Apprentissage par renforcement flou pour l'auto-adaptation des systèmes conteneurisés en périphérie de réseau

par

Imen Kanzali

MÉMOIRE PRÉSENTÉ À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
COMME EXIGENCE PARTIELLE À L'OBTENTION DE LA MAÎTRISE
AVEC MÉMOIRE EN GÉNIE DES TECHNOLOGIES DE
L'INFORMATION
M. Sc. A.

MONTRÉAL, LE 11 AOÛT 2021

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

PRÉSENTATION DU JURY

CE MÉMOIRE A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE:

M. Abdelouahed Gherbi, directeur de mémoire
Département de génie logiciel et des technologies de l'information à l'École de technologie supérieure

M. Mohamed Faten Zhani, président du jury
Département de génie logiciel et des technologies de l'information à l'École de technologie supérieure

Mme. Nadjia Kara, membre du jury
Département de génie logiciel et des technologies de l'information à l'École de technologie supérieure

IL A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC

LE '21 JUILLET 2021'

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

REMERCIEMENTS

Tout d'abord, je tiens à remercier le Professeur Abdelouahed Gherbi pour son encadrement durant la réalisation de ma maitrise.

Je suis également reconnaissant à Ahmed Bali, pour ses précieux conseils et son encadrement.

Un grand merci à Mitacs et à l'ÉTS pour le financement.

Je dédie mon travail à mes parents pour tout le soutien, les sacrifices et la patience qu'ils m'ont toujours affordés.

Enfin, je tiens à remercier mon frère, ma sœur et mes amis qui ont toujours cru en moi et m'ont soutenu.

Apprentissage par renforcement flou pour l'auto-adaptation des systèmes conteneurisés en périphérie de réseau

Imen Kanzali

RÉSUMÉ

Un des aspects désirés d'une architecture de systèmes conteneurisés auto-adaptative est la nécessité de pouvoir surveiller en permanence l'environnement opérationnel, de détecter et d'observer les anomalies, et de fournir une politique raisonnable de mise à l'échelle, de récupération et de configuration automatique des ressources de traitement afin de répondre dynamiquement à un changement brutal de son environnement opérationnel.

Le comportement d'une architecture de microservices évolue constamment dans le temps, ce qui rend difficile l'utilisation d'un modèle statique pour planifier l'action d'exécution appropriée en cas de détection d'une anomalie.

Pour obtenir les excellents niveaux d'auto-adaptabilité voulus, cette recherche permet de réaliser un modèle d'architecture de microservices suivant le modèle MAPE-K. Notre solution propose une combinaison de l'apprentissage Q, de la logique floue et des réseaux de neurones pour sélectionner dynamiquement l'action d'adaptation qui apporte la plus grande récompense.

La présence d'apprentissage Q flou (FQL) permet d'apprendre et de modifier les règles d'adaptation floues au moment de l'exécution sans nécessité de connaissances préalables. L'implémentation de Réseau Q profond flou (FDQN) dans le contexte du processus d'adaptation améliore l'efficacité de l'adaptation et réduit les risques associés à l'adaptation, notamment la fluctuation des ressources.

L'expérimentation démontre la faisabilité et l'utilité de nos approches. De plus, les résultats montrent que les performances de FDQN sont meilleures que celles de FQL.

Mots-clés: systèmes auto-adaptatifs, conteneurs, microservice, apprentissage par renforcement, la logique floue, réseaux de neurones, MAPE-K

Fuzzy reinforcement learning for self-adaptive containerized systems on the Edge

Imen Kanzali

ABSTRACT

Among the desired aspects of a self-adaptive containerized systems architecture is the need to be able to permanently survey the operational environment, catch and observe anomalies, and provide a good policy for self adaptation, recovery, and configuration of processing resources to dynamically respond to an abrupt change in its operational environment.

The behavior of a microservices architecture is constantly changing over time, making it difficult to use a static model to plan the appropriate runtime action when an anomaly is detected.

To achieve the excellent levels of self-adaptability desired, this research allows us to realize a microservices architecture model following the MAPE-K model. Our solution proposes a combination of Q-learning, fuzzy logic and neural networks to dynamically select the adaptation action that brings the highest reward.

The presence of fuzzy Q-learning (FQL) allows fuzzy adaptation rules to be learned and modified at runtime without the need for prior knowledge. The implementation of Fuzzy Deep Q Network (FDQN) in the context of the adaptation process improves the efficiency of adaptation and reduces the risks associated with adaptation, including resource fluctuation.

The experimentation demonstrates the feasibility and utility of our approaches. Moreover, the results show that the performance of FDQN is better than that of FQL.

Keywords: auto-adaptive system, containers, microservice, reinforcement learning, fuzzy logic, neural networks, MAPE-K

TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

Page

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

MAPE-K	Monitor Analysis Plan Execute Knowledge
FQL	Fuzzy Q Learning
RL	Reinforcement Learning
FDQN	Fuzzy Deep Q Network
DQN	Deep Q Network
MSE	Mean Squared Error
MAE	Mean Absolute Error
RELU	Rectified Linear Unit
CPU	Central processing unit
VM	Virtual Machine
HMM	Hidden Markov Model
HHMM	Hierarchical Hidden Markov Model

INTRODUCTION

0.1 Contexte et motivation

L'Internet des objets (IoT) aborde graduellement tous les aspects de notre vie. Comme les systèmes IoT deviennent de plus en plus nombreux, ils jouent un rôle important dans nos activités courantes. A titre d'exemple, les smartphones, nous les exploitons pour nous assister dans de nombreuses tâches (informations sur les horaires des transports publics, météo, courrier électronique, messagerie instantanée, etc.).

Les ressources limitées des appareils nécessitaient une plateforme riche en ressources pour traiter les données et effectuer des calculs au profit de ces appareils. L'informatique en nuage a été la solution à ce problème, car elle offrait des ressources plus importantes (par exemple, des ressources de traitement, de mémoire et de stockage) pour répondre aux besoins de calcul des applications. Cependant, la collecte de toutes les ressources informatiques dans un environnement en nuage distant a posé des problèmes pour les applications qui sont sensibles à la latence et au temps de réponse.

L'informatique de périphérie a été conçue comme une solution à ce défit, plusieurs applications bénéficient de cette technologie. Avec le développement rapide des infrastructures en périphérie de réseau et des techniques de virtualisation, une forte demande pour le déploiement d'architectures microservices dans un environnement entièrement virtualisé a émergé. Les microservices améliorent la modularité des logiciels et rendent l'application facile à développer et à maintenir. Cette nécessité a été satisfaite par le développement de conteneurs tels que Docker ainsi que framework de gestion de cluster comme Docker swarm.

Toutefois, ce nouveau système informatique se distingue par la limitation en termes de ressources système, de capacité de calcul ainsi que par le paradigme de calcul qui est basé sur le concept de conteneur. Cela exige une demande d'adaptation de la scalabilité et de la réparti-

tion de la charge offerte par le leader du cluster. Cela entraîne phénomène de fluctuation des performances du système.

De nos jours, il est indispensable que l'architecture microservice soit capable de traiter son propre état de son environnement dans une boucle de contrôle fermée et d'agir dynamiquement au moment de l'exécution pour atteindre un haut niveau d'adaptabilité.

0.2 Problématique

En raison de la complexité des systèmes, de la nature distribuée, de l'hétérogénéité et de la grande échelle des conteneurs et des nœuds, les utilisateurs peuvent rencontrer des anomalies entraînant une dégradation des performances et d'éventuelles défaillances des applications.

Les architectures de ces systèmes IoT incluent des nœuds disposant de ressources très limitées pour le traitement des programmes, ainsi que de processeurs faibles fonctionnant à des fréquences très basses.

En effet, ces nœuds ne disposent pas de composants capables de surveiller et de s'adapter en permanence à l'environnement opérationnel. Il est donc impossible de garantir une reprise dynamique lorsque le contexte change au moment de l'exécution.

Le processus d'auto-adaptation permet d'acquérir ou de libérer automatiquement des ressources en fonction de l'évolution du contexte. la fluctuation de la mise à l'échelle des ressources reste toujours un défi qui n'est pas encore totalement résolue.

0.3 Objectifs du travail

Notre objectif est de fournir un système auto-adaptatif fiable et léger qui peut effectuer une ou plusieurs interventions pour différentes défaillances survenues dans un environnement informatique conteneurisé.

Afin de réaliser un système de récupération automatique stable, nous divisons notre objectifen deux sous objectifs, comme suit :

- Notre approche doit avoir la capacité d'adapter horizontalement et / ou verticalement les ressources de l'architecture afin d'atteindre un état stable ;
- Notre architecture doit avoir la capacité de répondre dynamiquement à la charge de travail et de maintenir l'état du cluster en réduisant le degré de fluctuation de l'adaptation.

0.4 Contributions

Les principaux points de ce mémoire sont deux contributions qui visent à fournir un modèle auto-adaptatif complet capable de reconfigurer et de récupérer des composants logiciels lors d'une défaillance.

Cette recherche implémente une architecture de microservices suivant le modèle MAPE-K (Monitor Analysis Plan Execute Knowledge).

1. Première contribution : une intégration de la logique Floue avec l'apprentissage Q (FQL)

   Afin d'atteindre des niveaux élevés d'auto-adaptation aux changements contextuels de l'environnement opérationnel, les systèmes d'inférence floue ont l'avantage d'obtenir de bonnes approximations dans la fonction Q et permettent simultanément d'utiliser l'apprentissage Q dans des problèmes d'espaces continus d'états et d'actions ;
2. Deuxième contribution : une intégration de la logique floue et le DQN en exploitant les avantages des deux approches(FDQN). Ce modèle est idéal pour résoudre des problèmes complexes en estimant la valeur Q, en définissant une fonction d'approximation et en entraînant le modèle dans le DQN.

0.5 Organisation du mémoire

La mémoire est organisée comme suit :

- le premier chapitre explique les définitions de différentes notions pour comprendre les systèmes auto-adaptatifs et le cycle d'adaptation MAPE-K ;
- le deuxième chapitre aborde les différentes approches et contributions concernant le développement de systèmes auto-adaptatifs ;
- le troisième chapitre concentre sur nos deux modèles en tant que deux approches de la planification dans la boucle MAPE-K ;
- le quatérième chapitre décrit l'architecture conteneurisée utilisée et les services de surveillance et de détection des anomalies adoptés. Ce chapitre présente également une analyse des résultats.

CHAPITRE 1

DÉFINITIONS ET NOTIONS DE BASE

1.1 Introduction

Ce chapitre explique les différents termes qui ont été utilisés dans notre thèse.

Le premier concept à connaître est la définition des systèmes auto-adaptatifs. Il est nécessaire de comprendre les propriétés de base de ce système. De plus, la gestion des conteneurs et des clusters est une partie intéressante de notre travail. Les comportements de chaque type d'anomalie et leurs performances sont également importants.

Finalement, les notions importantes de méthodes sont bien exploitées dans notre domaine, comme l'apprentissage automatique et la logique floue.

1.2 Systèmes auto-adaptatifs

Un système auto-adaptatif modifie son propre comportement suite aux changements de son environnement et de son système d'exploitation. Il dépend des propriétés du système, des exigences de l'utilisateur et des caractéristiques de l'environnement.

Le système auto-adaptatif est consistitué d'un système en boucle fermée qui change au moment de l'exécution en utilisant des activités de surveillance (M), d'analyse (A), de planification (P) et d'exécution (E) basées sur la connaissance de l'expertise de détection d'anomalies tel que montré par la figure 1.1 .

Les systèmes informatiques autonomes connaissent ces activités sous le nom de cycle MAPE-K.

Ce mécanisme d'auto-adaptation est constitué de :

- **surveillance :** ce service permet de collecter en permanence des données précises concernant les nœuds, les services et les conteneurs du cluster, notamment :
  - l'utilisation du processeur ;
  - la mémoire ;
  - les lectures de disque en octets/seconde ;
  - Réseau, etc.
- **analyse :** ce service est chargé de consulter les observations collectées lors de la phase de surveillance et d'analyser les changements significatifs dans un système ;
- **plan :** une fois qu'un comportement anormal est détecté, ce service propose les actions nécessaires pour atteindre les objectifs du système. Cet objectif consiste à identifier l'action d'adaptation qui donne la meilleure récompense et préserve l'état de l'architecture.
- **execution :** ce service applique les actions spécifiées dans la phase de planification pour améliorer le comportement des éléments gérés ;
- **la connaissance :** ce stockage contient des données standard telles que des historiques accompagnés de mesures, de règles et de politiques. Ces données sont partagées entre les étapes de contrôle, d'analyse et de planification en fonction des exigences du service.

L'auto-adaptation est déclenchée par la détection des défaillances ou la prévision des futures violations des exigences.

Les exigences jouent un rôle important dans la conception de systèmes auto-adaptatifs, car leur satisfaction est le principal objectif du développement de tout système.

Un système auto-adaptatif surveille en permanence les changements et met à jour les modèles, qui sont analysés pour détecter toute violation des exigences. Pour ce faire, divers modèles d'analyse des fonctionnalités sont utilisés au moment de l'exécution, et des politiques de récupération peuvent également être appliquées pour traiter les violations. Par exemple, les modèles de Markov peuvent être utilisés pour prédire la fiabilité et les performances du système.

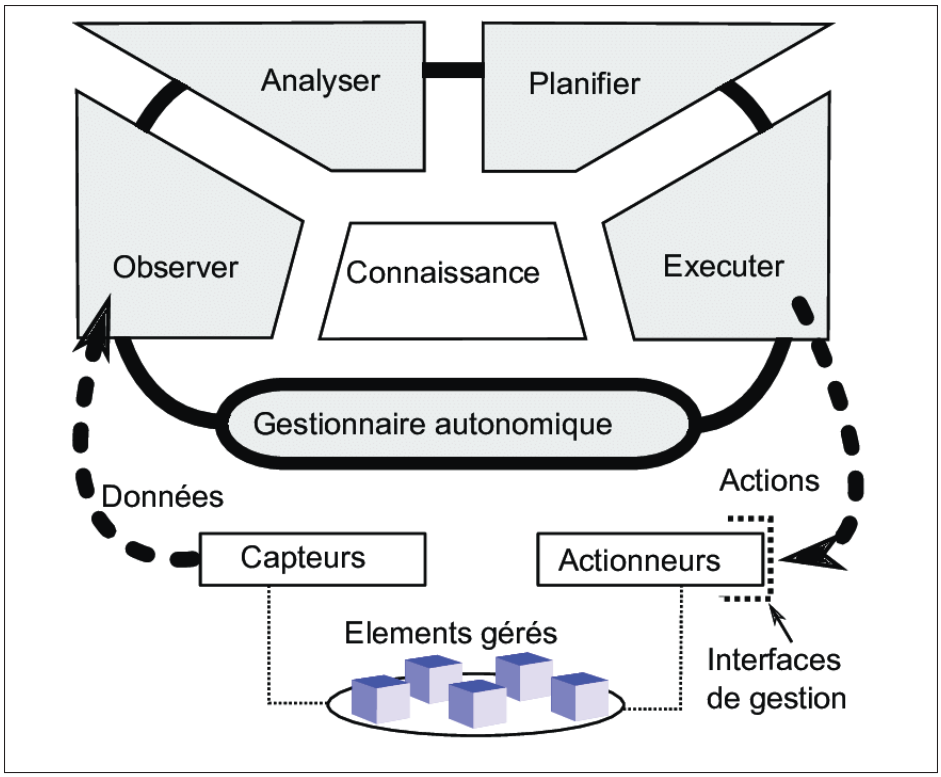

Figure 1.1 MAPE-K

Un système auto-adaptatif traite les changements en cours d'exécution en ce qui concerne les exigences fonctionnelles et non fonctionnelles.

Les exigences fonctionnelles représentent les fonctionnalités d'un système (par exemple, l'envoi de données d'un endroit à un autre).

Les exigences non fonctionnelles décrivent généralement la qualité de la fonctionnalité d'un système (par exemple, le temps de réponse d'une requête).

1.3 Anomalie de performance

Une anomalie est une divergence dans le comportement d'un système qui dégrade ses performances.

Certains auteurs, comme Malkowski *et al.* (2009), décrivent une anomalie comme la raison fondamentale d'un comportement inattendu en termes de performances, causé par une saturation élevée de certaines ressources majeures du système.

Ces composants montrent souvent une congestion fréquente de la charge de travail. De plus, d'autres études telles que Jung *et al.* (2006) mentionnent que certaines applications ou métriques système en corrélation avec une limitation de performance observée sont appelées métriques d'anomalie.

1.4 Types d'anomalies

La bonne maîtrise des propriétés des fautes est un aspect fondamental pour fournir un système fiable et évolutif. Une anomalie se traduit par l'apparition d'une ou plusieurs défaillances qui affectent de manière significative les performances des composants du système, ainsi que celles du système global.

Si un composant présente une faible performance inattendue, cela indique une anomalie de performance.

L'anomalie de performance se manifeste de différentes manières selon les applications et les systèmes :

- **apparition d'une anomalie unique :** dans ce cas, une anomalie présente une saturation prédominante d'une seule ressource (par exemple, CPU, mémoire, disque, E/S réseau) pour un composant (conteneur, nœud ou cluster) ;
- **apparition d'anomalies multiples :** les ressources de nombreux composants du système peuvent saturer simultanément en raison de l'interdépendance entre les composants.

  Les travaux de (Malkowski *et al.*, 2009; Lee *et al.*, 2011) se concentrent sur la saturation d'une ressource.

  Ils classent la saturation d'une ressource en plusieurs comportements de saturation tels que :

  • simultané se produit si plusieurs ressources sont entièrement saturées indépendamment ;

- oscillatoire se produit lorsque plusieurs ressources forment un étranglement combiné tandis que l'anomalie concurrente se produit si plusieurs ressources saturées indépendamment pendant une fraction de la période d'observation.

Toutefois, cette classification n'est pas retenue comme base pour la classification des anomalies, puisque le travail de Avizienis (2001) fournit une autre classification :

- **prédiction des défaillances :** estimation de l'occurrence actuelle et future des défaillances ;
- **tolérance aux défaillances :** évite les défaillances en cas d'anomalies ;
- **la prévention des défaillances :** elle empêche l'apparition d'une défaillance ;
- **élimination des défaillances :** réduit le nombre et la gravité des défaillances.

Le traitement des comportements indésirables empêche les anomalies de se reproduire. Pour traiter ces anomalies, de multiples techniques sont utilisées comme la survivabilité, la vérification et la validation (par exemple, les tests et l'analyse dynamique) (Avizienis, 2001).

1.5 Gestion des conteneurs et des clusters

Un nœud est une machine de travail qui peut être une machine virtuelle (VM) ou une machine physique, il est généralement associé à un cluster.

