National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Elastic Resource Management in Microservices Architectures: A Hybrid Approach Combining Reinforcement Learning, Supervised Learning and Critical Path Extraction

Diploma Thesis

of

PANAGIOTIS V. TSIKRITEAS

Supervisor: Nectarios Koziris
Professor NTUA

Athens, November 2025

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Elastic Resource Management in Microservices Architectures: A Hybrid Approach Combining Reinforcement Learning, Supervised Learning and Critical Path Extraction

Diploma Thesis

of

PANAGIOTIS V. TSIKRITEAS

Supervisor: Nectarios Koziris
Professor NTUA

Approved by the examination committee on 7th November 2025.

.....	.....	.....
Nectarios Koziris	Ioannis Konstantinou	Dimitrios Tsoumakos
Professor NTUA	Associate Professor NTUA	Associate Professor NTUA

Athens, November 2025

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism.

.....
Panagiotis V. Tsikriteas
Graduate of School of
Electrical and Computer
Engineering, National
Technical University of
Athens

7 November 2025

Περίληψη

Οι αυξανόμενες υπολογιστικές απαιτήσεις και οι περιορισμοί κλιμάκωσης των μονολιθικών αρχιτεκτονικών έχουν οδηγήσει στην υιοθέτηση αρχιτεκτονικών που βασίζονται σε μικροϋπηρεσίες. Για τη διαχείριση αυτής της πολυπλοκότητας με αυτοματοποιημένο τρόπο, πλατφόρμες όπως το Kubernetes έχουν γίνει τόσο το βιομηχανικό όσο και το ακαδημαϊκό πρότυπο λόγω της ανθεκτικότητας, της επεκτασιμότητας και της ευέλικτης υποστήριξης ενορχήστρωσης αυτών των αρχιτεκτονικών που βασίζονται σε μικροϋπηρεσίες, ειδικά με την εισαγωγή scalers όπως το Horizontal Pod Autoscaler (HPA). Ωστόσο, αυτοί οι scalers βασίζονται σε απλοϊκές ευρετικές μεθόδους που βασίζονται σε κατώφλι, οι οποίες δεν ανταποκρίνονται βέλτιστα στα πολύπλοκα πρότυπα φόρτου εργασίας που αντιμετωπίζουν αυτά τα συστήματα που βασίζονται σε μικροϋπηρεσίες. Ο κύριος στόχος αυτής της διπλωματικής εργασίας είναι να προτείνει ένα pipeline διαχείρισης πόρων που συνδυάζει την εποπτευόμενη μάθηση με το πιθανολογικό calibration για τον εντοπισμό των κρίσιμων στοιχείων που εξάγονται από τα δεδομένα ιχνηλάτησης του συστήματος και την ενισχυτική μάθηση για την αποτελεσματική κατανομή των πόρων του Kubernetes. Ταυτόχρονα, πραγματοποιείται εξαγωγή κρίσιμης διαδρομής με τη χρήση του CRISP καθοδηγώντας ανάλογα τη λήψη αποφάσεων του πράκτορα ενισχυτικής μάθησης. Για την αξιολόγηση των προαναφερθέντων πληροφοριών που εξήχθησαν, η παρούσα εργασία συγκρίνει τρεις πράκτορες ενισχυτικής μάθησης, τον Proximal Policy Optimization (PPO), τον Trust Region Policy Optimization (TRPO) και τον Synchronous Advantage Actor-Critic (A2C), με τον πράκτορα PPO να εκπαιδεύεται με δύο διαφορετικούς αριθμούς επεισοδίων, αξιολογώντας την ικανότητά τους να διαχειρίζονται αποτελεσματικά τους πόρους Kubernetes με μετρικές όπως η από άκρο-σε-άκρο καθυστέρηση (end-to-end latency) σε διάφορα εκατοστημόρια σε συνδυασμό με την σημαντική μετρική του μέσου όρου αναπτυσσόμενων Pods ενώ ταυτόχρονα τονίζει τα μειονεκτήματα που προκύπτουν από τις προαναφερθείσες μετρικές των διαφορετικών αυτών πρακτόρων. Τα αποτελέσματα της αξιολόγησης αναδεικνύουν ότι η μακροχρόνια εκπαίδευση σε τέτοια πολύπλοκα συστήματα είναι απαραίτητη προκειμένου να επιτευχθεί η ικανότητα βέλτιστης κατανομής πόρων ενώ ακόμη και με περιορισμένη εκπαίδευση, οι πράκτορες πέτυχαν εξαιρετική απόδοση σε σύγκριση με αυτή του KHPA, αναδεικνύοντας τη σημασία τέτοιων πρακτόρων στα συστήματα Kubernetes.

