système réagit rapidement en réajustant les allocations, contrairement au VPA dont les recommandations sont recalculées sur des fenêtres temporelles plus larges.

4.3.7 Comportement face aux charges dynamiques

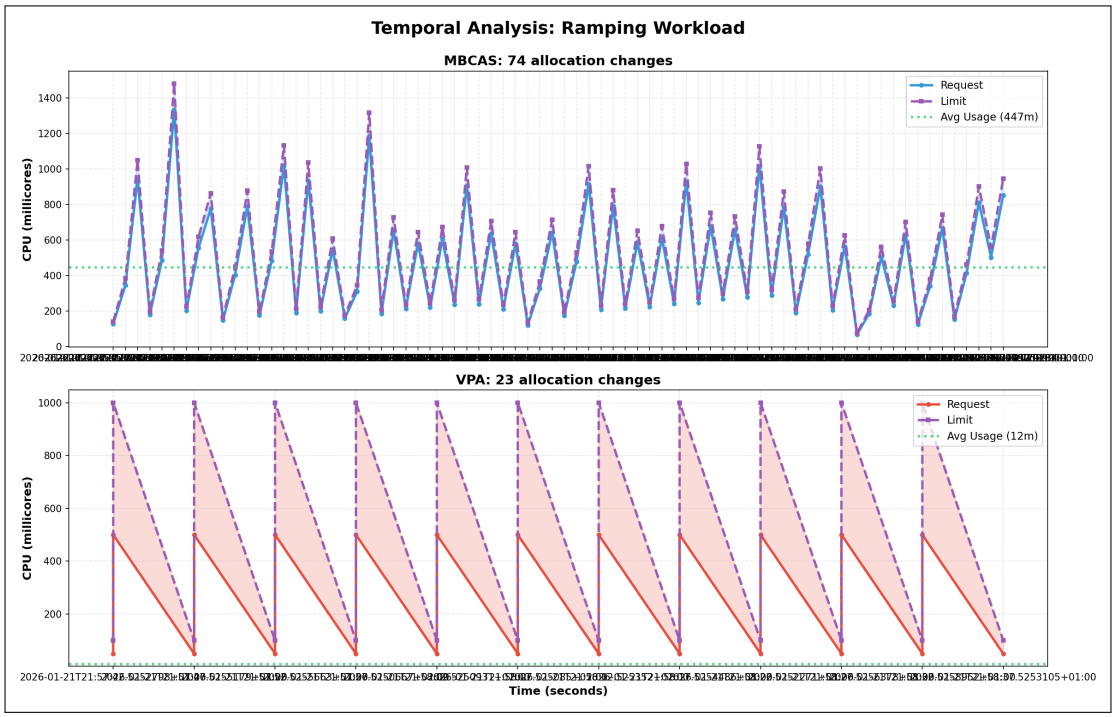

FIGURE 4.11 – Adaptation à une charge de type ramping

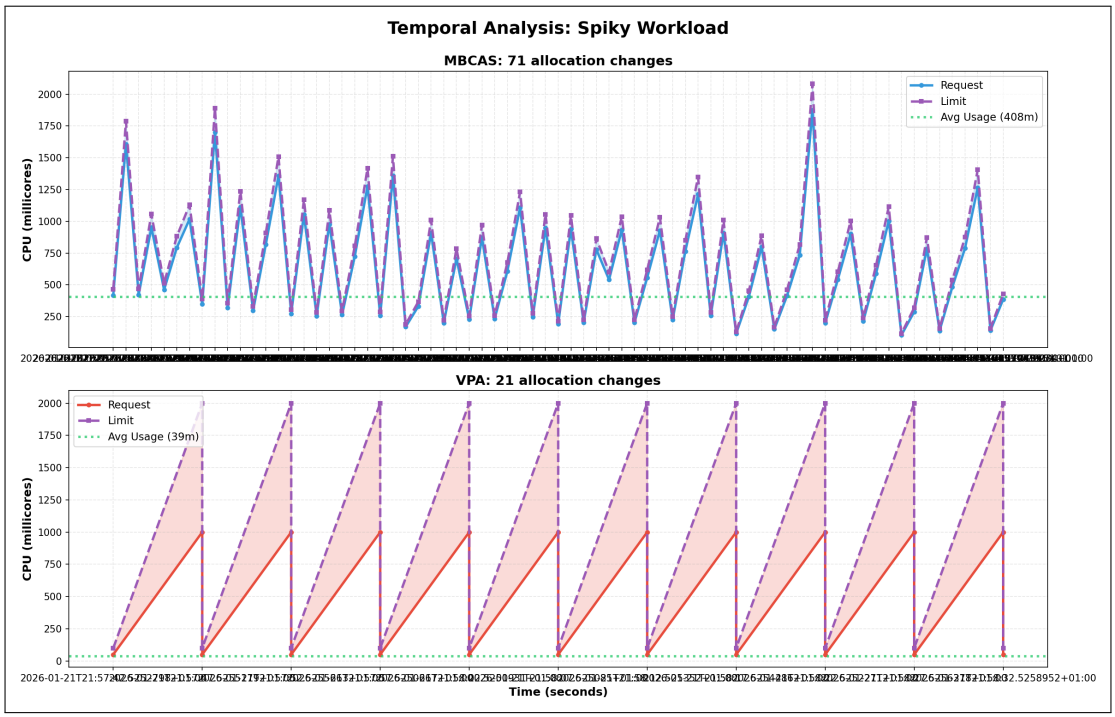

FIGURE 4.12 – Comportement face à des pics CPU courts et intenses

Ces figures illustrent la capacité de MBCAS à anticiper et amortir les variations rapides de charge. Le VPA, conçu pour des tendances à moyen terme, peine à suivre efficacement