Il peut être composé d'un ou plusieurs conteneurs, chaque conteneur exécute un ensemble de tâches. Les nœuds et les conteneurs sont associés à un cluster qui exécute les nœuds sur une machine locale, ou sur un environnement en nuage.

Les conteneurs d'un nœud sont générés à partir d'une image de conteneur, qui correspond à un package exécutable d'un logiciel.

Afin de planifier et d'orchestrer les conteneurs, un agent peut être installé au niveau du nœud.

Chaque nœud d'un cluster contient des informations sur son adresse, son statut ou sa disponibilité, c'est-à-dire des informations concernant la disponibilité de ses ressources par exemple, CPU, mémoire et le nombre maximum de conteneurs qui peuvent être programmés sur le nœud.

Il possède également des informations générales telles que le nom du système d'exploitation.

Les ressources sont divisées de façon égale ou inégale entre ses conteneurs. Il existe à cet effet de nombreux outils de gestion de conteneurs tels que Docker Swarm.

1.5.1 Docker swarm

Docker [1] est une plate-forme permettant de développer et d'exécuter des applications indépendamment de l'infrastructure.

Le mode swarm permet aux développeurs et aux administrateurs de créer et de gérer un système virtuel, un " swarm ", composé d'un ou de plusieurs nœuds qui constituent un cluster.

Chaque nœud d'un cluster exécute un agent swarm, contrôlé par les managers qui programment et orchestrent les conteneurs.

Il utilise une interface standard (par exemple, API) pour accomplir ses tâches.

Docker swarm se compose principalement de deux nœuds :

- **nœud gestionnaire (manager) :** gère l'infrastructure de Docker, comme la programmation des services et le maintien de l'état du cluster. Il contient un ou plusieurs nœuds de travail(worker) ;
- **nœud worker :** responsable de l'exécution des conteneurs. Un nœud travailleur peut devenir un nœud manager, lors d'une maintenance sur le nœud manager.

1. https ://docs.docker.com/engine/swarm/

Docker swarm sépare les travailleurs des gestionnaires de façon que les nœuds des travailleurs se concentrent sur les applications de l'entreprise, tandis que les nœuds des managers gèrent les tâches de gestion du cluster.

Docker swarm permet de réparér les nœuds en appliquant des actions telles que le redémarrage, l'arrêt, la suppression, la création du nouveau nœud et la restauration. Lorsqu'un nœud ayant des tâches en cours d'exécution devient indisponible, les tâches sont confiées aux nœuds non occupés, ce noeud reçoit une nouvelle charge pour rééquilibrer la répartition.

Sinon, les conteneurs du nœud peuvent être migrés vers un autre nœud disponible, cependant, cette opération redémarre et déplace les tâches vers différents nœuds, de sorte que les clients utilisant ces tâches seraient perturbés. Une autre solution proposée par Docker swarm consiste à ajouter un nœud gestionnaire pour gérer la charge.

1.6 Apprentissage par Renforcement

L'apprentissage par renforcement (RL) (Sutton, 1998) est une approche de prise de décision automatique qui a été utilisé par plusieurs auteurs pour réaliser des adaptations automatiques. Sans aucune connaissance a priori, les techniques d'apprentissage par renforcement sont capables de déterminer la meilleure action de mise à l'échelle à prendre pour chaque situation.

Dans le cadre de l'apprentissage par renforcement, un agent prend une action a_i lorsque le système est dans l'état s_t, laisse le système évoluer vers l'état suivant s_{t+1} et observe le signal de renforcement r_{t+1}.

La prise de décision dans les systèmes élastiques peut être représentée comme une interaction entre les contrôleurs de nuages et l'environnement.

Le contrôleur de nuage surveille l'état actuel du système à l'aide de ses capteurs. En fonction de certaines connaissances, il choisit une action et évalue la récompense en retour sous forme de fonctions d'utilité (Walsh *et al.*, 2004).

La situation permet au système de savoir quand il doit surveiller l'état, et également quand il doit effectuer l'action (c'est-à-dire déclencher l'action de mise à l'échelle).

Un système flexible peut rester dans le même état, mais doit prendre des mesures différentes selon les différentes situations et l'intensité de la charge de travail.

L'objectif de l'agent est de trouver une politique π qui associe chaque état s à la meilleure action a que l'agent devrait choisir.

L'agent doit maximiser les récompenses attendues actualisées obtenues à long terme selon l'équation 1.1.

$$R_t = r_{t+1} + \gamma r_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{1.1}$$

La politique est basée sur une fonction de valeur Q(s, a), généralement appelée fonction Q-valeur. Chaque valeur Q(s, a) estime les récompenses cumulées futures en exécutant une action a dans un état s. Autrement, elle représente l'intérêt d'exécuter l'action a dans l'état s.

1.7 Conclusion

Ce chapitre aborde les différents concepts qui ont été utilisés dans la mémoire tels que les systèmes auto-adaptatifs, la gestion des conteneurs et des clusters, les anomalies de performance, les types de comportements anormaux et la notion d'apprentissage par renforcement.

CHAPITRE 2

REVUE DE LA LITTÉRATURE

2.1 Introduction

Ce chapitre présente un aperçu des recherches les plus pertinentes qui mettent en évidence l'analyse des performances anormales et les stratégies d'adaptation efficaces et réussies.

2.2 Processus d'auto-adaptation : la boucle MAPE

Le processus d'auto-adaptation correspond à la boucle MAPE des systèmes autonomes (Maurer *et al.*, 2011), qui se compose de quatre phases : Surveillance (M), Analyse (A), Planification (P) et Exécution (E).

Tout d'abord, un système de surveillance recueille des informations sur l'état du système et de l'application. L'auto-scaler englobe les phases d'analyse et de planification : il utilise les informations récupérées pour faire des estimations de l'utilisation et des besoins futurs en termes de ressources (analyse), puis planifie une action de modification des ressources appropriée, par exemple la suppression d'une VM ou l'ajout de mémoire supplémentaire.

Le fournisseur exécute ensuite les actions demandées par l'auto-scaler.

Les phases de la boucle MAPE sont décrites de manière plus détaillée dans ce qui suit.

2.2.1 Surveillance

Un système de mise à l'échelle automatique nécessite le soutien d'un mécanisme de surveillance fournissant des mesures sur les demandes des utilisateurs, l'état du système (application)

Les infrastructures en nuage fournissent, par l'intermédiaire d'une API, un accès à des informations utiles pour le fournisseur lui-même (par exemple, pour vérifier le bon fonctionnement de l'infrastructure et le niveau d'utilisation) et pour le client (par exemple, pour vérifier le niveau de service).

Les décisions de mise à l'échelle automatique dépendent de l'existence de données utiles et actualisées.

Dans la littérature, les auteurs ont utilisé une variété de métriques comme facteurs de décision pour la mise à l'échelle.

Une liste complète de ces mesures est fournie par Ghanbari *et al.* (2011), pour les charges de travail transactionnelles .

- **le matériel :** utilisation du CPU par VM, accès au disque, accès à l'interface réseau, utilisation de la mémoire ;
- **processus général du système d'exploitation :** temps du CPU par processus, mémoire réelle ;
- **serveur d'application :** nombre total de threads, nombre de threads actifs, mémoire utilisée, nombre de sessions, demandes traitées, demandes en attente, demandes abandonnées, temps de réponse.

Afin de réduire la complexité du système de surveillance, des mesures proxy sont parfois utilisées : par exemple, l'utilisation de la CPU (au niveau de l'hyperviseur) comme un proxy de la charge de travail actuelle du système (au niveau de l'application).

2.2.2 Analyse

La phase d'analyse consiste à traiter les métriques recueillies directement auprès du système de surveillance, obtenir des données sur l'utilisation actuelle du système et, éventuellement, des prévisions sur les besoins futurs.

Certains systèmes de mise à l'échelle automatique ne font aucune prédiction et répondent seulement à l'état actuel du système : ils sont réactifs.

Cependant, d'autres utilisent des techniques sophistiquées pour prédire les demandes futures afin d'organiser l'approvisionnement en ressources avec suffisamment d'anticipation : ils sont proactifs.

L'anticipation est importante car il y a toujours un délai entre le moment où une action d'adaptation est exécutée (Mooers, 1977) .

Une partie des études revues se concentre uniquement sur cette phase d'analyse, sur la manière de traiter les métriques pour déterminer l'état actuel de l'application ou anticiper les besoins futurs.

2.2.3 Planification

Une fois que l'état actuel (ou futur) est déterminé (ou prédit), l'auto-scaler est chargé de planifier la mise à l'échelle des ressources affectées à l'application en fonction de l'état actuel.

Des exemples de décisions d'adaptation sont la suppression d'une VM ou l'ajout de mémoire à une VM particulière.

Les décisions seront prises en tenant compte des données obtenues lors de la phase d'analyse (ou directement à partir du système de surveillance) et du SLA visé, ainsi que d'autres facteurs liés à l'infrastructure en nuage, notamment les modèles de tarification et le temps de démarrage des VM. Cette phase constitue le noyau de toute proposition de mise à l'échelle automatique.

2.2.4 Exécution

Cette phase consiste à exécuter réellement les actions de mise à l'échelle décidées à l'étape précédente.

Conceptuellement, il s'agit d'une phase simple, implémentée via l'API du fournisseur. Les complexités réelles sont cachées au client. il faut un certain temps entre le moment où une ressource est demandée et celui où elle est effectivement disponible, et que les limites de ces délais peuvent faire partie de l'accord de service pour la ressource.

2.3 Techniques d'auto-adaptation

Plusieurs travaux ont utilisé de nombreuses approches et techniques afin de surveiller, détecter et adapter leur architecture. Ces travaux sont classés de la manière suivante :

2.3.1 Règles basées sur des seuils (règles)

Les règles basées sur des seuils constituent un mécanisme facile à déployer et à utiliser pour gérer la quantité de ressources attribuées à une application hébergée sur une plateforme en nuage, en adaptant dynamiquement ces ressources à la demande d'entrée (Gorbil & Gelenbe, 2013; Dutreilh *et al.*, 2010; Han *et al.*, 2012).

Cependant, la création des règles nécessite un effort de la part du gestionnaire de l'application (le client), qui doit sélectionner la métrique de performance appropriée ou la combinaison logique de métriques, ainsi que les valeurs de plusieurs paramètres, principalement les seuils.

Les expériences menées selon Koperek & Funika (2011) montrent que les mesures spécifiques aux applications (par exemple, le temps d'attente moyen dans la file d'attente), obtiennent de meilleures performances que les mesures spécifiques au système (par exemple, la charge du CPU).

Les gestionnaires d'applications doivent également définir les seuils supérieurs (par exemple, 70 %) et inférieurs (par exemple, 30 %) correspondants pour la variable de performance (par exemple, la charge CPU).

En particulier, Dutreilh *et al.* (2010) constatent que les seuils doivent être ajustés avec précaution afin d'éviter les oscillations dans le temps.

Pour prévenir ce problème, il est recommandé de fixer une période de repos, c'est-à-dire une période pendant laquelle aucune décision de mise à l'échelle ne peut être prise, une fois qu'une action a été effectuée.

La plupart des travaux et des fournisseurs de cloud n'utilisent que deux seuils par mesure de performance. Les conditions de ces règles sont généralement basées sur une seule ou au maximum deux mesures de performance, les plus populaires sont la charge CPU moyenne des VMs, le temps de réponse, et le débit des requêtes d'entrée.

Dutreilh *et al.* (2010); Han *et al.* (2012) utilisent le temps de réponse moyen de l'application. Au contraire Hasan *et al.* (2012) préfèrent utiliser des mesures de performance provenant de plusieurs resources, stockage et réseau), ou même une corrélation de plusieurs d'entre eux.

Le plus souvent, les règles utilisent directement les métriques obtenues du moniteur, agissant ainsi de manière purement réactive. Cependant, il est possible d'effectuer une analyse préalable des données surveillées afin de prédire le comportement futur (attendu) du système et d'exécuter les règles de manière proactive (Khatua *et al.*, 2010).

L'algorithme de mise à l'échelle automatique de RightScale propose de combiner des règles réactives régulières avec un processus de vote.

Si la plupart des VM sont d'accord pour augmenter ou réduire leur taille, cette action est réalisée, sinon, aucune action n'est prévue Chaque VM vote pour augmenter ou diminuer sa dimension en fonction d'un ensemble de règles évaluées individuellement.

Après chaque action de mise à l'échelle, RightScale recommande une période de calme de 15 minutes parce que les nouvelles machines mettent généralement entre 5 et 10 minutes pour être opérationnelles.

Cette technique a été adoptée par plusieurs auteurs (Ghanbari *et al.*, 2011; Simmons *et al.*, 2011).

La popularité des règles en tant que méthode de mise à l'échelle automatique est probablement due à leur simplicité et à leur bonne compréhension par les clients.

Cependant, ce type de technique présente deux problèmes principaux :

- sa nature réactive ;
- la difficulté de sélectionner le bon ensemble d'indicateurs de performance et les seuils correspondants.

L'efficacité de ces seuils dépend fortement de l'évolution de la charge de travail et nécessite parfois des ajustements fréquents.

Afin de résoudre le problème des seuils statiques, Lorido-Botrn *et al.* (2013) introduisent le concept de seuils dynamiques : des valeurs initiales sont établies, mais elles sont automatiquement modifiées en fonction des violations de SLA observées.

En conclusion, les règles peuvent être utilisées pour automatiser facilement l'adaptation de ressources d'une application particulière sans trop d'effort, notamment dans le cas d'applications avec des tâches assez réguliers et prévisibles.

Cependant, dans le cas de charges de travail éclatées, il faudrait envisager un système auto-adaptatif plus avancé et plus puissant parmi les autres propositions (Lorido-Botran *et al.*, 2014).

2.3.2 Apprentissage Machine

L'apprentissage automatique a été déployé dans plusieurs domaines, notamment dans les systèmes auto-adaptatifs. Il a également été mis en œuvre avec d'autres approches, comme le montre la section suivante.

2.3.2.1 Modèle de Markov caché

De nombreuses études utilisent les HMM et leurs dérivés pour détecter les anomalies. L'auteur Sukhwani *et al.* (2014) propose différentes techniques implémentées pour la détection d'anomalies et d'intrusions dans le réseau en utilisant les HMM.

L'auteur Ge *et al.* (2015) détecte les défauts dans les systèmes embarqués en temps réel en utilisant les HMMs en décrivant les états de santé et de défaut des composants matériels du système.

Un modèle HMM est utilisé pour déterminer les anomalies qui font partie des mêmes scénarios d'injection d'anomalies (Tijsma, 2019).

L'étude présenté par Nikravesh *et al.* (2014) utilise les HMMs pour dimensionner automatiquement les ressources des applications en fonction de leur charge. Toutefois, ces travaux ne prennent pas en considération la charge d'une architecture dynamique multi-niveaux de mise à l'échelle automatique.

Les auteurs utilisent un modèle de Markov pour déterminer une défaillance du système en passant d'un état du système à un autre en fonction d'un niveau de défaillance. Ils appliquent une approche de régénération pour détecter un problème logiciel en estimant le temps moyen et la probabilité d'une défaillance.

Une réparation manuelle est menée en parallèle avec la réparation automatique pour corriger les défaillances dures et légères.

Samir & Pahl (2019) proposent une architecture auto-adaptative en utilisant des HMM d'apprentissage automatique et d'analyse statistique pour détecter, identifier et traiter les comportements anormaux dans un environnement de cluster conteneurisé.

Cependant, l'utilisation des HMMs nécessite de spécifier des configurations appropriées pour les paramètres des HMMs afin d'obtenir des résultats précis.

En outre, Samir & Pahl (2019) ont entraîné leur modèle sur des ensembles de données de petite et moyenne taille en utilisant un nombre fixe de nœuds qui peuvent être personnalisés ultérieurement afin d'éviter la pénalité de la complexité de calcul, qui peut survenir une fois que le nombre d'états augmente.

2.3.2.2 Apprentissage par renforcement

L'utilisation de l'apprentissage par renforcement (RL) pour la sélection dynamique des actions est populaire dans le domaine de l'architecture auto-adaptative.

Trois éléments de base doivent être définis pour appliquer RL à la mise à l'échelle automatique : l'ensemble d'actions A, l'espace d'état S et la fonction de récompense R.

Les deux premiers dépendent fortement du type d'auto-adaptation, horizontal ou vertical, tandis que la fonction de récompense prend généralement en compte le coût des ressources acquises (location de VM, bande passante, ...) et le coût de pénalité pour les violations de SLA.

Tesauro *et al.* (2006) proposent d'utiliser (w, u_{t-1}, u_t), où w est le nombre total de demandes d'utilisateurs observées par période de temps ; u_t et u_{t-1} sont le nombre de machines virtuelles allouées à l'application au cours du cycle actuel et du cycle précédent.

Dutreilh *et al.* (2011) ont considéré la définition de l'état (w, u, p), où p est la performance en termes de temps de réponse aux requêtes, bornée par une valeur Pmax choisie dans les observations expérimentales.

L'ensemble des actions de mise à l'échelle horizontale est généralement limité à trois : ajouter une nouvelle VM, supprimer une VM existante, et ne rien faire.

En revanche, la définition d'état pour la mise à l'échelle verticale tient compte de la quantité de ressources affectées à chaque VM (CPU et mémoire, principalement).

En effet, Rao *et al.* (2009); Xu *et al.* (2012) ont considéré la définition d'état suivante : (mem_1, $time_1$, $vcpu_1$, ... , mem_u, $time_u$, $vcpu_u$), où mem_i, $time_i$ et $vcpu_i$ sont la taille de la mémoire de la ième VM, le crédit de l'ordonnanceur et le nombre de CPU virtuels, respectivement.

Les études de Arabnejad *et al.* (2017) utilisent l'apprentissage par renforcement pour permettre à un logiciel ou à un système d'apprendre à réagir de manière adaptative dans un environnement déterminé afin de maximiser sa récompense.

Kim & Park (2009) ont démontré l'utilisation du Q-learning pour effectuer des changements de l'architecture dans un robot simulé. Ils utilisent une table Q pour toutes les paires état-action qui peuvent être utilisées pour calculer la valeur Q et de façon dynamique, l'algorithme sélectionne une action d'adaptation.

Cependant, les approches mentionnées ci-dessus sont confrontées à de nombreuses contraintes, notamment la tentative d'appliquer la RL à un ensemble discret d'états et d'actions limités à un domaine spécifique, ce qui limite la généralisation des approches présentées par Salehie & Tahvildari (2005); Tesauro (2007).

Pour dépasser ces limites, plusieurs efforts ont été faits pour utiliser des réseaux de neurones dans la mise en œuvre d'algorithmes de RL (Van Hasselt, 2012). (Van Hasselt, 2012).