Λέξεις Κλειδιά

Kubernetes, Κατανομή Πόρων, Ελαστικότητα, Υπολογιστική νέφους, Μηχανική Μάθηση, Ενισχυτική Μάθηση, DeathStarBench, FIRM, CRISP, Αρχιτεκτονική Μικροϋπηρεσιών

Abstract

The increasing computational demands and scalability limitations of monolithic architectures have driven the adoption of microservices-based architectures where applications are composed of loosely coupled deployable services. To manage that complexity in an automated way, platforms like Kubernetes have become both the industry and academic standard due to their resilience, scalability and versatile support of orchestrating these microservices-based architectures, especially with the introduction of scalers like Horizontal Pod Autoscaler (HPA). However, these scalers rely on simplistic threshold-based heuristics which do not respond well to the complex workload patterns these microservices-based systems encounter. To address this limitation, this thesis's main objective is to propose a resource management pipeline, that combines supervised learning with probabilistic calibration to identify the critical components extracted from the system's trace data and reinforcement learning to allocate Kubernetes resources effectively. This work uses CRISP to extract the constantly changing critical paths of the system, guiding the decision-making of the reinforcement learning agent accordingly. To assess the aforementioned information extracted, this thesis compares three reinforcement learning agents, Proximal Policy Optimization (PPO), Trust Region Policy Optimization (TRPO) and Synchronous Advantage Actor-Critic (A2C), with the PPO agent being trained with two different number of episodes, evaluating their ability to manage Kubernetes resources efficiently with metrics such as the end-to-end latency on different percentiles (50, 95, 99) and the average deployed Pods in the cluster while at the same time highlighting each agent's limitations based on the aforementioned metrics. The evaluation results demonstrate that long-term training in such complex systems is necessary in order to obtain the ability to allocate resources optimally. Notably, even with limited training, the agents achieved strong performance compared to each other and the KHPA baseline, showcasing the importance of such agents in Kubernetes clusters.

Keywords

Kubernetes, Resource Allocation, Elasticity, Cloud Computing, Machine Learning, Reinforcement Learning, DeathStarBench, FIRM, CRISP, Microservices Architectures

στον αδελφό μου και στους γονείς μου

Acknowledgements

This work was done during my undergraduate studies in Electrical and Computer Engineering at National Technical University of Athens (NTUA). First of all, I would like to thank my supervisor Nectarios Koziris and especially my co-supervisor Ioannis Konstantinou for his important supervision and thoughtful advice throughout my thesis and his constant availability at all times. In addition, I would like to thank my brother and my parents for the love and help they provided me throughout these hard working years. Last but not least, I would like to thank all of my friends for their undeniable support they provided me, both emotionally and intellectually during my hard working undergraduate years.

Athens, November 2025

Panagiotis V. Tsikriteas

Contents

List of Figures

List of Tables

Κεφάλαιο 0

Εκτενής Περίληψη

Τα τελευταία χρόνια, το υπολογιστικό νέφος (cloud computing) έχει μεταμορφώσει τον τρόπο με τον οποίο αναπτύσσονται εφαρμογές. Τα σύγχρονα κέντρα δεδομένων μεγάλης κλίμακας φιλοξενούν πλέον ένα ευρύ φάσμα δημοφιλών υπηρεσιών που επηρεάζουν σχεδόν κάθε πτυχή της ανθρώπινης δραστηριότητας. Για την κάλυψη των σύνθετων απαιτήσεων των χρηστών μεταξύ των διαφορετικών υπηρεσιών ενός παρόχου, η συνήθης προσέγγιση στο παρελθόν ήταν η κατασκευή μιας μεγάλης, συχνά περίπλοκης αρχιτεκτονικής, της λεγόμενης μονολιθικής αρχιτεκτονικής (monolithic architecture), στην οποία όλες οι λειτουργίες υλοποιούνται ως ενιαίο σύνολο κώδικα και αναπτύσσονται ως μία ενιαία εκτελέσιμη εφαρμογή.