ces dynamiques, ce qui se traduit soit par du throttling, soit par une sur-allocation durable.

4.3.8 Sur-allocation et efficacité globale

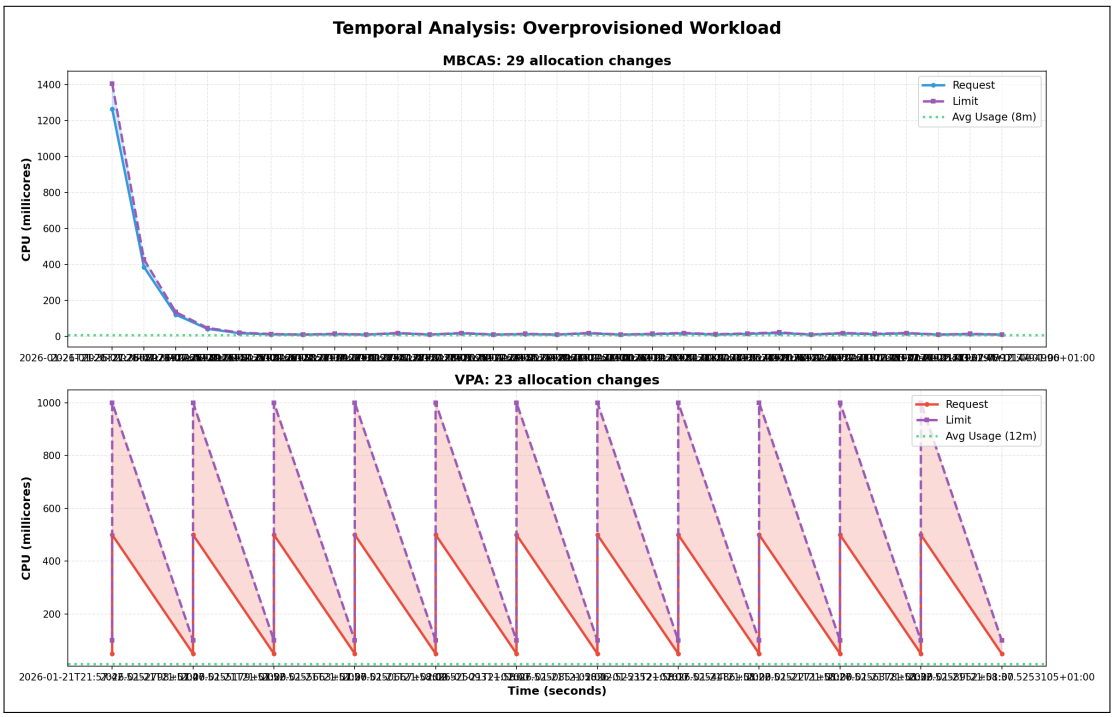

FIGURE 4.13 – Sur-allocation CPU : comparaison MBCAS vs VPA

La Figure 4.13 montre que MBCAS limite efficacement la sur-allocation CPU dans les scénarios faiblement chargés. Le VPA conserve des marges importantes par sécurité, entraînant un gaspillage de ressources, particulièrement pénalisant dans des environnements multi-locataires.