1. **Réseau de neurones** : l'idée est d'utiliser le DQN pour identifier la relation mathématique entre les données d'entrée et pour identifier la fonction de récompense maximale pour trouver la sortie. Ces efforts se retrouvent dans les travaux de Lample & Chaplot (2017), où RL et le réseau de neurones ont été utilisés pour jouer à des jeux Atari ou à des jeux de Go (Sutton, 1998)

   Plusieurs méthodes ont également été proposées pour résoudre des problèmes de vision par ordinateur (Yun *et al.*, 2017), tels que la localisation d'objets (Caicedo & Lazebnik, 2015) ou la reconnaissance d'actions (Jayaraman & Grauman, 2016), en utilisant les algorithmes RL profonds

Basé sur l'algorithme DQN, différents réseaux neuronaux tels que le double DQN (Van Hasselt *et al.*, 2016) et le DQN duel (DDQN) (Wang *et al.*, 2016) ont été proposés pour améliorer les performances et maintenir la stabilité.

Une combinaison de mémoire à long et court terme (LSTM) (Zeng *et al.*, 2020) et de RNN est utilisée pour améliorer les connaissances des agents sur l'environnement observable.

Récemment, Magableh & Almiani (2020) ont deployé un réseau Q récurrent profond pour gérer les changements contextuels dans les logiciels dynamiques ;

2. **La logique Floue** : Xu *et al.* (2007) ont appliqué un contrôleur flou adaptatif au niveau de la logique commerciale d'une application Web, et ont estimé la charge CPU requise pour la charge de travail d'entrée.

   Une approche similaire est suivie par Wu *et al.* (2018), mais ici les auteurs se concentrent sur le niveau de la base de données, Ils affirment utiliser le modèle flou pour prédire les besoins futurs en ressources. Cependant, ils utilisent la charge de travail du moment t actuel pour calculer les besoins en ressources r_{t+1} du t + 1, en supposant que aucun changement brusque ne s'est produit pendant un intervalle de temps.

   FQL a été proposé comme méthode pour ajuster l'allocation des ressources dans l'infrastructure en nuage (Jamshidi *et al.*, 2016)

   Une autre amélioration consiste en un contrôleur neuronal flou, qui utilise un réseau de neurones à quatre couches pour représenter le modèle flou.

   Chaque nœud de la première couche correspond à une variable d'entrée. La deuxième couche détermine l'appartenance de chaque variable d'entrée à l'ensemble flou (le processus de fuzzification). Chaque nœud de la troisième couche représente la partie condition préalable d'une règle de logique floue.

   Et enfin, la couche de sortie agit comme un défuzzificateur, qui convertit les conclusions floues de la couche 3 en sortie numérique en termes d'ajustement des ressources.

   Aux premières étapes, le réseau de neurones ne contient que les couches d'entrée et de sortie.

Les nœuds d'appartenance et de règle sont générés dynamiquement par l'apprentissage de la structure et des paramètres d'apprentissage.

Lama & Zhou (2010) se sont appuyés sur un contrôleur flou neuronal, capable d'auto-construire sa structure (à la fois les règles floues et les fonctions d'appartenance) et d'adapter ses paramètres par un apprentissage en ligne rapide (un type de contrôleur reconfigurant).

Les tableaux 2.1 et 2.2 résument brièvement certaines études pertinentes mentionnées. Nous désignons par "x" la littérature qui soutient la détection, l'identification ou la récupération. Si la littérature ne supporte aucun de ces éléments, nous la citons par "-". Nous indiquons également par 'A' une approche automatisée et par 'NA' une approche non autorisée.

Tableau 2.1 Aperçu de quelques approches d'analyse des performances

Description	**Littérature**	**Méthodologie**	**Contribution**			
			Détection	Analyse	Recovery	Auto
Mise à l'échelle automatique permettant de gérer les charges de travail dynamiques	Salman 2018	algorithme	-	-	X	A
Mise à l'échelle automatique destiné à prédire Utilisation CPU	SamyAravind 2017	Analyse des séries chronologiques	X	-	X	A
Mise à l'échelle automatique pour réagir aux fluctuations de la charge de travail	Arabnejad 2017	Logique floue + Apprentissage par Renforcement	-	-	X	A
Modèle auto-adaptation des ressources	Nikravesh 2014	HMM	-	-	X	A
Outil aidant à trouver des bugs de performance en recherchant des boucles dans le code	Nistor 2013	algorithmes	X	-	-	A
Mise à l'échelle automatique pour gérer le bruit et l'incertitude dans le nuage.	Jamshidi 2016	Logique floue + Apprentissage par Renforcement	X	-	-	A

Tableau 2.2 Aperçu de quelques approches d'analyse des performances

Description	**Littérature**	**Méthodologie**	**Contrubition**			
			Détection	Analyse	Recovery	Auto
Une approche pour détecter une anomalie et d'appliquer la récupération des défauts	Avritzer 2015	modèle de Markov pour définir un modèle de défaillance	X	X	X	A
Mise à l'échelle automatique des VM pour gérer l'utilisation des ressources des applications	Murthy 2014	Algorithme	-	-	X	A
Surveiller, détecter, identifier, guérir et prévoir les charges de travail dans un environnement containerisé	Areeg Samir 2020	HMM et Algorithme	X	X	X	A
Architecture auto-adaptative de microservices distribués	Basel Magableh 2020	DQN	X	X	X	A
Apprentissage par renforcement flou pour l'auto-adaptation des systèmes conteneurisés en périphérie de réseau	Notre étude 2021	FQL et FDQN	X	X	X	A

Contrairement au déploiement de la logique floue dans les études mentionnées dans les sous-sections précédentes,

La gestion des changements contextuels inattendus dans l'environnement opérationnel est un problème complexe et une tâche difficile qui dépasse l'hypothèse d'un ensemble discret d'états et d'actions.

Cette situation exige un mécanisme pour gérer des espaces d'états et/ou d'actions continus, ce qui nécessite un modèle de contexte à l'exécution pour refléter les changements récents dans les environnements opérationnels.

Le travail proposé dans ce mémoire consiste à fournir un mécanisme de récupération flexible pour gérer les défaillances dans un environnement dynamique surveillé en déployant une combinaison de logique floue, d'apprentissage Q et de réseaux neuronaux.

2.4 Conclusion

Ce chapitre explore les travaux liés au sujet de mémoire , et discute des problèmes actuels de détection, d'identification et de récupération des anomalies, ainsi que des espaces d'état et des actions considérés.

CHAPITRE 3

NOTRE MODÈLE D'AUTO-ADAPTATION

En poursuivant le principe du modèle MAPE-K, nous présentons notre modèle auto-adaptatif qui nous permet de surveiller en permanence notre environnement opérationnel, de détecter et d'observer les comportements anormaux, et de fournir une politique d'adaptation raisonnable en utilisant une approche d'apprentissage machine. Dans ce chapitre, nous détaillons notre approche adaptative pour l'auto-réparation des ressources afin de répondre dynamiquement à un changement brutal de son contexte.

3.1 Description générale : Systèmes d'inférence flous

Dans le contexte de RL, la logique floue peut être utilisée comme une approche pour traiter des espaces continus d'états et d'actions. D'autre part, dans le contexte de la logique floue, l'apprentissage machine peut être utilisé pour régler les contrôleurs flous (Jouffe, 1998).

Soit x la variable d'entrée, x= (x^1, x^2,x^3,.....,x^N) et a_i =[a_i^1,.....,a_i^k,.....,a_i^K] est l'ensemble des actions de sortie du i ème règle. La règle R_i dans le système d'inférence floue a la forme suivante :

R_i : Si (x^1 Dans L_i^1) ...et (x^n Dans L_i^n) ... et (x^N Dans L_i^N) donc a= a_i et q[L_i,a_i]

Avec : L_i = [L_i^1,L_i^2,....,L_i^n,.....L_i^N] est appelé le vecteur de modèle correspondant à la règle i.

q(L_i , a_i^k) s'appelle la fonction de valeur q de l'état L_i et de l'action a_i^k.
N :les variables d'état dont l'agent a besoin pour prendre des décisions.
Le diagramme de FIS est illustré par la figure 3.1

Tout d'abord, les variables d'entrée et de sortie du système sont mappées en ensembles flous. Ce mappage est défini par une fonction d'appartenance qui détermine une valeur dans l'intervalle [0,1].

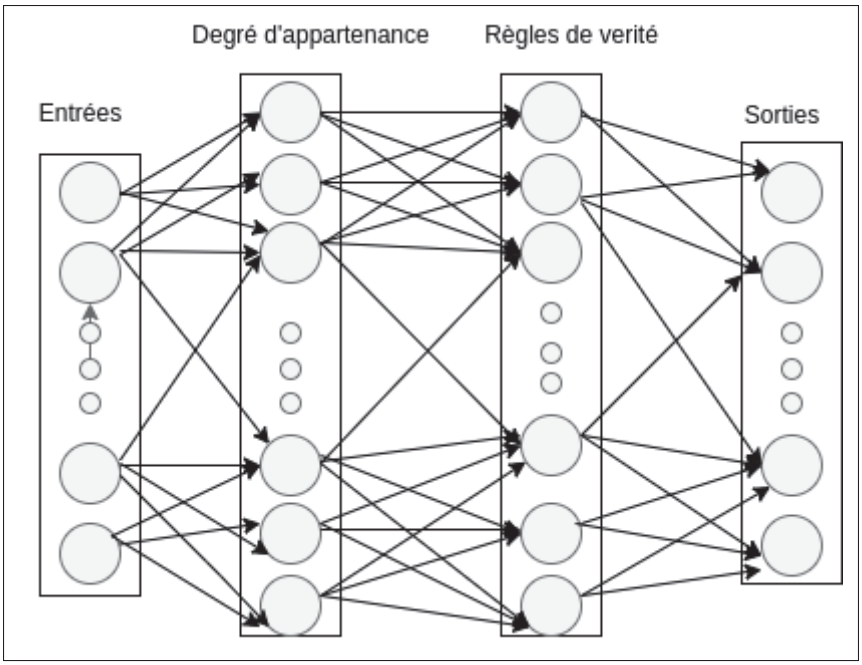

Figure 3.1 Structure du FIS

Un modèle flou est basé sur un ensemble de règles qui relient les variables d'entrée (condition préalable de la règle) aux variables de sortie (conséquence de la règle).

Comme notre problème concerne des espaces continus d'états et d'actions, nous utilisons la logique floue dans notre approche proposée comme représentation des données pour les variables d'entrée. Par conséquent, nous sommes concernés par le degré des valeurs de vérité.

3.2 Modèle de Q-learning flou

Le Q-learning flou (FQL) est une extension floue de l'algorithme Q-learning. Il s'agit d'une combinaison qui traite l'incertitude causée par des connaissances incomplètes. Si les connaissances expertes sont disponibles, elles sont encodées en termes de règles. De plus, elle permet de les encapsuler dans la table d'apprentissage, ce qui accélère le processus de formation. Dans le cas où il n'y a pas (ou peu) de connaissances accessibles au moment de la conception, FQL est toujours capable de fonctionner.

FQL a été appliqué avec succès aux problèmes de mise à l'échelle vertical dans le domaine du cloud computing. Par ailleurs, nous utilisons FQL comme solution à notre problème, les modules utilisés sont décrits ci-dessous.

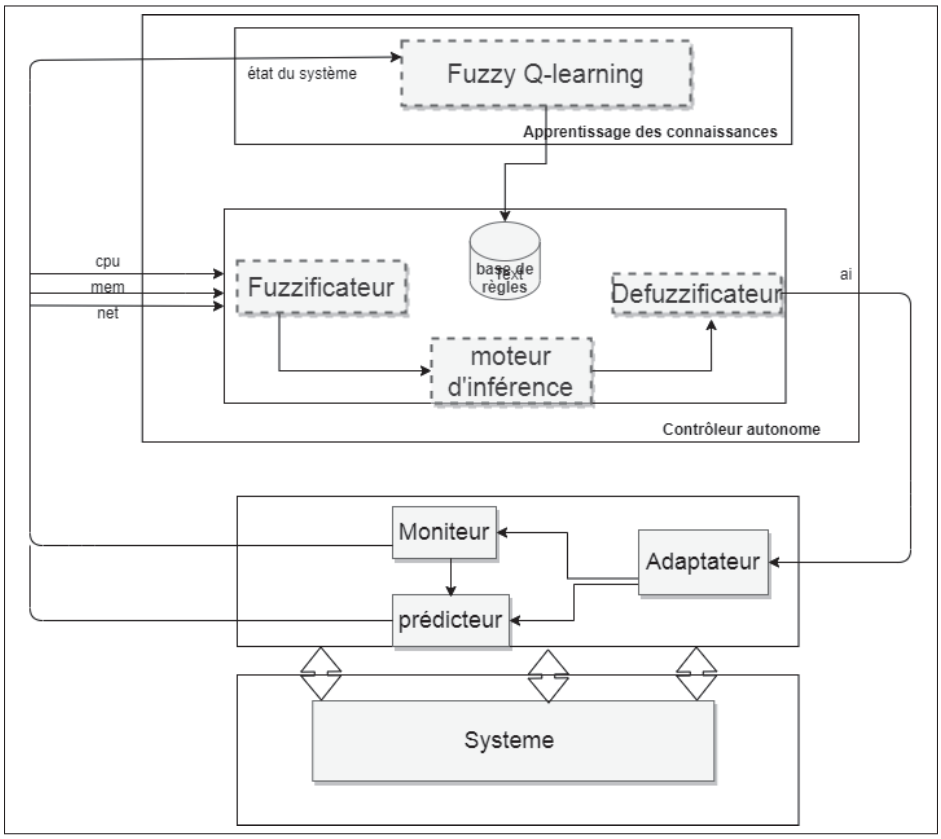

Figure 3.2 Architecture du modèle 1

3.2.1 Contrôleur à logique floue

L'inférence floue est le processus de mise en correspondance d'un ensemble d'entrées de contrôle avec un ensemble de sorties de contrôle par le biais de règles floues. Les entrées du contrôleur sont les métriques (cpu, net, mémoire) et la sortie est l'action d'adaptation (ai). La conception d'un contrôleur flou, en général, implique les tâches suivantes :

1. Définir les ensembles flous et les fonctions d'appartenance des signaux d'entrée ;

2. Définir la base de règles qui détermine le comportement du contrôleur en fonction des actions de contrôle ; effectuées à l'aide des variables linguistiques décrites dans la tâche précédente.

La première étape du processus consiste à partitionner l'espace d'état de chaque variable d'entrée en différents ensembles flous par le biais de l'appartenance. Chaque ensemble flou est associé à un terme linguistique tel que "faible" ou "élevé".

La fonction d'appartenance, notée $\mu_{L_i^n}(x)$, quantifie le degré d'appartenance d'un signal d'entrée x à l'ensemble flou L.

Dans cette étude, les fonctions d'appartenance, décrites dans les Figures 3.3, 3.5 et 3.4 sont considérées trapézoïdales parce qu'ils sont simples et se sont montrés suffisants dans un certain nombre d'applications.

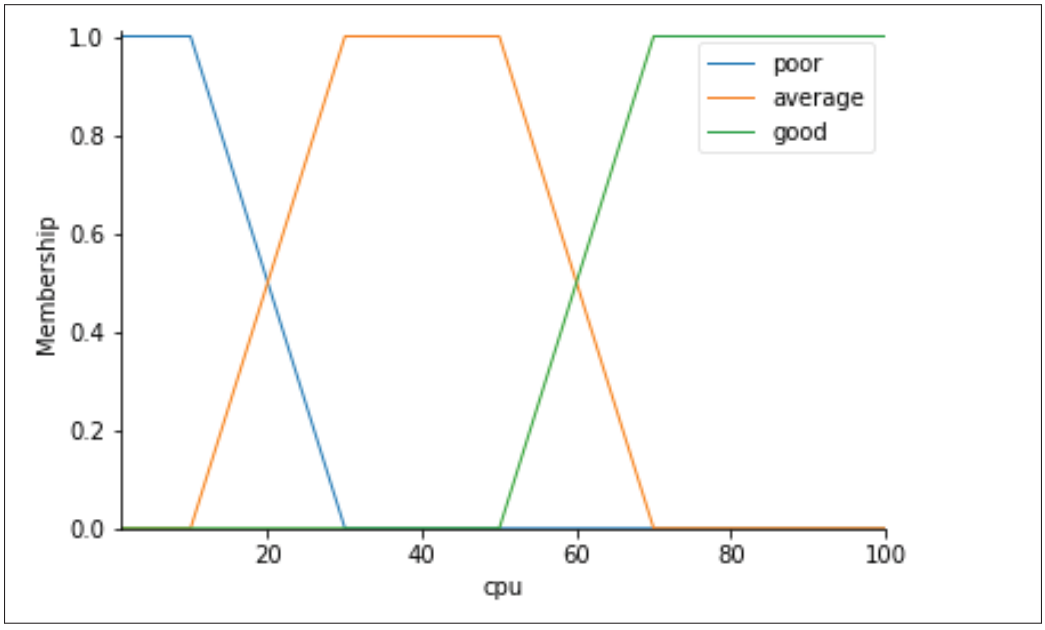

Figure 3.3 Fonction d'appartenance floues de CPU

Afin de conserver la lisibilité de la règle, nous utilisons une partition floue forte, qui vérifie par l'équation 3.1 :

$$\forall X_n, \sum_{j=1}^{N} \mu_{L_n}^{j}(X_n) = 1 \tag{3.1}$$

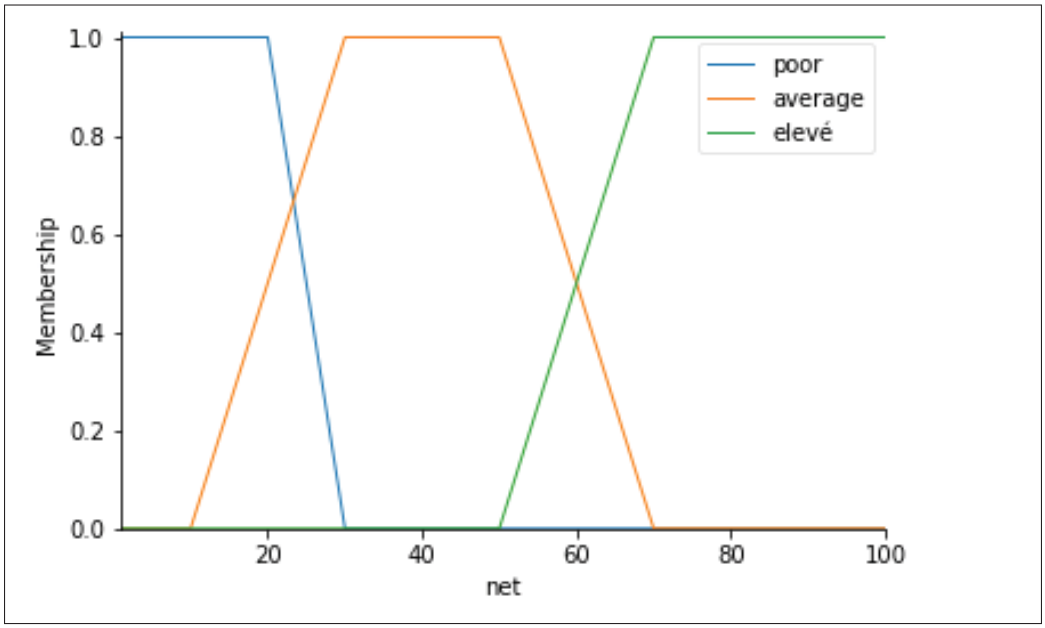

Figure 3.4 Fonction d'appartenance floues de net

L'étape suivante consiste à définir le mécanisme d'inférence pour le contrôleur. Ici, nous devons définir les politiques d'élasticité en termes de règles :

"Si (cpu est élévé) et (net est faible) et (mem est faible) alors (sa = 2)" par exemple, où la fonction de sortie est une valeur constante qui peut être un entier dans (0,1,2,3,4,5,6,7,8) qui est associée à l'action d'adaptation appropriée .