Αν και αυτή η τεχνική ήταν επαρκής σε απλούστερα περιβάλλοντα, καθώς ο αριθμός των υπηρεσιών αυξάνεται, επιφέρει σημαντική πολυπλοκότητα, καθιστώντας τη διαχείριση δύσκολη ή και ανέφικτη. Τα τελευταία χρόνια, οι προγραμματιστές έχουν στραφεί σε πιο αποκεντρωμένες και επεκτάσιμες προσεγγίσεις. Μεταξύ των αρχιτεκτονικών που προέκυψαν από αυτή τη μετάβαση, οι μικροϋπηρεσίες (microservices) έχουν αποκτήσει ιδιαίτερη δημοφιλία λόγω της σημαντικής κλιμακωσιμότητας και ανθεκτικότητάς τους [14].

0.1 Θεωρητικό Υπόβαθρο

Σε αυτήν την ενότητα περιγράφονται οι θεμελιώδεις τεχνολογίες που χρησιμοποιήθηκαν σε αυτή την διπλωματική εργασία, ξεκινώντας με την ανάλυση του Kubernetes και των συνοδευόμενων τεχνολογιών. Έπειτα, θα αναλυθεί το κομμάτι της Μηχανικής Μάθησης το οποίο επίσης χρησιμοποιήθηκε εκτενώς, με ιδιαίτερη έμφαση να παραχωρείται στην Εποπτευόμενη Μάθηση και την Ενισχυτική Μάθηση.

0.1.1 Kubernetes και συνοδευόμενες τεχνολογίες

Το Kubernetes [1] (K8s) είναι ένα σύστημα ενορχήστρωσης container ανοιχτού κώδικα για την αυτοματοποίηση της ανάπτυξης, της κλιμάκωσης και της διαχείρισης εφαρμογών σε container. Παρακάτω περιγράφονται τα κύρια εξαρτήματα από τα οποία απαρτίζεται η πλατφόρμα αυτή:

- **Nodes**: Ένας κόμβος ενδέχεται να είναι μια εικονική ή φυσική μηχανή, ανάλογα με την υποδομή του νέφους. Κάθε κόμβος διαχειρίζεται από το control plane ή τον κύριο κόμβο και περιέχει τις υπηρεσίες που είναι απαραίτητες για την διαχείριση Pods.

- **Pods**: Τα Pods είναι οι μικρότερες αναπτυσσόμενες μονάδες υπολογιστών που ενδέχεται να δημιουργηθούν και να διαχειριστούν στο Kubernetes. Ένα Pod είναι μια ομάδα ενός ή περισσότερων container, με κοινόχρηστους πόρους αποθήκευσης και δικτύου, και μια προδιαγραφή για τον τρόπο εκτέλεσης των container. Τα περιεχόμενα ενός Pod βρίσκονται πάντα σε κοινή τοποθεσία και προγραμματίζονται ταυτόχρονα και εκτελούνται σε ένα κοινόχρηστο περιβάλλον. Ένα Pod μοντελοποιεί έναν ¨λογικό κεντρικό υπολογιστή¨ που αφορά συγκεκριμένες εφαρμογές, περιέχοντας ένα ή περισσότερα container εφαρμογών. Τα συγκεκριμένα container είναι στενά συνδεδεμένα με κοινόχρηστους πόρους αποθήκευσης και δικτύου, και μια προδιαγραφή για τον τρόπο εκτέλεσης τους[1].