4.3.9 Synthèse comparative

Critère	VPA	MBCAS
Réactivité	Moyenne	Élevée
Stabilité	Moyenne	Élevée
Throttling	Fréquent en charge dynamique	Limité et contrôlé
Sur-allocation	Importante	Réduite
Décision locale	Non	Oui
Fondement théorique	Heuristique	Théorie des jeux (Nash)

TABLE 4.10 – Comparaison synthétique entre VPA et MBCAS

4.3.10 Conclusion de la comparaison

Les résultats expérimentaux montrent que MBCAS surpasse le VPA sur plusieurs dimensions clés, notamment la réactivité, la stabilité et l'efficacité globale de l'utilisation CPU. En s'appuyant sur une approche décentralisée, des métriques bas niveau et une négociation coopérative inspirée de la théorie des jeux, MBCAS offre une alternative robuste et économiquement plus efficiente au scaling vertical classique de Kubernetes.

4.4 Conclusion

Synthèse du chapitre

Ce chapitre a présenté une implémentation détaillée du système MBCAS, illustrant la mise en œuvre concrète de concepts avancés de théorie des jeux et d'apprentissage par renforcement au sein d'un opérateur Kubernetes *cloud-native*, robuste et extensible.

L'architecture modulaire, la tolérance aux pannes et l'intégration native avec Kubernetes démontrent la viabilité de l'approche proposée pour l'allocation dynamique et intelligente des ressources CPU dans des environnements de production.

Chapitre 5

Cinquième Chapitre

Conclusion Générale

Le Projet Fédérateur constitue, par essence, la clé de voûte de notre formation d'ingénieur en Génie Logiciel à l'ENSIAS. Plus qu'un simple exercice de développement, le projet **MBCAS** (Market-Based CPU Allocation System) a représenté pour nous un véritable laboratoire de recherche et d'expérimentation, nous permettant de synthétiser et de mettre à l'épreuve l'ensemble des compétences acquises durant notre cursus. Face à la complexité croissante des infrastructures Cloud, nous avons dû dépasser le cadre de l'application classique pour embrasser une démarche scientifique rigoureuse, allant de la modélisation mathématique abstraite jusqu'à l'implémentation système de bas niveau.

Sur le plan conceptuel, ce projet a été l'occasion d'une construction intellectuelle majeure : l'appropriation de la Théorie des Jeux et des Systèmes Multi-Agents pour résoudre des problèmes d'ingénierie concrets. L'étude approfondie de la Solution de Négociation de Nash et du Q-Learning ne s'est pas limitée à leur application technique ; elle nous a permis de développer une capacité d'abstraction nécessaire pour modéliser des interactions complexes. Nous avons appris à traduire des contraintes industrielles (gaspillage de ressources, latence) en équations mathématiques d'optimisation (efficacité de Pareto), consolidant ainsi notre aptitude à concevoir des systèmes non seulement fonctionnels, mais théoriquement robustes et justifiables.

D'un point de vue technique, la réalisation de MBCAS a exigé une montée en compétences significative sur les technologies de pointe ("Bleeding Edge"). La maîtrise de l'écosystème Kubernetes et du langage Go a dû être poussée bien au-delà des standards académiques habituels pour toucher aux mécanismes internes du noyau Linux (Cgroups v2). La mise en œuvre de l'*In-Place Pod Vertical Scaling* a été particulièrement formatrice : elle nous a confrontés à la réalité de la recherche technologique, nous obligeant à naviguer dans des documentations expérimentales et à innover là où les solutions "sur étagère" n'existaient pas encore. Cette immersion dans la programmation système a affiné notre rigueur et notre compréhension des enjeux de performance à grande échelle.

Au-delà des aspects purement techniques, ce travail a forgé notre identité de futurs ingénieurs. Il nous a fallu orchestrer une architecture distribuée complexe, gérer l'incertitude liée à l'apprentissage par renforcement et arbitrer des choix de conception critiques entre réactivité et stabilité. Cette expérience a renforcé notre autonomie, notre capacité d'analyse critique vis-à-vis des outils existants (comme le VPA standard) et notre aptitude à proposer des alternatives innovantes. Elle témoigne de notre passage d'une posture d'apprenant à celle de concepteur, capable d'appréhender un problème systémique dans sa globalité.

En définitive, ce projet fédérateur aura été le catalyseur de notre transition vers le monde professionnel et la recherche. En réussissant à faire converger l'Intelligence Artificielle, la Recherche Opérationnelle et le Cloud Computing au sein d'une solution unifiée, nous avons acquis la conviction que l'ingénierie logicielle moderne ne se résume pas à l'écriture de code, mais réside dans la capacité à bâtir des ponts entre des disciplines variées pour répondre aux défis technologiques de demain.

Bibliographie