Dans ce travail, nous avons conçu 9 actions, aussi aucune connaissance a priori pour définir de telles règles n'est supposée. En particulier, FQL tente de trouver le conséquent L pour les règles

Une fois que le contrôleur flou a été désigné, l'exécution du contrôleur se compose de trois étapes :

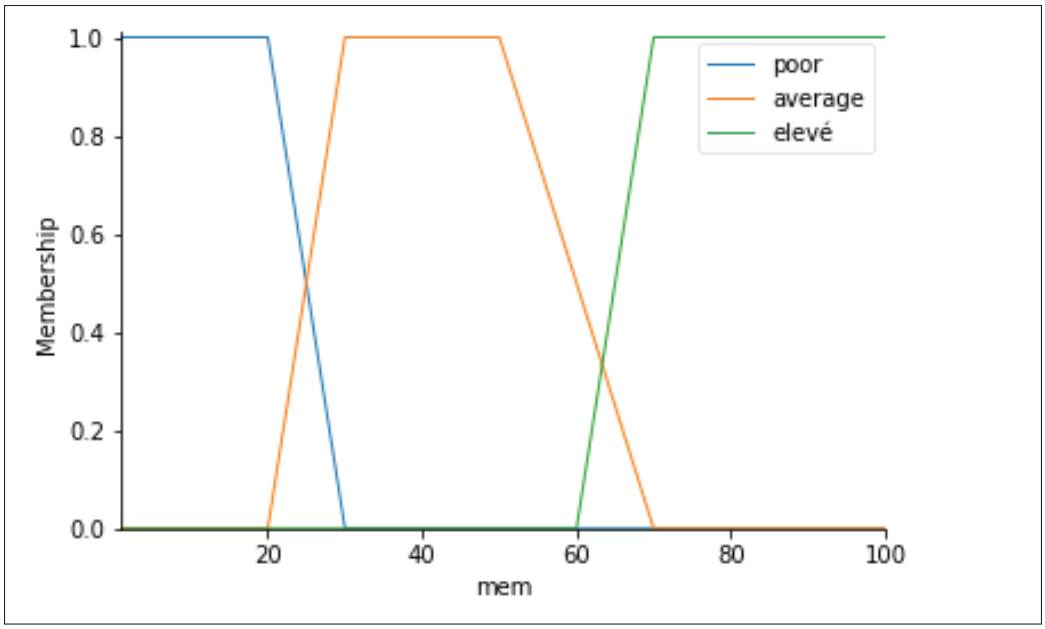

Figure 3.5 Fonction d'appartenance floues de mem

- **la fuzzification des entrées :** le Fuzzificateur transforme les données numériques en données floues à l'aide de fonctions d'appartenance ;
- **raisonnement flou :** le moteur flou traite les informations sur la base d'un ensemble de règles floues et déduit des actions floues ;
- **défuzzification de la sortie :** le défuzzificateur convertit les résultats en mode numériques et active une action d'adaptation.

Dans la couche de fuzzification des FIS, chaque fonction d'appartenance (μ_L) associe une composante de l'état au degré d'appartenance à un ensemble flou correspondant à une étiquette donnée.

Soit R_x qui désigne l'ensemble de toutes les règles de la couche d'évaluation des règles de FIS. L'appartenance d'un vecteur d'état x, ou le degré de vérité dans la terminologie de la logique floue (représenté par α),par rapport à la i ème règle, i $\in R_x$ est défini comme le produit des fonctions d'appartenance correspondantes de la règle 3.2 :

$$\alpha_i(x) = \prod_{n=1}^{N} \mu_{L_i^n}(x) \tag{3.2}$$

3.2.2 Algorithme de Q-learning floue

Dans l'algorithme FQL 3.1, nous avons une sélection d'actions à deux niveaux.

3.2.2.1 Une sélection des actions locales

Un ensemble d'actions est sélectionné en fonction d'une politique d'exploration/exploitation (EE). Une politique d'EE permet à l'agent d'explorer des actions non testées pour acquérir plus d'expérience, et combiner cela avec l'exploitation des actions déjà connues pour leur succès afin d'assurer un niveau élevé de récompense.

La méthode ε -greedy (Tokic & Palm, 2011) est utilisée comme politique d'EE pour nos études expérimentales. Avec la méthode ε-greedy, à chaque étape temporelle, l'agent sélectionne une action aléatoire avec une probabilité fixe de 1 -ε . ε est généralement choisie proche de 1 et inférieure à 1, au lieu de sélectionner avec avidité une des actions optimales apprises par rapport à la table q :

$$\begin{cases} avec\ Prob\ 1-\varepsilon & \forall i \in R_i : a_i^l = argmax_{k \in K} q(L_i, a_i^k) \\ avec\ Prob\ \varepsilon & \forall i \in R_i : a_i^l = random_{k \in K}\ (a_i^k) \end{cases} \tag{3.3}$$

Avec a_i^l est l'action locale selectionnée de la i eme régle

3.2.2.2 Une sélection d'actions inférées

Une action nommée pour le vecteur d'entrée x est sélectionnée dans l'ensemble des actions locales comme suit :

$$a = \max_{i \in R_x} \alpha_i(x) a_i^l \tag{3.4}$$

Afin de mettre en évidence la dépendance du temps, il est nécessaire de préciser l'indice de temps dans l'équation. L'approximation de la valeur de la qualité de l'état x_t est calculée sur la base de l'équation suivante :

$$Q(x_t, a) = \sum_{i \in R_{x_t}} \alpha_i(x_t) \times q_t(L_i, a_i^l) \tag{3.5}$$

Après avoir pris une mesure, le système passe à l'état suivant x_{t+1} et observe la récompense r_{t+1}. La valeur de l'état du vecteur d'entrée x_{t+1} est calculée comme suit :

$$V(x_{t+1}) = \sum_{i \in R_{x_{t+1}}} \alpha_i(x_{t+1}) \times \max_k q_t(L_i, a_i^k) \tag{3.6}$$

Sur la base des équations précédentes, l'erreur de différence de temps (ΔQ) est calculée comme suit :

$$\Delta Q = r_{t+1} + \gamma V(x_{t+1}) - Q(x_t, a) \tag{3.7}$$

Avec γ est le facteur de réduction,

Finalement, la fonction q est actualisée pour chaque règle activée $i \in R_{x_t}$ selon la règle :

$$q_{t+1}(L_i, a_i^l) = q_t(L_i, a_i^l) + \beta \alpha_i(x_t) \Delta Q \tag{3.8}$$

Où β est le taux d'apprentissage.

Algorithme 3.1 Fuzzy Q-Learning

Require γ , β and ε (discount factor, learning rate and ε-greedy respectively)
Initialize q-values :
q[i,j] = 0, 1 < i < N , 1 < j <J
Observe the current state s_t.
Select an action for each fired rule :
a_i = argmax$_k q[i,k]$ *avec Prob* $1-\varepsilon$
a$_i = random_{k\in K}\ (a_i^k)$ *avec Prob* ε
Select the rule i^* for which $\alpha_{i^*}(s_t)$ (computed using equation 3.2) is maximal and set the action a to $a(i^*)$
Update the Q function using :
Q$(s_t, a) = \sum_{i\in R} \alpha_i(s_t) \times q_t(i, a_i)$
Perform the action a and receive the reward r_{t+1}, and the new state s_{t+1}.
The value of the next state s_{t+1} is calculated by :
V$(s_{t+1}) = \sum_{i\in R} \alpha_i(s_{t+1}) \times \max_k q(i,k)$
The error signal is calculated as the following formula :
$\Delta Q = r_{t+1} + \gamma V(s_{t+1}) - Q(s_t, a)$
Update the q-values :
q[i,a$_i$] $= q[i,a_i] + \beta\, \alpha_i(s_t) \Delta Q$
Repeat the process for the new state until it converge

3.2.3 La fonction de Récompense

À ce stade, nous avons montré la conception d'un contrôleur flou pour l'auto-adaptation de notre architecture où les politiques d'élasticité sont fournies par les utilisateurs au moment de la conception. Dans cette section, nous introduisons un mécanisme pour apprendre les politiques au moment de l'exécution, permettant l'évolution des connaissances.

Le contrôleur reçoit les valeurs courantes de cpu, mem et net qui représentent l'état du système. Le signal de contrôle s_a indiqué par la figure 3.2, représente l'action a que le contrôleur prend à chaque boucle. Ce dernier doit essayer de sélectionner les actions dans le passé qui ont produit de récompenses positives en garantissant une grande précision dans la sélection de la meilleure action d'adaptation qui convient au contexte d'exécution actuel.

Pour ce faire, nous devons examiner chaque état s de notre l'architecture, nous considérons un ensemble de valeurs de contexte $c \in C$ mesurant les matrices de l'environnement opérationnel.

telles que : CPU, mémoire, et réseau. Ces informations sont données par le Moniteur selon la figure 3.2

Le service de détection des anomalies qui est représenté par le prédicteur dans la figure 3.2, calcule le score d'anomalie (*as*) et la probabilité d'anomalie (*al*) de toutes les valeurs de contexte actuelles *c*. Le détecteur HTM (développé par Numenta et les auteurs) est basé sur la mémoire temporelle hiérarchique (HTM), une technologie d'intelligence artificielle inspirée de la structure du néocortex (Hawkins & Blakeslee, 2004).
Considérant un flux de données en temps réel x_{t-2}, x_{t-1}, x_t, x_{t+1}, x_{t+2}, l'algorithme modélise les séquences temporelles de ce flux. À chaque instant t, le HTM effectue de multiples prédictions pour x_{t+1}. Au moment t+1, ces prédictions sont comparées aux valeurs réelles pour déterminer un score d'anomalie instantané entre 0 et 1. Si l'une des prédictions sont significativement différentes de x_{t+1}, le score d'anomalie sera égal à 1. Par contre, si l'une des prédictions est exactement égale à x_{t+1}, le score d'anomalie sera 0. Le système conserve la moyenne et la variance de la distribution récente des scores d'anomalie. À chaque étape, il indique la probabilité que le score d'anomalie actuel provient de la distribution normale correspondante. Le score de probabilité de l'anomalie est soumis à un seuil pour détecter l'anomalie finale.

La probabilité d'anomalie définit avec précision le degré d'anomalie de la métrique actuelle par rapport à la distribution des valeurs acquises par le service de détection des anomalies.

Cela signifie que si le contexte c1 fait référence à une valeur d'utilisation du CPU de 70 % et et que la valeur de probabilité d'anomalie al(cpu) est égale à 1, alors notre algorithme a une probabilité plus élevée de sélectionner une action d'adaptation qui ajouterait un nouveau nœud au cluster afin de réduire la charge du CPU et de maintenir le cluster dans l'état souhaité.

Notre objectif est de fournir à l'algorithme d'apprentissage Q flou une valeur évolutive qui peut être utilisée pour attribuer une pondération W(s, c) pour toutes les valeurs de contexte c trouvées dans l'état s, cette valeur est calculée en utilisant l'équation 3.9 .

Dans ce cas, la récompense sera calculée selon l'équation 3.10 qui prend la Softmax de toutes les valeurs calculées pour toutes les métriques.

Ce processus calcule la récompense cumulée de l'exécution d'une action jusqu'à ce que le cluster atteint un état optimal, où "état optimal" signifie l'état qui rapporterait la plus grande récompense au FQL. Si la récompense est proche de zéro, cela indique que l'action n'est pas efficace, une récompense négative est considérée comme une punition pour l'agent d'apprentissage.

$$W(al_n, C_n) = \sum_{i=1}^{n} al_i.c_i.as_i \tag{3.9}$$

$$R(s,a) = 1 - softmax(W(al_n, C_n)) \tag{3.10}$$

3.3 Réseau Q profond flou (FDQN)

Comme mentionné dans la section précédente, la méthode Q-Learning ne peut pas gérer un grand nombre d'états, le réseau Q profond (DQN) est l'une des méthodes les plus utilisées pour résoudre ce problème.

Dans cette section, nous décrivons l'idée que nous proposons pour améliorer les performances du FQL, Cette approche est le FDQN, qui est basé sur l'utilisation de la logique floue pour générer des données et de l'utilisation du réseau profond.

Le modèle de notre approche proposée pour le processus d'auto-adaptation en périphérie de réseau est illustré à la figure 3.6

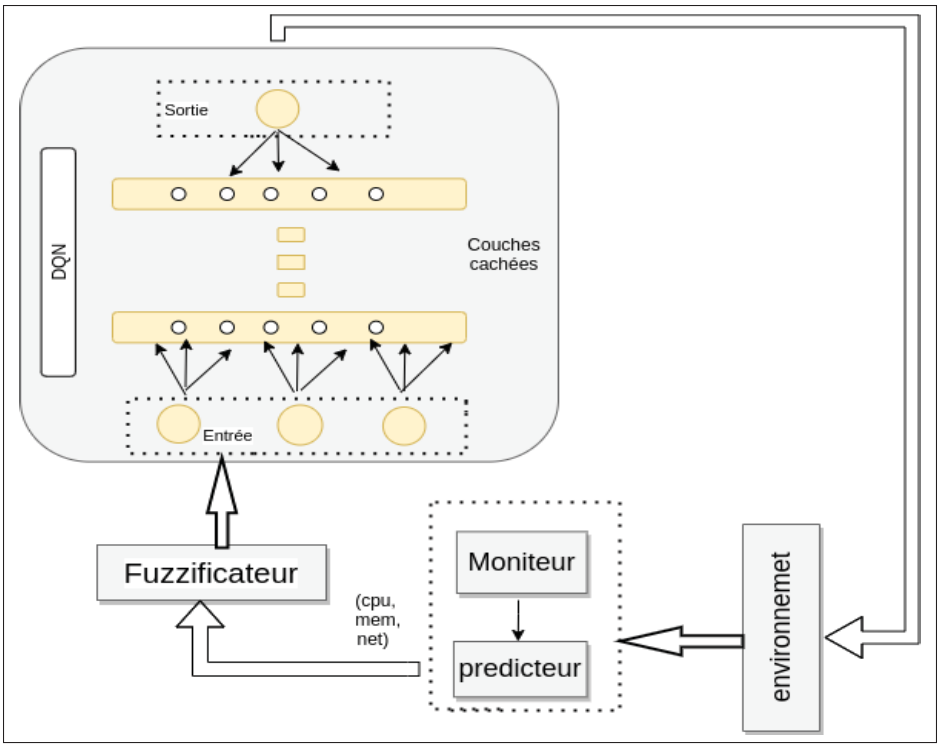

Figure 3.6 Architecture du modèle 2

3.3.1 Fuzzificateur

Ce module est responsable de la conversion des valeurs numériques (cpu, mem, net) en valeurs de degré de vérité selon les fonctions d'appartenance prédéfinies mentionnées dans la section précédente. Le DQN prend les valeurs transformées en entrée et en déduit les actions appropriées.

3.3.2 Les composants de FDQN

Dans cette section, nous présentons les éléments de base de FDQN, notamment l'espace d'état, l'espace d'action, la fonction de récompense et l'architecture du modèle DQN :

3.3.2.1 Espace d'état

Comme dans l'algorithme FQL, chaque valeur de chaque variable d'entrée (cpu, mem, net) a un degré d'appartenance. Après avoir flouté cette valeur, on a un vecteur dont la somme est =1 (comme dans l'équation 3.2).

Pour toutes les variables, on a alors un produit de vecteurs. Le résultat de ce produit est un vecteur de dimension L (F_1, F_2, F_3, ...F_L), où L désigne le nombre total de règles.

3.3.2.2 Espace d'action

l'espace d'action contient les actions suivantes :

- **créer :** créer un nouveau nœud, ou créer un nouveau conteneur (lancer un conteneur sur un nœud et allouer des ressources) ;
- **promouvoir :** promouvoir un manager à partir d'un travailleur ou l'inverse ;
- **ajouter :** ajouter des nœuds ou des réplicas à notre cluster ;
- **terminer (stop/Kill) :** il met fin à la vie d'un conteneur ou d'un nœud pour atténuer la pénurie de ressources si le conteneur ou le nœud s'arrête ;
- **rien faire :** le comportement observé ne reflète pas un défaut. Il indique une maintenance ou une mise à jour régulière du système. Dans cette situation, aucune action de récupération n'est appliquée.

3.3.2.3 Modèle DQN

Comme l'entrée du modèle DQN est un vecteur de taille L, il est nécessaire de disposer d'une architecture convenable pour le gérer. L'architecture DQN proposée est présentée dans la figure 3.7 :

La deuxième couche utilisée est une couche cachée 'dense', comprenant 256 unités.

La fonction d'activation RELU est employée pour activer la sortie de la couche dense qui suit.

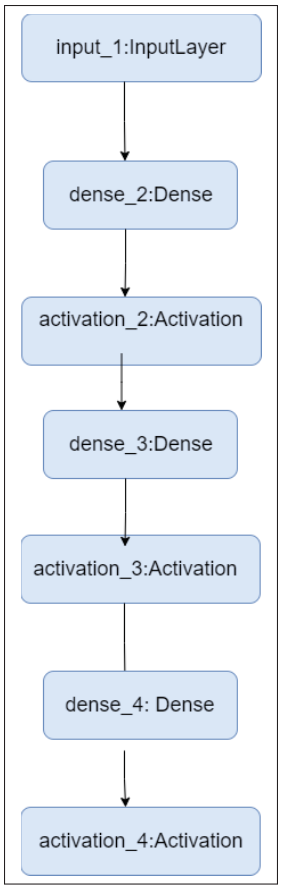

Figure 3.7 Réseau de neurones profonds

La dernière couche du DQN est une couche dense avec une activation linéaire. La sortie de la couche dense-4 est égale à la valeur Q estimée de la valeur d'action associée de l'espace d'action défini par l'agent d'adaptation. Le DQN a un problème de régression, nous avons donc utilisé la fonction de perte de Huber.