- **Services**: Σε ένα cluster, τα Pods είναι ασταθή, καθώς ενδέχεται να τερματιστούν ή να επανεκκινηθούν απροσδόκητα. Για αυτόν τον λόγο, τα Services λειτουργούν ως ένα αφηρημένο επίπεδο που επιτρέπει την αξιόπιστη πρόσβαση στα Pods μέσω σταθερών endpoints, ανεξάρτητα από τις δυναμικές IP διευθύνσεις τους. Κάθε Service ορίζει ένα σύνολο Pods και έναν τρόπο πρόσβασης σε αυτά.

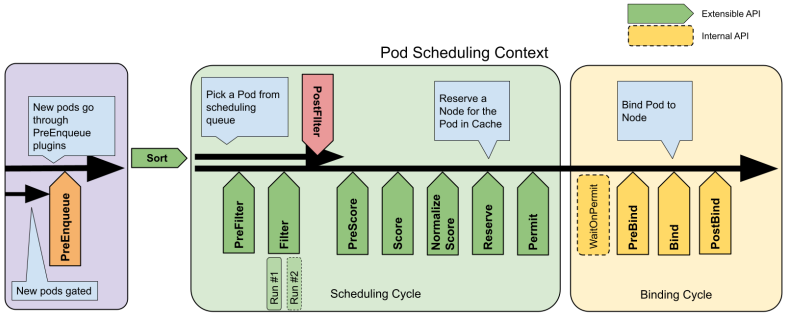
- **Deployment**: Το Deployment είναι απαραίτητο καθώς χειρίζεται το lifecycle των Pods διαχειρίζοντας λειτουργίες όπως η κλιμάκωση και διάφορες άλλες ιδιαίτερα χρήσιμες λειτουργίες.

Ένα Kubernetes σύστημα ή συστάδα αποτελείται από τους κόμβους εργασίας (worker nodes) οι οποίοι διαχειρίζονται τα Pods, τα οποία αποτελούν τα στοιχεία που λαμβάνουν το φορτίο εργασίας. Παράλληλα, το Control Plane ή αλλιώς κόμβος αφέντης (master-node) είναι υπεύθυνο για τη σωστή λειτουργία των worker nodes καθώς και για τον προγραμματισμό των Pods σε αυτούς. Στα παραγωγικά περιβάλλοντα, μια καλή πρακτική είναι το Control Plane να διαθέτει πολλαπλά αντίγραφα σε διαφορετικούς κόμβους εντός της συστάδας, παρέχοντας έτσι υψηλή διαθεσιμότητα και ανθεκτικότητα σε σφάλματα στο νέφος. Τα κύρια εξαρτήματα του κόμβου-αφέντη παρουσιάζονται παρακάτω:

- **kube-apiserver**: Ο βασικός διακομιστής που εκθέτει το Kubernetes HTTP API.

- **etcd**: Συνεπής και υψηλής διαθεσιμότητας βάση δεδομένων τύπου key-value που αποθηκεύει όλα τα δεδομένα του API server.

- **kube-scheduler**: Εντοπίζει Pods που δεν έχουν ακόμα δεσμευτεί σε κάποιον κόμβο και εκχωρεί το κάθε Pod σε κατάλληλο node. Χρησιμοποιώντας τον αλγόριθμο Round-Robin, ο scheduler επιλέγει έναν node και αξιολογεί αν είναι κατάλληλος για την εκτέλεση του Pod.