La perte de Huber est une combinaison de MSE et MAE, c'est-à-dire qu'elle est quadratique (MSE) lorsque l'erreur est faible, sinon MAE.

Ici, delta est l'hyperparamètre pour définir la marge de MAE et MSE qui peut être itérative pour s'assurer de la valeur correcte de delta. D'après l'équation 3.11 nous constatons que lorsque l'erreur est inférieure à delta, l'erreur est quadratique, sinon elle est absolue.

$$\begin{cases} \frac{1}{2}(y-f(x))^2, & Si \quad |y-f(x)| \leq \delta \\ \delta|y-f(x)| - \frac{1}{2}\delta, & Sinon \end{cases} \tag{3.11}$$

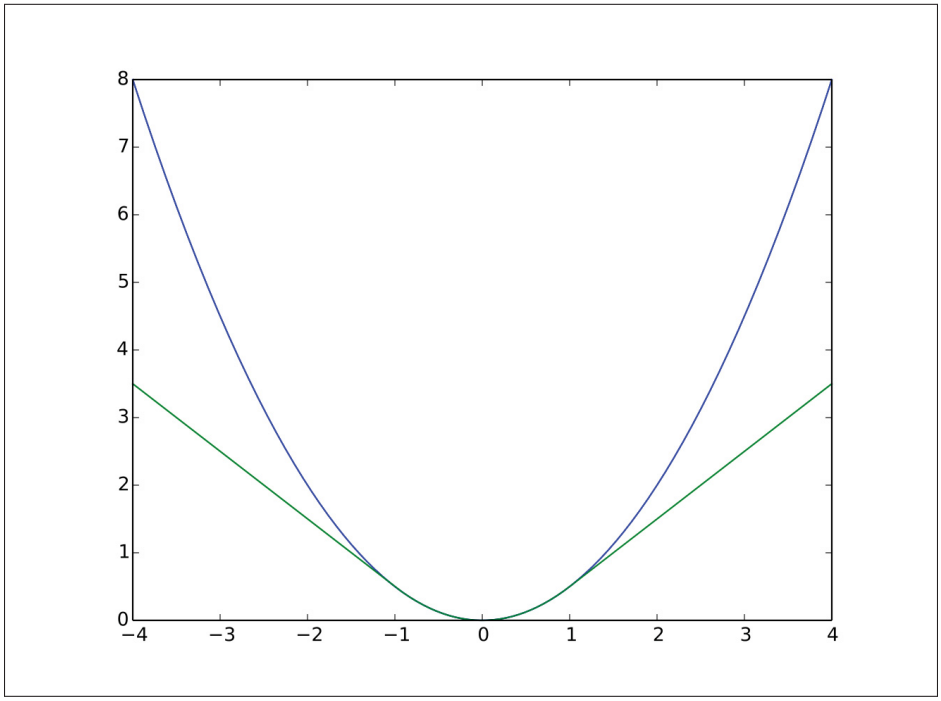

Figure 3.8 Perte du Huber

La courbe verte de la figure 3.8 représente la perte dans le cas de $\delta = 1$, la courbe bleue montre la perte par erreur quadratique en fonction de y-f(x)

Le pseudo-code du modèle Fuzzy DQN (en tant qu'agent) utilisé dans OpenAI Gym est représenté ci-dessous :

3.4 Design de modèle

L'architecture proposée, comme le démontrent nos modèles selon la figure 3.2 et la figure 3.6 est composée d'un module de gestion des défaillances, qui vise à détecter automatiquement les défaillances en observant les données collectées, et en prédisant l'occurrence future d'une

Algorithme 3.2 Agent Fuzzy DQN

```
def agent() :
”””””””””
 GLOBAL VARIABLE :
 env : Environment,
 agent : The agent
 fis : Fuzzy Inference System
”””””””””
s = env.reset() : reset environment
s = fis.fuzzyfy(s) : Fuzzify the input
total reward = 0
While not done :
a = agent.act(s) : interact with the environment
s’, r, done, info = env.step(a) : Obtain the next state, reward
s’= fis.fuzzyfy(s’) : Fuzzify the input
agent.observe((s,a,r,s’)) : Store states to experience buffer
agent.replay() : Train the DQN
s=s’
total reward + = r
return total reward
```

anomalie. Un modèle de récupération, qui applique une ou plusieurs actions de récupération en fonction du type d'anomalie identifié Par exemple, lors d'une saturation du CPU, notre modèle décide d'ajouter un nouveau nœud au cluster pour réduire la charge du CPU et maintenir le cluster dans l'état souhaité. .

Nous nous inspirons des étapes de l'architecture auto-adaptative de la boucle de contrôle à rétroaction (MAPE-K), qui est considérée comme un gestionnaire autonome composé de plusieurs éléments connectés qui interagissent pour effectuer la gestion automatique d'un système.

Dans notre travail, MAPE-K est utilisé pour :

- **la surveillance :** ce module fournit la collection régulière de métriques détaillées sur le cluster, les nœuds, les services et les conteneurs, y compris ;
  - l'utilisation du cpu ;
  - la mémoire ;

  - le réseau.

- **la prédiction des anomalies :** ce modèle est responsable de la lecture des observations collectées et calcule la probabilité d'anomalie de l'observation actuelle par rapport à l'observation historique de l'architecture ;
- **la planification de l'adaptation :** basé sur le comportement anormal détecté qui a à la fois un score d'anomalie élevé et une probabilité élevée. Ce module est chargé de sélectionner la ou les actions d'adaptation après avoir calculé la valeur Q de toutes les actions ;
- **l'exécution de l'adaptation :** après avoir sélectionné les actions appropriées par le module précédent, cette unité exécute l'action qui a la récompense la plus élevée en utilisant nos approches adaptées.

CHAPITRE 4

IMPLÉMENTATION ET RÉSULTATS EXPÉRIMENTAUX

Un de ces aspects importants d'une architecture de microservices auto-adaptative est sa possibilité de surveiller en continu l'environnement fonctionnel, de percevoir et analyser les changements contextuels.

Grâce à cette surveillance, elle détectera également et fournira une politique raisonnable d'auto-adaptation, d'auto-réparation et d'auto-réglage des ressources informatiques afin que ces ressources puissent s'adapter dynamiquement à tout changement brutal de son milieu opérationnel.

Dans ce chapitre, nous détaillons la configuration de notre environnement, les scénarios d'anomalie qui sont injectés dans les différents composants du système, ainsi que l'évaluation de nos deux approches d'apprentissage par renforcement.

4.1 Implémentation de l'architecture

Afin de valider les concepts présentés dans notre travail, nous avons développé un prototype fonctionnel d'architecture de microservices fonctionnant dans un Docker swarm.

Le cluster a un seul leader soutenu par des nœuds managers et workers.Le leader est un nœud manager qui est responsable de la planification des tâches sur les nœuds workers, dans le cas où il y a plusieurs managers, l'un d'entre eux est considéré comme leader. Notre architecture est conçue selon le modèle MAPE-K, comme le montre la figure 4.1. L'architecture met en place :

- **service de surveillance :** ce service permet de collecter en temps réel des métriques détaillées sur les nœuds de cluster, les services et les conteneurs, notamment : l'utilisation du CPU, la mémoire, le réseau (lecture/s) et le réseau (écriture/s) ;

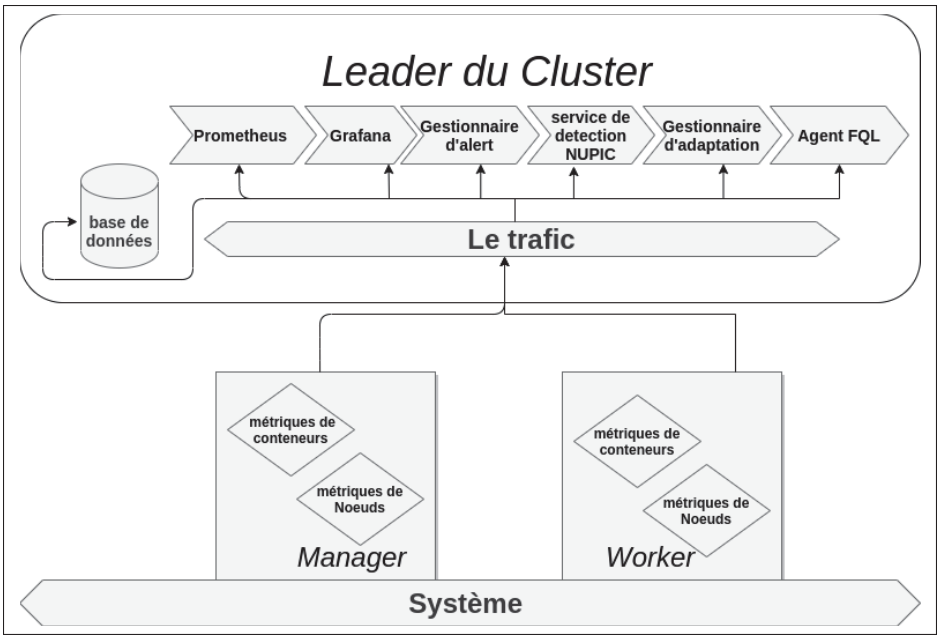

Figure 4.1 Architecture de microservices

- le framework Prometheus [2] est utilisé pour mettre en œuvre la base de données de mesures de séries temporelles pour la collecte et le suivi du contexte en temps réel ;
- les métriques de tous les nœuds du cluster sont collectées à l'aide du service node exporter [3] ;
- la collecte de métriques à granularité fine sur tous les conteneurs en cours d'exécution dans le cluster swarm est effectuée à l'aide du service Cadvisor [4].

- **service de détection d'anomalie :** ce service est chargé de consulter les observations collectées et de calculer le degré d'anomalie de l'observation actuelle par rapport à l'observation du comportement historique de l'architecture. Cette analyse est réalisée en implémentant un service de détection d'anomalie basé sur le framework NUPIC (Ahmad *et al.*, 2017). Les données collectées en temps réel sont transmises au service de détection des anomalies NUPIC, qui offre deux fonctionnalités suivantes ;
  - la détection permanente des comportements anormaux avec une grande précision ;

2. https ://prometheus.io/
3. https ://prometheus.io/docs/guides/node-exporter/
4. https ://github.com/google/cadvisor

- des prédictions concernant les performances de l'architecture basées sur les données historiques collectées. Le service de détection d'anomalies est par ailleurs capable de détecter les événements le plus tôt possible avant que le comportement anormal ne perturbe la fonctionnalité des services en cours d'exécution dans le cluster.

- **service de planification :** une fois qu'un comportement anormal a été détecté et qu'il présente à la fois un score d'anomalie élevé et une probabilité élevée, Ce qui se produit ensuite est :

  - le gestionnaire d'alerte [5] notifie au gestionnaire d'adaptation les changements contextuels qui ont été détectés ;
  - le gestionnaire d'adaptation utilise les états des valeurs métriques , les scores d'anomalie, les probabilités d'anomalie, la contrainte d'architecture(le nombre limité de nœuds), telle que spécifiée par le DevOp pendant le déploiement ;
  - le gestionnaire d'adaptation sélectionne la ou les actions d'adaptation après avoir calculé la valeur Q pour toutes les actions comme expliqué dans la section précedente ;
  - le gestionnaire d'adaptation calcule la meilleure variation de l'adaptation qui a la meilleure récompense en utilisant un de nos deux algorithmes 3.1 et 3.2. Chacun est exécuté séparément comme expliqué dans la section précedente.

  Ce service est implémenté via FDQN/FQL. Ces deux algorithmes ont été réalisés à l'aide du framework Keras-rl [6] et du framework Tensorflow [7].

- **service d'exécution :** une fois les actions d'adaptation sélectionnées, le gestionnaire d'adaptation est chargé d'exécuter ces actions en les déployant dans l'architecture. Afin d'éviter les conflits entre plusieurs politiques d'adaptation, le gestionnaire permet aux actions d'adaptation d'être complètement achevées et vérifiées par le manager de cluster selon le consensus obtenu par RAFT, ensuite il fixe un délai d'attente avant d'initier de nouvelles actions d'adaptation, cette méthode est employée pour préserver l'état du cluster lors de

5. https ://prometheus.io/docs/alerting/latest/alertmanager/
6. https ://keras.io/
7. https ://www.tensorflow.org/

l'auto-récupération. Finalement, le gestionnaire d'adaptation envoie au leader du cluster un ensemble d'instructions d'adaptation, Le leader du cluster et tous les managers du cluster voteront pour l'action d'adaptation selon l'algorithme du consensus (Ongaro & Ousterhout, 2014). Les résultats du vote sont utilisés pour valider et vérifier la possibilité de déployer l'action d'adaptation. Si l'action d'adaptation l'action a gagné le vote, cette action sera exécutée par le leader du cluster, le gestionnaire d'adaptation enregistre la tentative d'adaptation comme étant réussie. Si l'action l'action d'adaptation a perdu le vote, alors le gestionnaire maintient l'état actuel du cluster et enregistre la tentative d'adaptation comme un échec. Dans les deux cas, le gestionnaire enregistre le nombre de tentatives utilisées pour compléter les actions d'adaptation.

Afin d'assurer le transfert du traffic entre tous les services du cluster, un proxy [8] inverse a été déployé.

4.2 Résultats Expérimentaux

Cette section décrit d'une part les configurations et les paramètres nécessaires à l'implementation de nos deux modèles. D'autre part l'analyse des résultats obtenus par l'algorithme FQL et le modèle FDQN.

4.2.1 Configuration Expérimentale

Tableau 4.1 Spécification du matériel et du logiciel utilisés dans l'expérience

Composants	**Configurations**
CPU	2.7 GH Intel Cores i5
Disk Space	477G
Mémoire	8 GB
Systeme d'exploitation	Ubuntu 18.04.3 LTS X64

8. https ://caddyserver.com/docs/proxy

La plateforme d'expérimentation est un cluster installé dans notre machine. Les caractéristiques de notre machine sont présentées dans le tableau 4.1. Elle est composée de deux nœuds : un nœud gestionnaire et un nœud travailleur. . Il est composé de deux noeuds : un noeud manager et un worker.

Les caractéristiques des nœuds sont présentées dans le tableau 4.2.

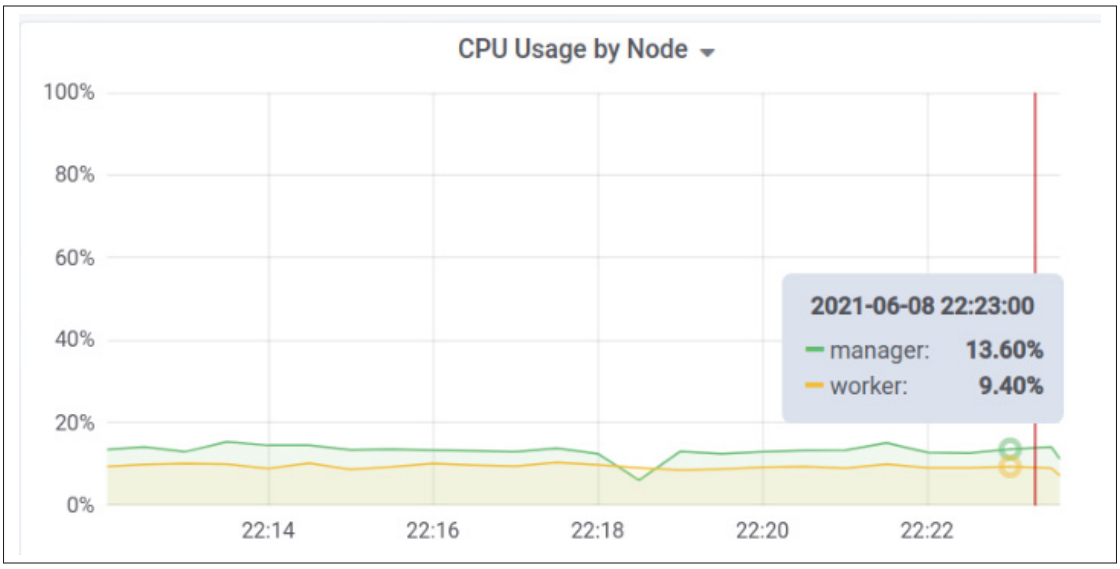

Figure 4.2 Utilisation du CPU par nœud

Les figures 4.2 et 4.3 montrent l'état initial du système en termes de consommation de mémoire et de cpu par nœud respectivement. La figure 4.4 montre les services déployés dans notre cluster lors de l'état initial.

Une configuration nécessaire pour réaliser les expériences ainsi que les paramètres choisis pour l'algorithme FQL et le modèle FDQN.

4.2.1.1 Paramètres de l'algorithme FQL

Concernant les paramètres de l'algorithme FQL, nous choisissons :

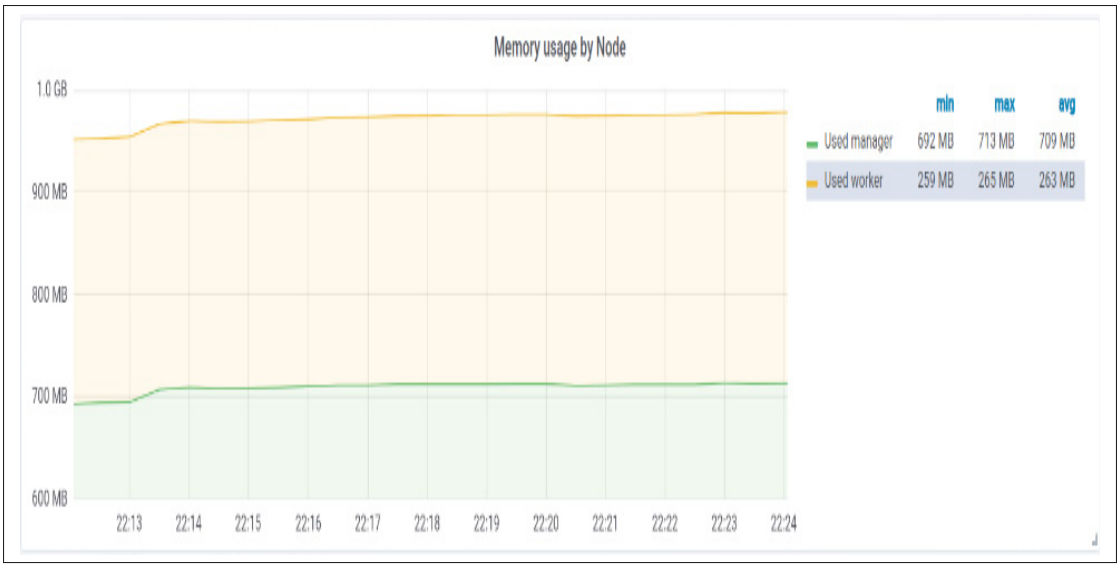

Figure 4.3 Utilisation des mémoires par nœud

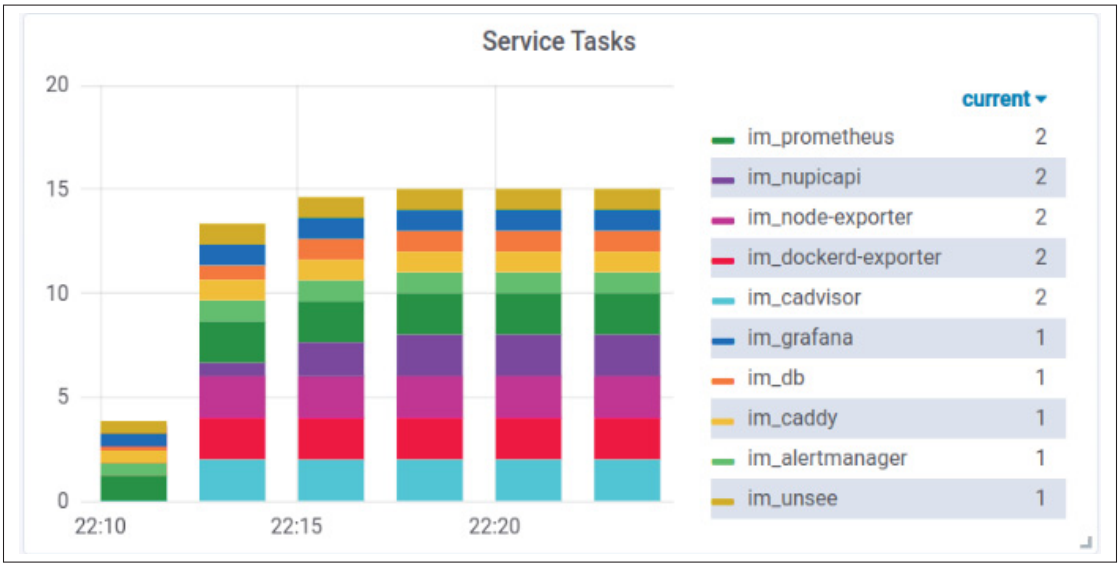

Figure 4.4 Tâches de services

- le facteur de réduction $\gamma = 0.99$ pour mieux prendre en compte les futures récompenses selon l'equation 1.1. Si l'agent doit choisir entre obtenir une certaine récompense maintenant et obtenir une récompense beaucoup plus importante plus tard, la récompense la plus importante est très probablement préférable ;
- le taux d'apprentissage $\beta = 0.1$ pour avoir moins d'impact sur les récompenses récentes ;

Tableau 4.2 Configuration des noeuds du cluster et des conteneurs

Composants	**Configurations**
Nombre de noeuds	2
nombre de Conteneurs	15
Nombre de services	10
Nombre de CPU	1
CPU MHz	1787.632
Outil d'orchestration	Docker

- pour ε - greedy, nous fixons la valeur maximale à 1,0 (exploration complète), La valeur diminue au fil du temps pendant l'apprentissage, passant de 1,0 à à une valeur minimale de 0,1.

L'objectif est de maintenir l'utilisation de nos métriques dans une zone de sécurité qui peut se situer dans une zone [seuil bas, seuil haut]. Dans nos expériences, nous avons choisi la marge de contrôle [30, 70] avec un seuil moyen est de 50. Nos métriques (cpu, mem, net) utilisent des termes linguistiques comme faible, normal, élevé. Chaque ensemble de variables est lié à un terme linguistique tel que défini. Les termes sont obtenus par un ajustement précis au cours des expériences. Par conséquent, il existe 27 états en combinant entièrement les fonctions d'appartenance des variables cpu, mémoire et réseau.

Le processus d'apprentissage sera arrêté lorsque les variations dans le tableau des valeurs de Q sont faibles ou après assez d'itérations d'apprentissage.

4.2.1.2 Paramètres de l'algorithme FDQN

En ce qui concerne les paramètres du FDQN, nous sélectionnons :

- le facteur de réduction $\gamma = 0.99$;
- l'architecture du réseau de neurones se compose de sept couches, comme expliqué dans le chapitre précédent.

Le DQN est implémenté à l'aide d'un Optimiseur Adam, qui est utilisé pour calculer la valeur Q à chaque paire état-action et pour ensuite renvoyer la récompense la plus élevée.

Le processus d'apprentissage sera également arrêté lorsque la moyenne de la récompense atteint un seuil ou après un nombre suffisant d'époques. Dans notre cas, la valeur seuil de 140 a été utilisée au cours d'apprentissage. Les paramètres utilisés sont pris en compte par un ajustement précis lors de la réalisation des expériences.

4.2.1.3 Espace d'action

L'espace d'action contient :

- **action 0 :** rester dans l'état S0 ;
- **action 1 :** maintenir l'état S1 ;
- **action 2 :** ajuster les réplicas (ajustement verticale) ;
- **action 3 :** supprimer des conteneurs ;
- **action 4 :** ajouter des noeuds ;
- **action 5 :** supprimer des noeuds ;
- **action 6 :** promouvoir un nœud manager ;
- **action 7 :** ajouter un manager ;
- **action 8 :** supprimer le cluster et en créer un nouveau.

4.2.1.4 Les règles de récupération

- le nombre max-min de nœuds : afin de fournir des ajustements de reprise dynamiques basés sur une valeur cible pour une métrique spécifique, nous définissons la limite maximale du nombre de nœuds dans le cluster comme le nombre total de nœuds nécessaires pour satisfaire l'allocation de ressources. Selon notre architecture, le nombre maximal de nœuds est égal à 4, tandis que le nombre minimal est de 1 ;

- période de repos (cool off) : une fois que l'adaptation est appliquée et vérifiée par les votes de l'algorithme du consensus, le gestionnaire d'adaptation met en place un délai de repos de 300 secondes avant d'initier toute nouvelle action d'adaptation.

4.2.2 Scénarios de stress

Nous utilisons l'outil Docker Stress [9] pour stresser le CPU et la mémoire en appliquant différentes configurations au niveau du conteneur et du nœud. En outre, nous utilisons l'image de docker Apache Benchmark [10] pour provoquer la congestion du réseau.

4.2.2.1 Stress du processeur

Le tableau 4.3 montre que le conteneur est stressé de manière à consommer 100 % des CPU à l'aide de la commande '–cpu' qui stresse un '–cpu 1' ou plusieurs cpu '–cpu 3', avec '–timeout 1080s' pour gérer la durée du stress. Le paramètre '–vm bytes' permet d'allouer un nombre d'octets pour la mémoire virtuelle par VM. La valeur par défaut de '–vm bytes' est de 256 Mo.

Par exemple, la commande : 'docker run –rm –name cpu2256 -it progrium/stress –cpu 3 –io 3 –vm 3 –vm-bytes 256MB –timeout 1080s' signifie que nous extrayons l'image progrium/stress, créons un conteneur avec le nom cpu2256, exécutons 3 VMs, limitons la mémoire du conteneur à 256 MB, et activons 3 CPUs en même temps pour la mesure de la pression.

Nous configurons et exécutons les commandes du Tableau pour chaque conteneur afin de créer différentes pressions.

4.2.2.2 Fuite de mémoire

Au cours de l'expérience, l'utilisation de la mémoire disponible dans le composant est largement augmentée sur une période de temps relativement courte. Si toute la mémoire est occupée,

9. https ://hub.docker.com/r/progrium/stress/
10. https ://hub.docker.com/r/jordi/ab

Tableau 4.3 Commandes de stress du CPU

Commandes de stress docker
docker run –rm –name cpu128 -it progrium/stress –cpu 1 –io 1 –vm 1 –vm-bytes 128M –timeout 1080s
docker run –rm –name cpu2256 -it progrium/stress –cpu 3 –io 3 –vm 3 –vm-bytes 256MB –timeout 1080s

une erreur de mémoire est générée et le statut du composant est déclaré comme une anomalie. Nous utilisons un script qui continue à allouer de la mémoire jusqu'à ce qu'il crashe, comme indiqué dans le tableau suivant 4.4.

L'image 'nyxcharon/docker-stress' est celle utilisée pour les tests de stress. Le site python /scripts/mem-fill' est un script python permettant de tester les limites de la mémoire.

Nous employons également un autre script pour allouer 256MB de mémoire virtuelle pour chaque VM, et nous limitons la mémoire disponible pour le conteneur à 512 Mo.

Tableau 4.4 Commandes de stress du mémoire

Commandes de stress docker
docker run nyxcharon/docker-stress python /scripts/mem-fill
docker run –rm –name stress –memory 512m stress –vm 3 –vm-bytes 256M –timeout 1080

4.2.2.3 Congestion du réseau

Un composant est surchargé par une grande quantité de requêtes qui provoque une perte aléatoire de paquets par le réseau. La saturation du réseau se manifeste par plusieurs scénarios tels que le retard dans la mise en file d'attente, la perte de paquets ou le blocage de nouvelles connexions.

Dans un réseau encombré, le temps de réponse ralentit et le débit du réseau diminue. Nous utilisons Apache Benchmark pour émuler les problèmes de réseau.

Le tableau 4.5 présente le code que nous utilisons pour exécuter l'image docker Apache Benchmark 4.6.

Tableau 4.5 Exécution d'Apache Benchmark sur Docker

Commande d'exécution d'Apache Benchmark
docker run –rm jordi/ab -k -c 100 -n 100000 http ://localhost :80/

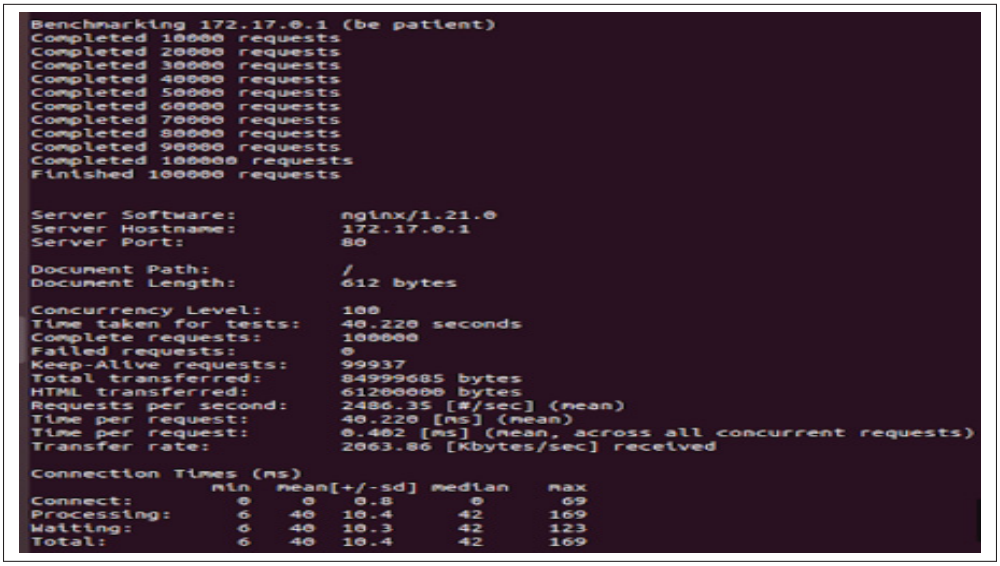

Figure 4.5 Terminal d'Apache Benchmark

4.2.3 Validation

Deux expériences différentes ont été réalisées pour évaluer l'efficacité de l'utilisation de nos deux algorithmes dans le cadre de l'adaptation dynamique. Pour chaque expérience, les métriques suivantes sont utilisées pour évaluer les performances des algorithmes : la récompense totale obtenue par itération, et le temps d'adaptation en secondes, qui mesure le temps nécessaire à chaque algorithme pour terminer la sélection des actions ainsi que le nombre total d'étapes effectués par l'algorithme à chaque itération afin de se retrouver dans un état stable.

Ces métriques sont très importantes à considérer, car certains algorithmes restent coincés dans un état en croyant qu'il offre la récompense maximale, ce qui prolonge le temps d'adaptation et ralentit la transition vers un état optimal.

4.2.3.1 Phase d'apprentissage

Dans le premier scénario, nous avons exécuté un test de stress dans le manager de cluster jusqu'à ce que son utilisation du CPU atteint 70 %, comme le montre la figure 4.6, ce qui a déclenché une alerte à l'agent d'adaptation.

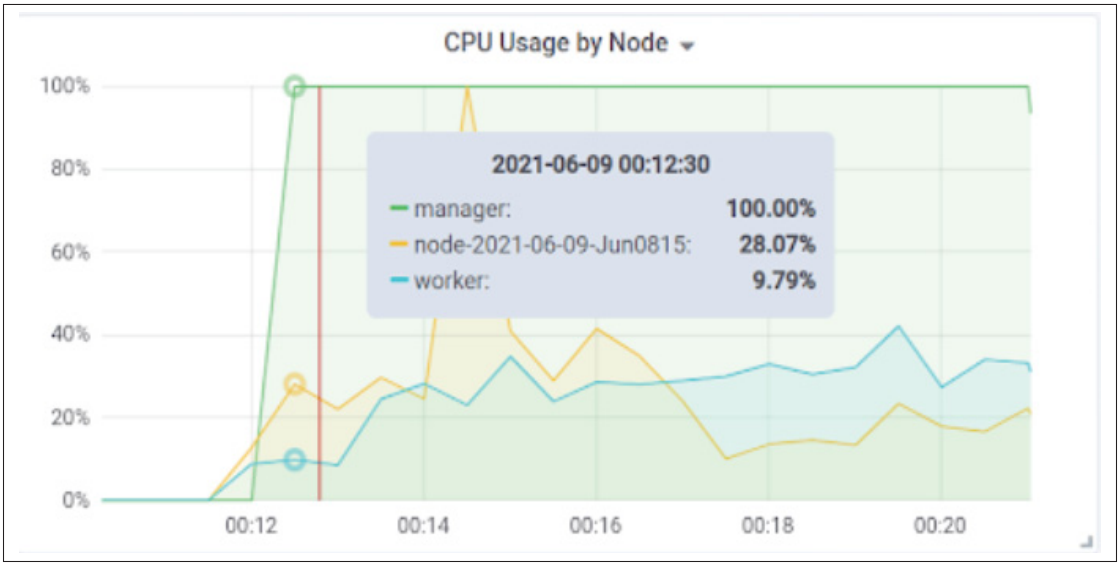

Figure 4.6 Injection de stress au niveau du nœud du manager

La figure 4.7 montre la moyenne des 50 récompenses cumulées récentes de FDQN après l'apprentissage. La figure 4.8 montre la moyenne des 50 récompenses cumulées récentes de FQL après l'apprentissage.

Nous constatons que l'approche FDQN surpasse l'algorithme FQL en termes de performances d'auto-adaptation, car la récompense cumulative de l'algorithme FDQN converge beaucoup plus rapidement que celle de l'algorithme FQL, environ 300 itérations dans la figure 4.7, contre 1200 itérations dans la figure 4.8.

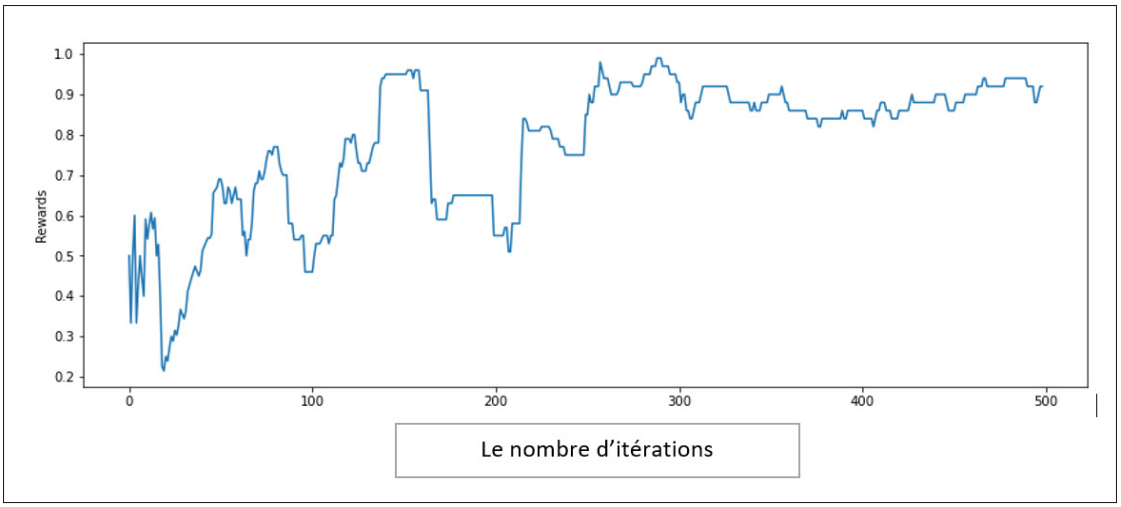

Figure 4.7 Récompenses cumulées moyennes par FDQN

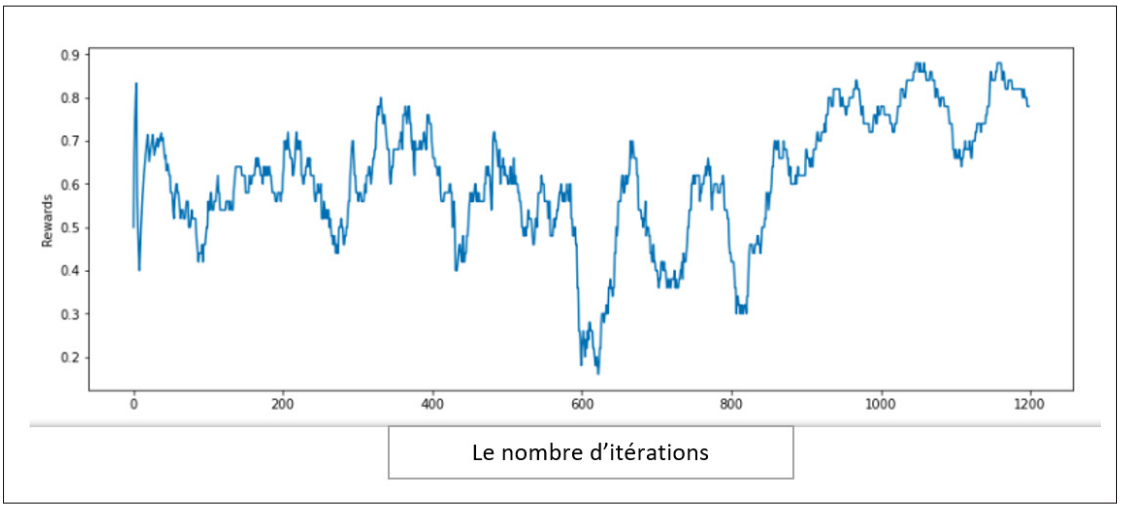

Figure 4.8 Récompenses cumulées moyennes par FQL

En outre, la récompense cumulée pendant l'apprentissage du modèle FQL a beaucoup de fluctuations à cause de la forte variation des valeurs de récompense, alors que celle du FDQN est nettement plus stable.