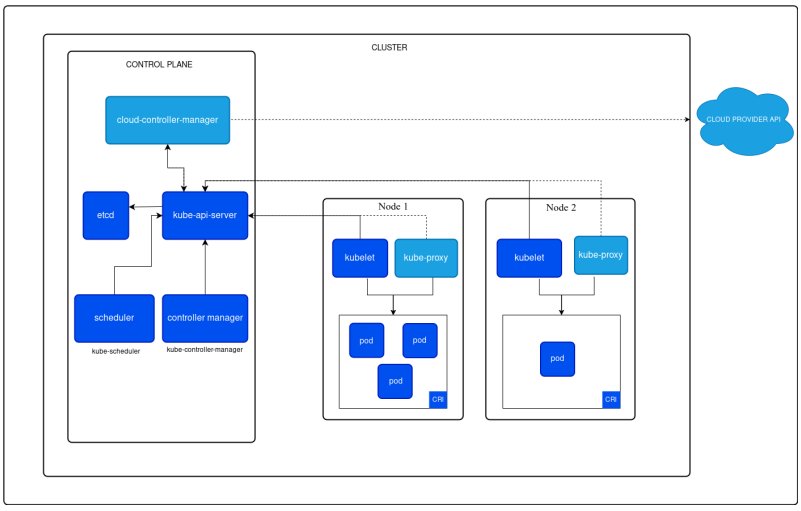


Σχήμα 0.1. *Πλαίσιο προγραμματισμού του Kubernetes, Πηγή: [1].*

- **kube-controller-manager**: Εκτελεί controllers για την εφαρμογή της συμπεριφοράς των APIs του Kubernetes.
- **cloud-controller-manager (προαιρετικό)**: Ενσωματώνεται με την υποδομή του παρόχου cloud. Παραδείγματα παρόχων είναι οι AWS [15], Microsoft Azure [16] και Google Cloud [17].

Συνεχίζοντας με τα στοιχεία του κόμβου (Node components) που εκτελούνται σε κάθε worker node και είναι υπεύθυνα για τη διατήρηση των ενεργών Pods και την παροχή του περιβάλλοντος εκτέλεσης Kubernetes:

- **kubelet**: Βεβαιώνεται ότι τα Pods εκτελούνται σωστά και χωρίς σφάλματα, συμπεριλαμβανομένων των containers τους.
- **kube-proxy (προαιρετικό)**: Διαχειρίζεται τους κανόνες δικτύου στους κόμβους ώστε να υλοποιούνται τα Services.
- **Container runtime**: Λογισμικό υπεύθυνο για την εκτέλεση των containers. Το πιο συνηθισμένο είναι το Docker (πλέον Docker Engine).

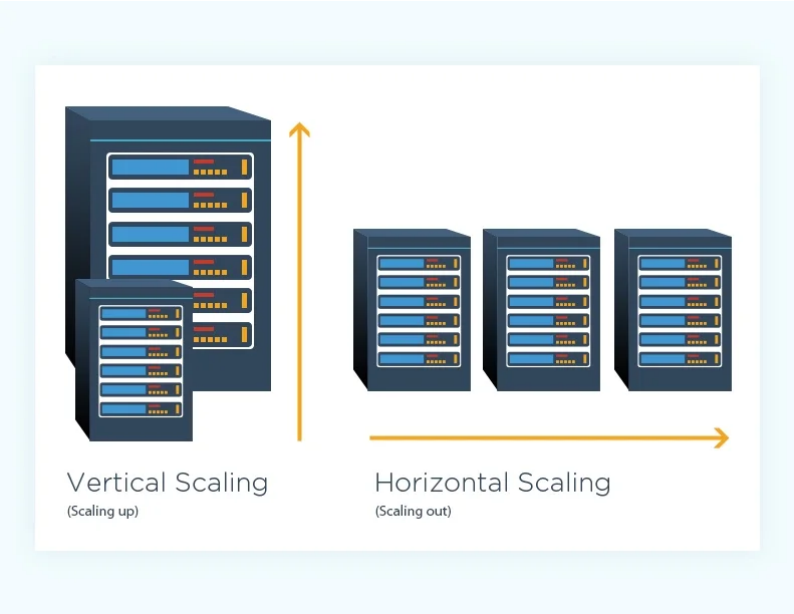


Σχήμα 0.2. *Αρχιτεκτονική του Kubernetes cluster, Πηγή: [1].*

Έπειτα, σε συνδυασμό με όλες τις παραπάνω τεχνολογίες, το Kubernetes δίνει τη δυνατότητα κλιμάκωσης, επιτρέποντας την αυτόματη προσαρμογή των πόρων ανάλογα με το φορτίο του συστήματος. Οι κύριες κατηγορίες κλιμάκωσης που εφαρμόζονται είναι οι παρακάτω:

- **Οριζόντια κλιμάκωση**: Μία από τις πιο σημαντικές και πιο μελετημένη προσέγγιση αυτόματης κλιμάκωσης πόρων στο Kubernetes είναι η οριζόντια κλιμάκωση, η οποία περιλαμβάνει την αύξηση ή τη μείωση του αριθμού των αντιγράφων Pod για να ταιριάζει με τη φόρτου εργασίας και ταυτόχρονα να κατανέμει το φόρτο εργασίας στα τρέχοντα αντίγραφα Pod. Με αυτόν τον τρόπο, οι εφαρμογές ενδέχεται να διαχειριστούν το αυξημένο φορτίο εργασίας ή να μειώσουν τη χρήση πόρων σε περιόδους χαμηλής ζήτησης.

- **Κάθετη κλιμάκωση**: Αντιθέτως, στην Κάθετη κλιμάκωση το Kubernetes αντί να αυξάνει ή να μειώνει τον αριθμό των αντιγράφων Pod της συγκεκριμένης υπηρεσίας, το Kubernetes είναι σε θέση να δώσει περισσότερους πόρους στο Pod, με τους πιο συνηθισμένους να είναι η CPU και η μνήμη, προκειμένου να καλύψει τις τρέχουσες απαιτήσεις του φόρτου εργασίας.



Σχήμα 0.3. *Κάθετη και οριζόντια κλιμάκωση, Πηγή: [2].*

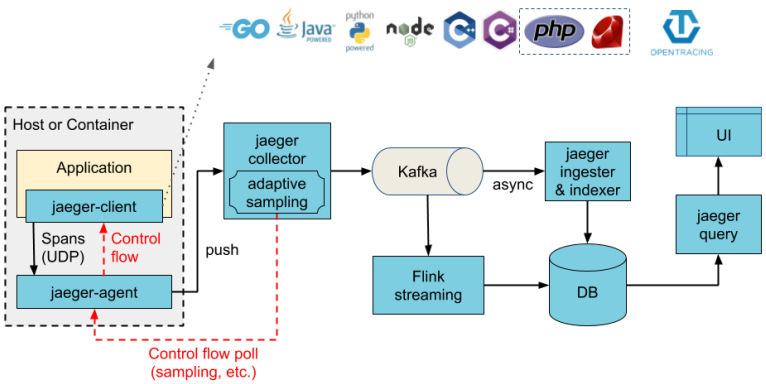
Έχοντας και οι δύο τεχνικές πάρα πολλά πλεονεκτήματα, για λόγους απλότητας για αρχή σε αυτή την διπλωματική εργασία, θα μελετηθεί ο μηχανισμός της οριζόντας κλιμάκωσης του Kubernetes τέθηκε ως σημείο αναφοράς στις μελέτες. Ο μηχανισμός αυτός είναι ο **Horizontal Pod Autoscaler (HPA)** και χρησιμοποιεί τον παρακάτω αλγόριθμο για να προσδιορίσει τον κατάλληλο αριθμό Pods για ένα συγκεκριμένο Service:

$$\text{desiredReplicas} = \text{ceil}\left[\text{currentReplicas} \times \frac{\text{currentMetric}}{\text{targetMetric}}\right]$$

Με βάση τον παραπάνω αλγόριθμο και μια διαμορφώσιμη ανοχή (συνήθως 10%) που εφαρμόζεται στην βασική αναλογία κλιμάκωσης, το HPA λαμβάνει την απόφαση να κλιμακώσει τα Pods. Είναι πολύ σημαντικό να σημειωθεί ότι αυτός ο αλγόριθμος λαμβάνει υπόψη όλα τα Pod σε κατάσταση Ready και αυτά με καθορισμένη χρονική σήμανση διαγραφής (αντικείμενα με χρονική σήμανση διαγραφής βρίσκονται σε διαδικασία τερματισμού/αφαίρεσης) αγνοούνται και όλα τα Pod που έχουν αποτύχει απορρίπτονται.