Ce résultat peut être expliqué en observant le temps d'adaptation/durée d'exécution de toutes les actions par itération dans la figure 4.9.

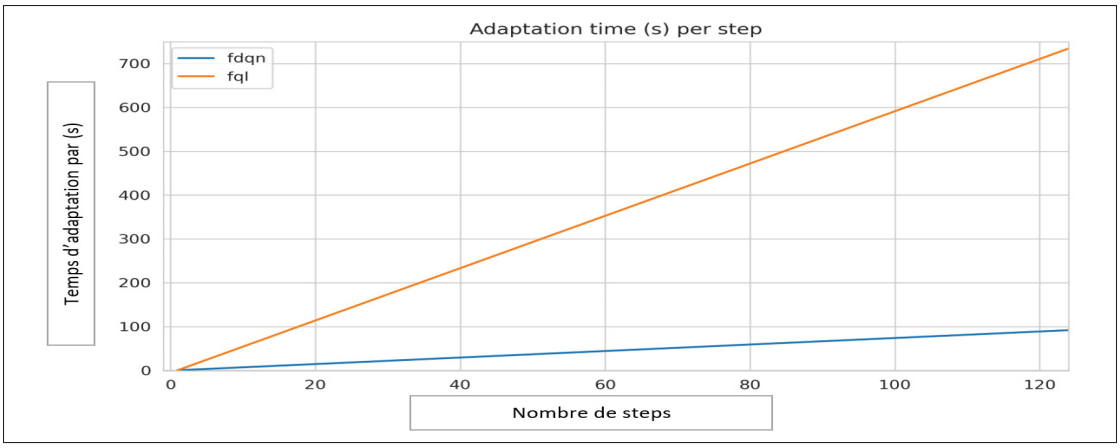

Figure 4.9 Durée d'adaptation(s)

FDQN a convergé avant 100s alors que FQL a dépassé 700s. Donc FDQN réussit à converger plus rapidement et à effectuer l'adaptation en moins de temps que le FQL .

Le FQL a besoin de plus de temps pour effectuer l'action d'adaptation, car il a besoin de plus de temps pour s'adapter aux changements trouvés durant l'observation. Il doit alors effectuer de nombreuses actions dans de nombreuses itérations jusqu'à ce qu'il puisse amener l'architecture à son état optimal.

4.2.3.2 Phase de test

1. **Adaptation Automatique Horizontal** : afin de vérifier la capacité de nos approches à ajuster dynamiquement le nombre de réplicas d'un service lancé dans le cluster, contre les fluctuations du trafic réseau et pour maintenir un temps de réponse acceptable du service web. Dans ce scénario, nous avons lancé un service web appelé 'nginx' avec un réplicas sur le port 80 dans le nœud manager, nous avons donc limité son nombre à 20 réplicas au maximum et à 1 réplicas au minimum lors de l'adaptation du service. Nous avons ensuite effectué une attaque par 'Apache HTTP server BenchMarking' contre ce service.

   La figure 4.10 montre la manière dont le nombre de réplicas est ajusté proportionnellement à l'utilisation du CPU par notre modéle FDQN. Elle montre qu'à chaque fois que le CPU augmente, le nombre de réplicas est immédiatement augmenté pour gérer les requêtes. En

outre, même lorsque la valeur du CPU devient faible et stable, le nombre de réplicas est immédiatement diminue.

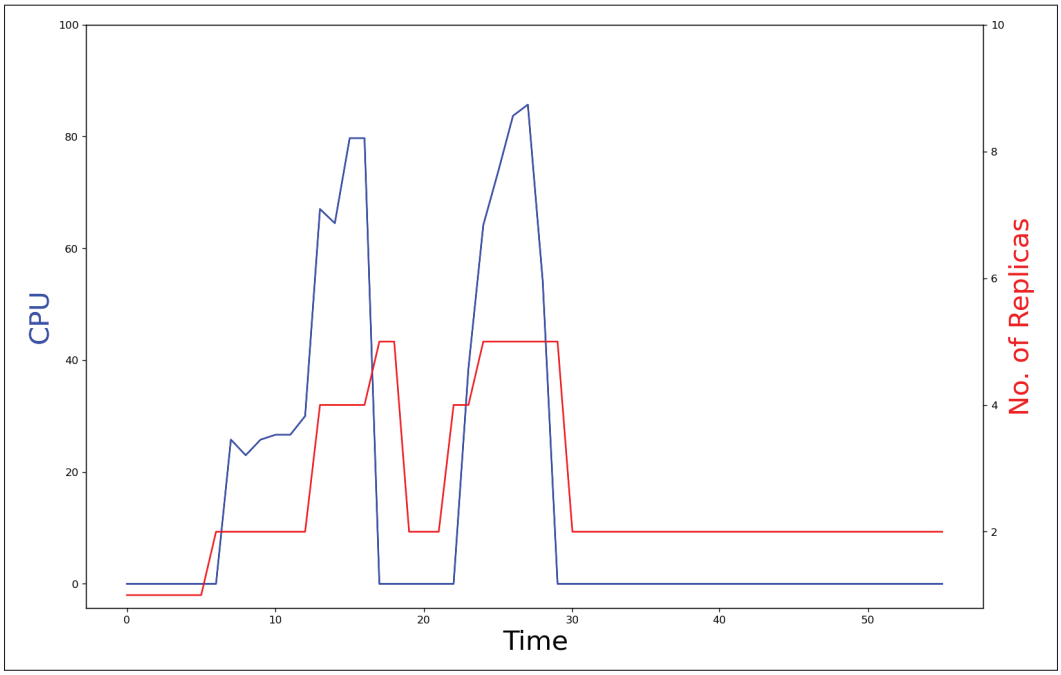

Figure 4.10 Nombre de réplicas en réponse à la CPU par FDQN

La figure 4.11 indique le nombre de réplicas de services web générées en fonction de la charge CPU. La figure montre que lorsqu'il y a une augmentation de la charge, le nombre de réplicas est élevé afin d'assurer un temps de réponse acceptable. Par contre, lorsqu'il y a une baisse soudaine de la charge, le nombre de Réplicas ne réduit pas instantanément ;

2. **Adaptation Automatique Vertical** : afin d'effectuer une auto-adaptation verticale, nos agents effectuent différentes actions comme le montrent les figures suivantes 4.12 et 4.13. D'après ces deux figures, nous remarquons que les actions 7 et 8 sont à 0%, l'action 7 consiste à ajouter un nœud gestionnaire et l'action 8 correspond à supprimer notre cluster et à en créer un nouveau.

   Par contre, nos agents FDQN et FQL choisissent de promouvoir un nœud existant en tant que manager, cette action a été appliquée par FDQN et FQL de 9% et 11% respectivement.

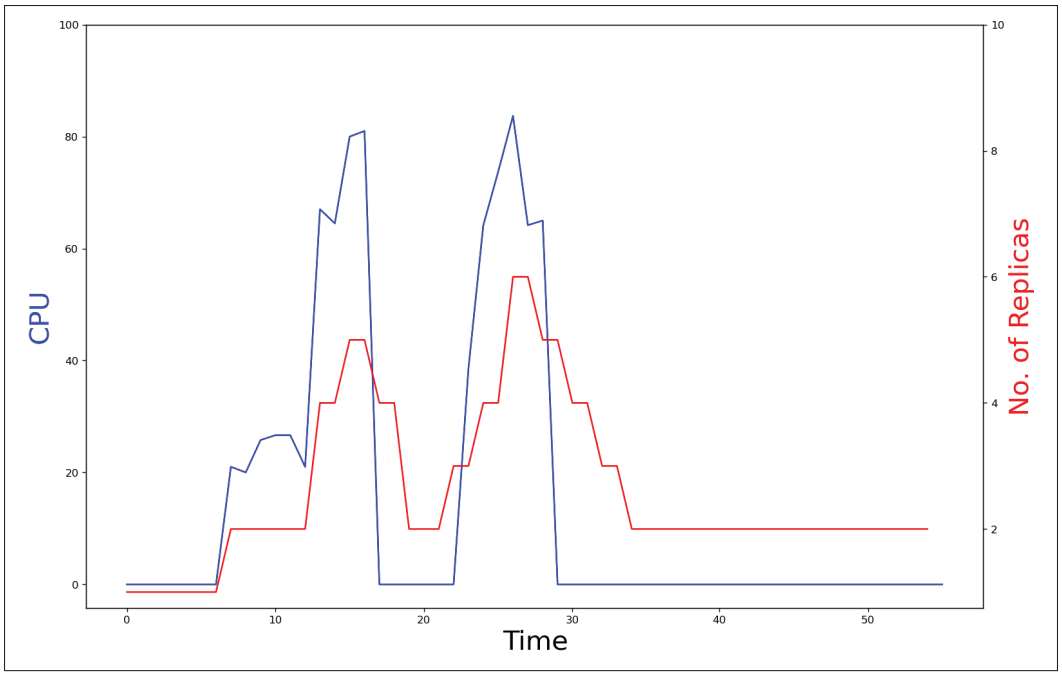

Figure 4.11 Nombre de réplicas en réponse à la CPU par FQL

En termes d'ajout et de suppression de nœuds, FDQN a appliqué l'action 4 à un taux de 13% et l'action 5 à un taux de 26%, tandis que FQL a appliqué ces deux actions à des taux de 23% et 16% respectivement.

Il est clair que FQL préfère ajouter des nœuds pour partager la charge tandis que FDQN ajoute des nœuds tout en favorisant la suppression de ces nœuds dès que l'architecture atteint un état stable. En revanche, l'agent FQL ajoute des nœuds à un rythme élevé et en supprime moins.

Nous observons également une différence dans l'application de l'action 0, elle a été appliquée à 20% par l'agent FDQN, alors que seulement 9% par l'agent FQL.

Nous pouvons constater que nos agents sont capables de supprimer automatiquement les actions non utiles et d'appliquer une variété d'actions afin d'obtenir une architecture stable et optimale ;

3. **Etat final du système** : la figure 4.14 indique la consommation de CPU de tous les noeuds du cluster pendant l'adaptation effectuée par FQL. Cette figure montre que l'agent FQL présente des fluctuations importantes dans l'utilisation du CPU .

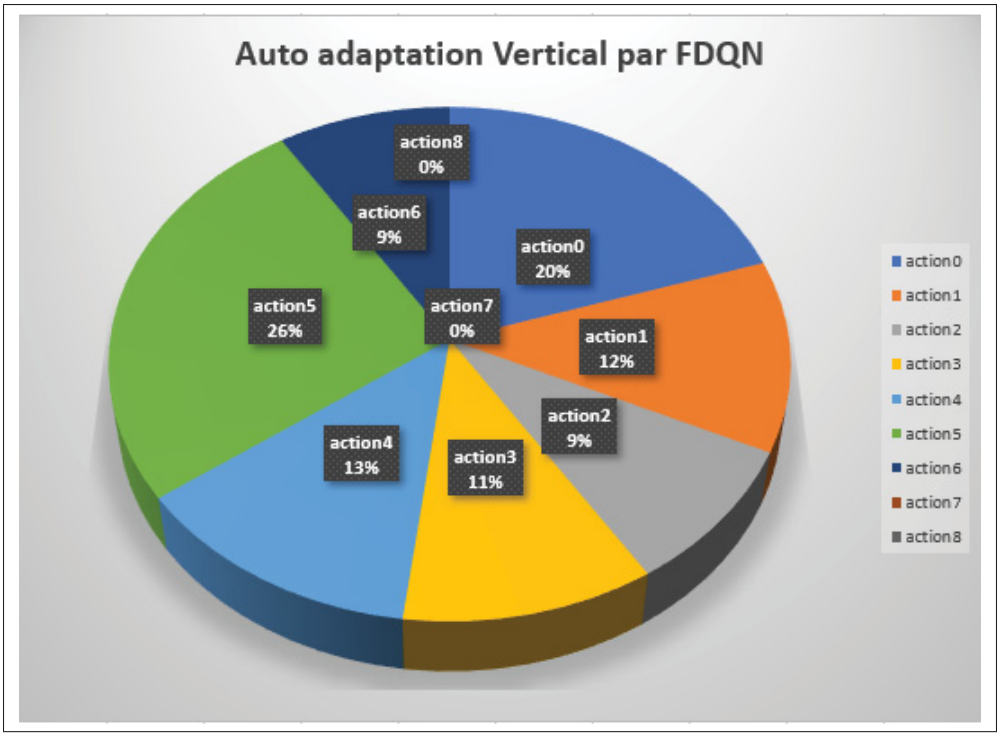

Figure 4.12 Exécution d'actions par FDQN

La figure 4.15 montre la légère fluctuation de la consommation du CPU de tous les nœuds du cluster pendant l'adaptation effectuée par FDQN.

Il est évident, d'après l'analyse de l'utilisation de CPU par noeud, que le gestionnaire d'adaptation FDQN a moins d'impact sur les nœuds du cluster et maintient des performances stables.

De plus, nous constatons que le FDQN a moins d'impact sur l'utilisation de la mémoire comme le montre la figure 4.16 par rapport FQL selon la figure 4.17.

En outre, les figures 4.18 et 4.19 prouvent que nos deux algorithmes sont capables de maintenir une surveillance et une observation permanentes en déployant tous les services nécessaires dans notre swarm.

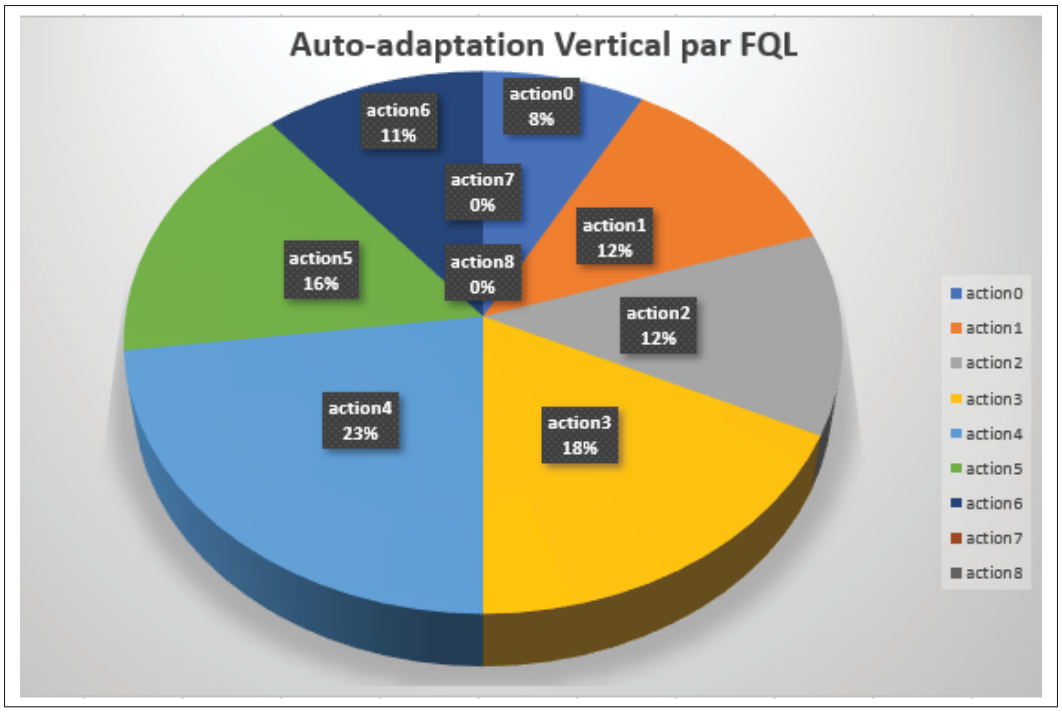

Figure 4.13 Exécution d'actions par FQL

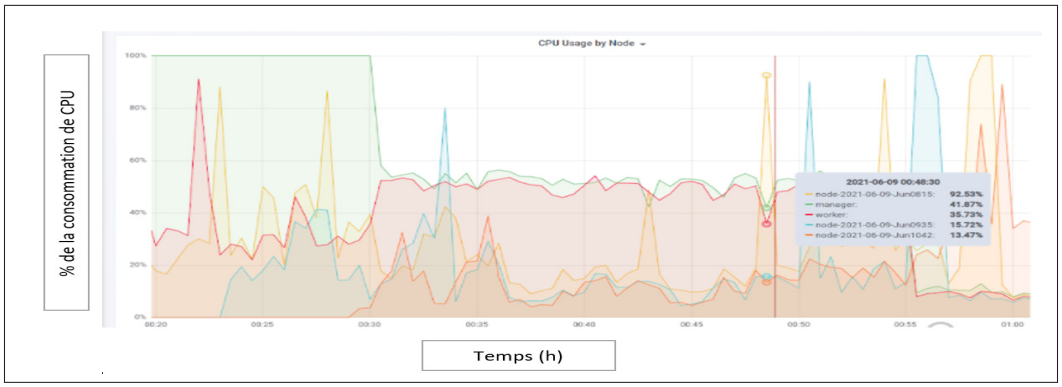

Figure 4.14 Utilisation du CPU par noeud exécutant FQL

4.2.4 Synthèse des résultats

A partir de ces résultats, nous constatons que FDQN a obtenu de bonnes performances sur les métriques étudiées pendant l'apprentissage.

En effet, cette approche a atteint une convergence rapide avec une récompense élevée.

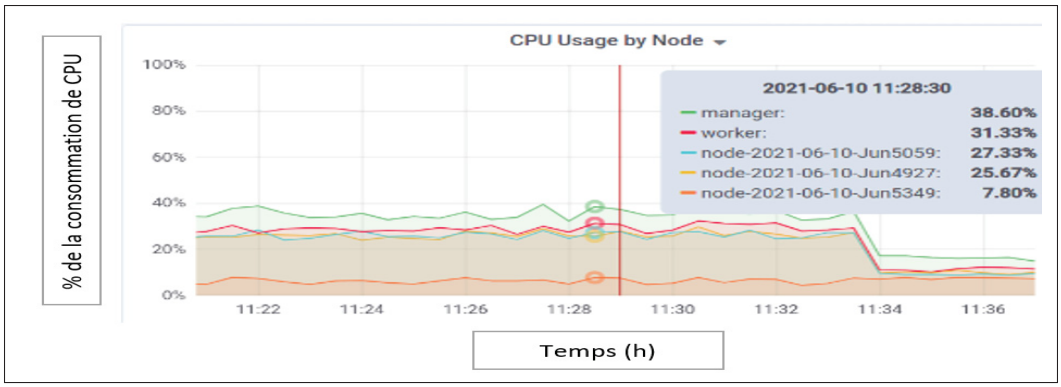

Figure 4.15 Utilisation du CPU par noeud exécutant FDQN

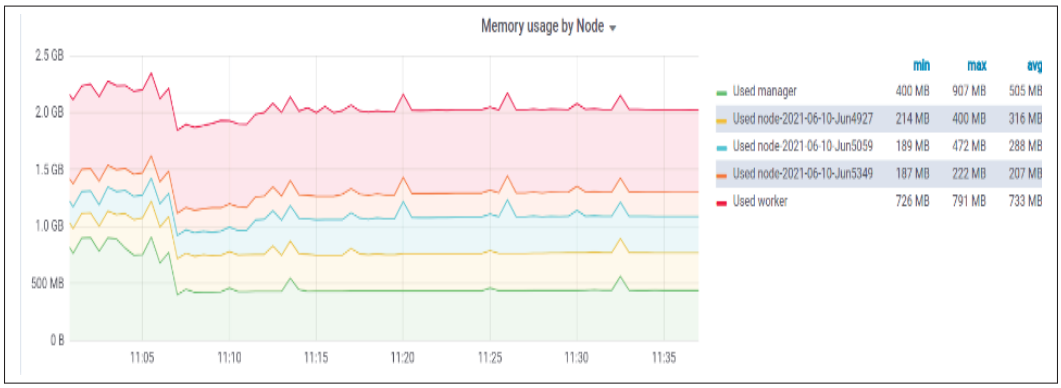

Figure 4.16 Utilisation de la mémoire par nœud d'un cluster exécutant FDQN

D'une part, FDQN a montré des décisions d'adaptation des réplicas plus stables que celles de l'algorithme FQL. D'autre part, FDQN maintient une architecture légère et optimale comparée à FQL en appliquant une auto-adaptation verticale.

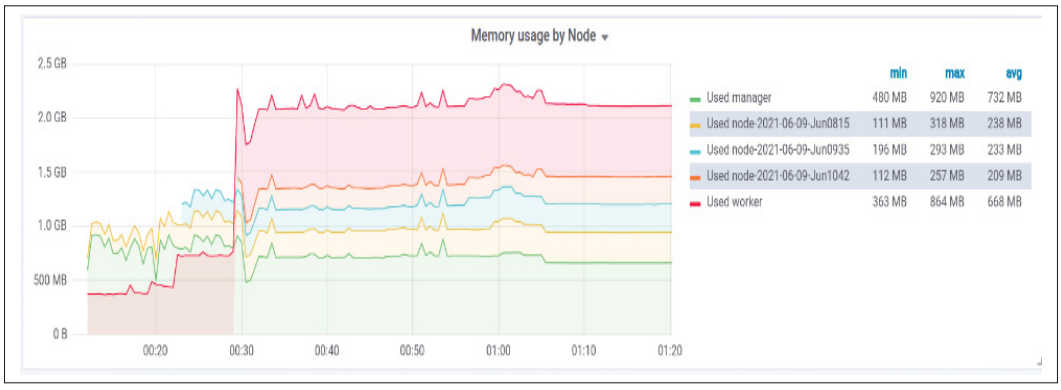

Figure 4.17 Utilisation de la mémoire par les nœuds d'un cluster exécutant FQL

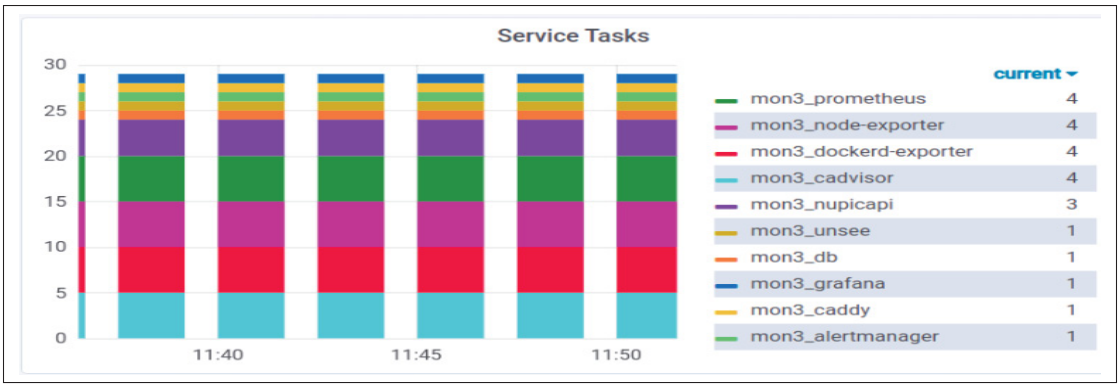

Figure 4.18 les services de tâches d'un cluster exécutant FQL

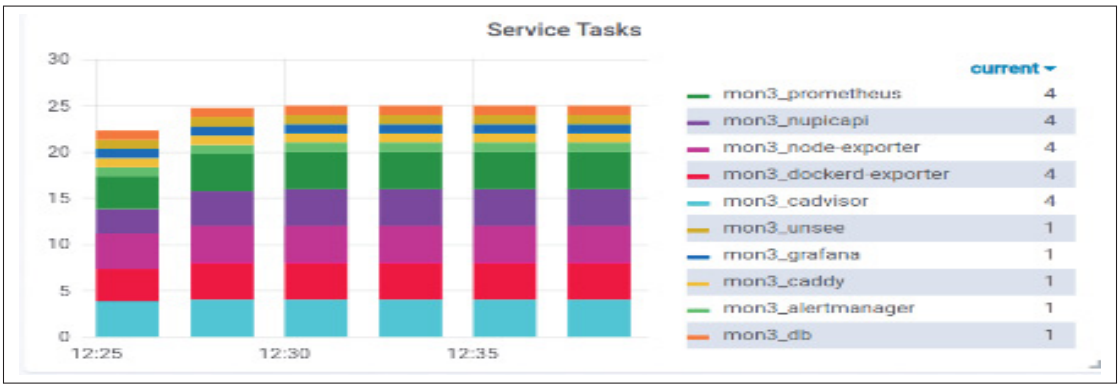

Figure 4.19 les services de tâches d'un cluster exécutant FDQN

4.3 Conclusion

La logique floue est une méthode de lissage qui permet de réduire la fluctuation des données.

Cependant, il semble que cette méthode ne soit pas suffisante pour faire face à des variations contextuelles extrêmement importantes, comme le montrent les résultats de FQL.

D'autre part, en combinant la logique floue avec le réseau Q profond, le FDQN est capable de bien apprendre les détails des données environnementales.

Il en résulte que le FDQN converge beaucoup plus rapidement et progressivement que l'algorithme FQL.

CONCLUSION

Notre étude présente un modèle d'architecture de microservices qui dispose d'une surveillance continue et d'une analyse continue de l'espace d'observation, et fournit à l'architecture une prise de décision dynamique basée sur l'utilisation d'algorithmes d'apprentissage Q flou. Sur la base de la boucle MAPE-K, notre agent d'adaptation observe les espaces d'état de l'architecture et effectue des actions d'adaptation sélectionnées en fonction des résultats de nos deux modèles proposés, notamment FQL et FDQN .

L'évaluation dans notre recherche montre que FDQN est plus approprié pour conduire l'adaptation dans l'environnement opérationnel. Le modèle FDQN a un temps d'apprentissage et de convergence plus rapide que FQL. La récompense totale obtenue par le modèle FDQN est plus élevée dans le nombre réduit d'itérations, ainsi qu'un temps d'adaptation plus rapide. L'utilisation d'algorithmes d'apprentissage flous permet à l'architecture de choisir dynamiquement une stratégie de raisonnement basée sur la récompense la plus élevée obtenue pour chaque paire action-état.

La propriété d'auto-adaptation est obtenue par le paramétrage des services à l'exécution dynamique de swarm. L'intégration de Réseau Q profond flou dans le processus de décision améliore l'efficacité de l'adaptation et réduit les risques. En outre, notre approche surpasse FQL en termes de temps d'apprentissage et de comportement auto-adaptatif pour maintenir l'état du cluster.

La recherche de configurations optimales et de seuils de convergence reste un défi, y compris dans les deux approches proposées.

Ce modèle peut être étendu en proposant de nouvelles actions à l'espace d'action, ce qui permettra à d'autres chercheurs de réaliser différents types d'expériences sur ce modèle, ainsi que de rendre les intervalles automatiquement ajustables.

BIBLIOGRAPHIE

Ahmad, S., Lavin, A., Purdy, S. & Agha, Z. (2017). Unsupervised real-time anomaly detection for streaming data. *Neurocomputing*, 262, 134–147.

Arabnejad, H., Pahl, C., Estrada, G., Samir, A. & Fowley, F. (2017). A fuzzy load balancer for adaptive fault tolerance management in cloud platforms. *European Conference on Service-Oriented and Cloud Computing*, pp. 109–124.

Avizienis, A. (2001). J, C. Laprie, and B. Randell. Fundamental Concepts of Dependability. *Proceedings*, pp. 7–12.

Caicedo, J. C. & Lazebnik, S. (2015). Active object localization with deep reinforcement learning. *Proceedings of the IEEE international conference on computer vision*, pp. 2488–2496.

Chalapathy, R., Menon, A. K. & Chawla, S. (2018). Anomaly detection using one-class neural networks. *arXiv preprint arXiv :1802.06360*.

Chen, X., Lu, C.-D. & Pattabiraman, K. (2014). Failure prediction of jobs in compute clouds : A google cluster case study. *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, pp. 341–346.

Dutreilh, X., Moreau, A., Malenfant, J., Rivierre, N. & Truck, I. (2010). From data center resource allocation to control theory and back. *2010 IEEE 3rd international conference on cloud computing*, pp. 410–417.

Dutreilh, X., Kirgizov, S., Melekhova, O., Malenfant, J., Rivierre, N. & Truck, I. (2011). Using reinforcement learning for autonomic resource allocation in clouds : towards a fully automated workflow. *ICAS 2011, The Seventh International Conference on Autonomic and Autonomous Systems*, pp. 67–74.

Ge, N., Nakajima, S. & Pantel, M. (2015). Online diagnosis of accidental faults for real-time embedded systems using a hidden Markov model. *Simulation*, 91(10), 851–868.

Ghanbari, H., Simmons, B., Litoiu, M. & Iszlai, G. (2011). Exploring alternative approaches to implement an elasticity policy. *2011 IEEE 4th International Conference on Cloud Computing*, pp. 716–723.

Ghobaei-Arani, M., Jabbehdari, S. & Pourmina, M. A. (2018). An autonomic resource provisioning approach for service-based cloud applications : A hybrid approach. *Future Generation Computer Systems*, 78, 191–210.

Gorbil, G. & Gelenbe, E. (2013). Resilient emergency evacuation using opportunistic communications. Dans *Computer and Information Sciences III* (pp. 249–257). Springer.

Han, R., Guo, L., Ghanem, M. M. & Guo, Y. (2012). Lightweight resource scaling for cloud applications. *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pp. 644–651.

Hasan, M. Z., Magana, E., Clemm, A., Tucker, L. & Gudreddi, S. L. D. (2012). Integrated and autonomic cloud resource scaling. *2012 IEEE network operations and management symposium*, pp. 1327–1334.

Hawkins, J. & Blakeslee, S. (2004). On Intelligence–Owl Books. Henry Holt and Company, New York.

Jamshidi, P., Sharifloo, A., Pahl, C., Arabnejad, H., Metzger, A. & Estrada, G. (2016). Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures. *2016 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*, pp. 70–79.

Jayaraman, D. & Grauman, K. (2016). Look-ahead before you leap : end-to-end active recognition by forecasting the effect of motion. *European Conference on Computer Vision*, pp. 489–505.

Jouffe, L. (1998). Fuzzy inference system learning by reinforcement methods. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 28(3), 338–355.

Jung, G., Swint, G., Parekh, J., Pu, C. & Sahai, A. (2006). Detecting bottleneck in n-tier it applications through analysis. *International Workshop on Distributed Systems : Operations and Management*, pp. 149–160.

Khatua, S., Ghosh, A. & Mukherjee, N. (2010). Optimizing the utilization of virtual resources in cloud environment. *2010 IEEE International Conference on Virtual Environments, Human-Computer Interfaces and Measurement Systems*, pp. 82–87.

Kim, D. & Park, S. (2009). Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software. *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pp. 76–85.

Koperek, P. & Funika, W. (2011). Dynamic business metrics-driven resource provisioning in cloud environments. *International Conference on Parallel Processing and Applied Mathematics*, pp. 171–180.

Lama, P. & Zhou, X. (2010). Autonomic provisioning with self-adaptive neural fuzzy control for end-to-end delay guarantee. *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 151–160.

Lample, G. & Chaplot, D. S. (2017). Playing FPS games with deep reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 31(1).

Lee, D., Cha, S. K. & Lee, A. H. (2011). A performance anomaly detection and analysis framework for DBMS development. *IEEE Transactions on Knowledge and Data Engineering*, 24(8), 1345–1360.

Lorido-Botran, T., Miguel-Alonso, J. & Lozano, J. A. (2014). A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 12(4), 559–592.

Lorido-Botrn, T., Miguel-Alonso, J. & Lozano, J. A. (2013). Comparison of auto-scaling techniques for cloud environments.

Magableh, B. & Almiani, M. (2020). A deep recurrent Q network towards self-adapting distributed microservice architecture. *Software : Practice and Experience*, 50(2), 116–135.

Malkowski, S., Hedwig, M. & Pu, C. (2009). Experimental evaluation of N-tier systems : Observation and analysis of multi-bottlenecks. *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 118–127.

Mariani, L., Monni, C., Pezzé, M., Riganelli, O. & Xin, R. (2018). Localizing faults in cloud systems. *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 262–273.

Maurer, M., Breskovic, I., Emeakaroha, V. C. & Brandic, I. (2011). Revealing the MAPE loop for the autonomic management of cloud infrastructures. *2011 IEEE symposium on computers and communications (ISCC)*, pp. 147–152.

Mooers, C. N. (1977). Preventing software piracy. *Computer*, 10(3), 29–30.

Nikravesh, A. Y., Ajila, S. A. & Lung, C.-H. (2014). Cloud resource auto-scaling system based on hidden markov model (hmm). *2014 IEEE International Conference on Semantic Computing*, pp. 124–127.

Ongaro, D. & Ousterhout, J. (2014). In search of an understandable consensus algorithm. *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pp. 305–319.

Pradhan, M., Pradhan, S. K. & Sahu, S. K. (2012). Anomaly detection using artificial neural network. *International Journal of Engineering Sciences & Emerging Technologies*, 2(1), 29–36.

Rao, J., Bu, X., Xu, C.-Z., Wang, L. & Yin, G. (2009). Vconf : a reinforcement learning approach to virtual machines auto-configuration. *Proceedings of the 6th international conference on Autonomic computing*, pp. 137–146.

Salehie, M. & Tahvildari, L. (2005). A policy-based decision making approach for orchestrating autonomic elements. *13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05)*, pp. 173–181.

Salman, E., Stanaćević, M., Das, S. & Djurić, P. M. (2018). Leveraging RF power for intelligent tag networks. *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, pp. 329–334.

Samir, A. & Pahl, C. (2019). Anomaly detection and analysis for clustered cloud computing reliability. *CLOUD COMPUTING*, 2019, 120.

Simmons, B., Ghanbari, H., Litoiu, M. & Iszlai, G. (2011). Managing a SaaS application in the cloud using PaaS policy sets and a strategy-tree. *2011 7th International Conference on Network and Service Management*, pp. 1–5.

Sukhwani, H., Sharma, V. & Sharma, S. (2014). A survey of anomaly detection techniques and hidden markov model. *International Journal of Computer Applications*, 93(18).

Sutton, R. S. (1998). Reinforcement learning : Past, present and future. *Asia-Pacific Conference on Simulated Evolution and Learning*, pp. 195–197.

Tesauro, G. (2007). Reinforcement learning in autonomic computing : A manifesto and case studies. *IEEE Internet Computing*, 11(1), 22–30.

Tesauro, G., Jong, N. K., Das, R. & Bennani, M. N. (2006). A hybrid reinforcement learning approach to autonomic resource allocation. *2006 IEEE International Conference on Autonomic Computing*, pp. 65–73.

Tijsma, Y. (2019). *Tomte : modelling environmental conditions of a data centre by means of a hardware scale model*. (Thèse de doctorat).

Tokic, M. & Palm, G. (2011). Value-difference based exploration : adaptive control between epsilon-greedy and softmax. *Annual conference on artificial intelligence*, pp. 335–346.

Van Hasselt, H. (2012). Reinforcement learning in continuous state and action spaces. Dans *Reinforcement learning* (pp. 207–251). Springer.

Van Hasselt, H., Guez, A. & Silver, D. (2016). Deep reinforcement learning with double q-learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 30(1).

Walsh, W. E., Tesauro, G., Kephart, J. O. & Das, R. (2004). Utility functions in autonomic systems. *International Conference on Autonomic Computing, 2004. Proceedings.*, pp. 70–77.

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M. & Freitas, N. (2016). Dueling network architectures for deep reinforcement learning. *International conference on machine learning*, pp. 1995–2003.

Wu, T., Li, Q., Wang, L., He, L. & Li, Y. (2018). Using reinforcement learning to handle the runtime uncertainties in self-adaptive software. *Federation of International Conferences on Software Technologies : Applications and Foundations*, pp. 387–393.

Xu, C.-Z., Rao, J. & Bu, X. (2012). URL : A unified reinforcement learning approach for autonomic cloud management. *Journal of Parallel and Distributed Computing*, 72(2), 95–105.

Xu, J., Zhao, M., Fortes, J., Carpenter, R. & Yousif, M. (2007). On the use of fuzzy modeling in virtualized data center management. *Fourth International Conference on Autonomic Computing (ICAC'07)*, pp. 25–25.

Yun, S., Choi, J., Yoo, Y., Yun, K. & Young Choi, J. (2017). Action-decision networks for visual tracking with deep reinforcement learning. *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2711–2720.

Zeng, Q., Fu, Y., Jeong, J., Zhen, T., Wang, T., Lei, Y., Mao, H., Jani, A. B., Patel, P., Curran, W. J. et al. (2020). Weekly supervised convolutional long short-term memory neural networks for MR-TRUS registration. *Medical Imaging 2020 : Ultrasonic Imaging and Tomography*, 11319, 1131910.