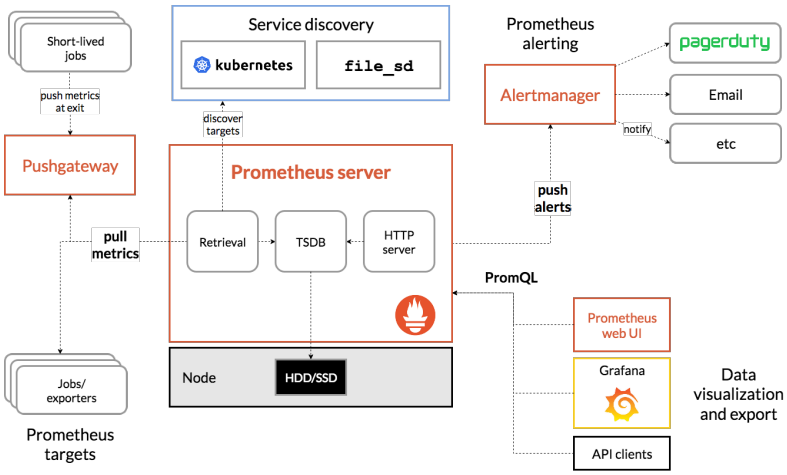
Για την συνεχή παρακολούθηση και αποθήκευση των δεδομένων από τη περίπλοκη αρχιτεκτονική των μικροϋπηρεσιών χρησιμοποιήθηκαν τα δύο παρακάτω εργαλεία, τα οποία αναλύονται συντοπτικά:

- **Jaeger**: Το Jaeger [3] είναι ένα σύστημα κατανεμημένου εντοπισμού ανοιχτού κώδικα, από άκρο σε άκρο, που αναπτύχθηκε αρχικά από την Uber για την παρακολούθηση και την αντιμετώπιση σύνθετων αρχιτεκτονικών μικρουπηρεσιών. Βοηθά τους προγραμματιστές να κατανοήσουν πώς τα αιτήματα ρέουν μέσω ενός συστήματος συλλέγοντας και οπτικοποιώντας δεδομένα εντοπισμού, τα οποία περιλαμβάνουν πληροφορίες χρονισμού για λειτουργίες σε πολλαπλές υπηρεσίες. Η αρχιτεκτονική του φαίνεται παρακάτω:



Σχήμα 0.4. *Αρχιτεκτονική του Jaeger, Πηγή: [3].*

- **Prometheus**: Το Prometheus είναι ένα σύστημα παρακολούθησης και ειδοποίησης συστημάτων ανοιχτού κώδικα που δημιουργήθηκε αρχικά στο SoundCloud [18], αλλά από το 2016 έχει ενταχθεί στο CNCF [19] ως το δεύτερο φιλοξενούμενο έργο, μετά το Kubernetes [1]. Το Prometheus συλλέγει και αποθηκεύει τις μετρήσεις του ως δεδομένα χρονοσειρών, δηλαδή οι πληροφορίες μετρήσεων αποθηκεύονται με τη χρονική σήμανση στην οποία καταγράφηκαν, μαζί με προαιρετικά ζεύγη κλειδιού-τιμής που ονομάζονται ετικέτες [4]. Η αρχιτεκτονική του φαίνεται παρακάτω:



Σχήμα 0.5. *Αρχιτεκτονική του Prometheus, Πηγή: [4].*

0.1.2 Μηχανική μάθηση, Εποπτευόμενη Μάθηση και Νευρωνικά δίκτυα

Η Μηχανική μάθηση [20] είναι ένας τομέας της τεχνητής νοημοσύνης που επιτρέπει στα συστήματα να μαθαίνουν αυτόματα μοτίβα και να λαμβάνουν αποφάσεις ή προβλέψεις με βάση δεδομένα, χωρίς να προγραμματίζονται ρητά για συγκεκριμένες εργασίες. Βασίζεται σε στατιστικές και υπολογιστικές τεχνικές για την εξαγωγή πληροφοριών από μικρά ή μεγάλα σύνολα δεδομένων και ταυτόχρονα τον μετριασμό της πολυπλοκότητας της απαιτούμενης εργασίας. Προκειμένου αυτά τα συστήματα να μάθουν μοτίβα, εφαρμόζονται μαθηματικοί αλγόριθμοι σε πολλές επαναλήψεις, που ονομάζονται εποχές, για την επαναληπτική προσαρμογή των παραμέτρων του μοντέλου. Μια ειδική κατηγορία της μηχανικής μάθησης είναι η επιβλεπόμενη μάθηση, στην οποία το σύνολο δεδομένων στο οποίο μαθαίνει ο επαναληπτικός αλγόριθμος εμπεριέχει ετικέτες επιτρέποντάς τους να προβλέπουν ή να ταξινομούν αποτελέσματα προσδιορίζοντας μοτίβα και σχέσεις μεταξύ χαρακτηριστικών εισόδου και των ετικετών. Έτσι το μοντέλο προσαρμόζει επαναληπτικά τις παραμέτρους του για να ελαχιστοποιήσει τις τιμές μεταξύ προβλεπόμενων και πραγματικών τιμών, όπως καλείται από μια συνάρτηση απώλειας. Στη συγκεκριμένη διπλωματική εργασία ο κύριος αλγόριθμος που χρησιμοποιήθηκε είναι ο **Support Vector Machine (SVM)** [21] ο οποίος είναι αλγόριθμος εποπτευόμενης μάθησης που χρησιμοποιούνται κυρίως για ταξινόμηση. Η βασική ιδέα είναι να βρεθεί ένα υπερεπίπεδο σε έναν χώρο χαρακτηριστικών υψηλής διάστασης που διαχωρίζει σημεία δεδομένων διαφορετικών κλάσεων με το μέγιστο δυνατό περιθώριο. Τα σημεία δεδομένων που βρίσκονται πιο κοντά σε αυτό το υπερεπίπεδο ονομάζονται διανύσματα υποστήριξης, καθώς ορίζουν το όριο και έτσι το γραμμικά διαχωρίσιμο πρόβλημα περιγράφεται ως:

$$\min_{w,\,b} \quad \frac{1}{2}\|w\|^2 \tag{1}$$

$$y_i(w^\top x_i + b) \geq 1, \tag{2}$$

$$i = 1, \ldots, N \tag{3}$$

όπου N είναι το συνολικό πλήθος των σημείων δεδομένων, με $x_i \in \mathbb{R}^d$ και $y_i \in \{-1, 1\}$ για $i = 1, \ldots, N$.

Στην περίπτωση που τα δεδομένα δεν είναι γραμμικά διαχωρίσιμα, εισάγονται δύο τροποποιήσεις στη βασική διατύπωση του SVM:

- **Μεταβλητές ανοχής (slack variables)** $\xi_i \geq 0$ ώστε να επιτρέπονται σφάλματα ταξινόμησης (soft margin).
- **Απεικόνιση με πυρήνα (kernel mapping)** $\phi(x)$ για τον μετασχηματισμό των δεδομένων σε χώρο χαρακτηριστικών μεγαλύτερης διάστασης, όπου είναι δυνατός ο γραμμικός διαχωρισμός.

Ορίζεται μια απεικόνιση για την επίλυση του παρακάτω προβλήματος βελτιστοποίησης:

$$\phi : \mathbb{R}^d \to \mathcal{H} \tag{4}$$

$$\min_{w,b,\xi} \quad \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N} \xi_i \tag{5}$$

$$y_i(w^\top \phi(x_i) + b) \geq 1 - \xi_i, \quad i = 1, \ldots, N \tag{6}$$

$$\xi_i \geq 0, \quad i = 1, \ldots, N \tag{7}$$

όπου $C > 0$ είναι παράμετρος που ρυθμίζει την ισορροπία μεταξύ του πλάτους του περιθωρίου και των σφαλμάτων ταξινόμησης.

Αντί να υπολογιστεί ρητά η $\phi(x)$, εφαρμόζεται το **κόλπο του πυρήνα (kernel trick)**, με το οποίο το εσωτερικό γινόμενο $\phi(x_i)^\top \phi(x_j)$ αντικαθίσταται από μια συνάρτηση πυρήνα:

$$K(x_i, x_j) = \phi(x_i)^\top \phi(x_j) \tag{8}$$

Συνηθισμένες επιλογές για συναρτήσεις πυρήνα είναι ο πολυωνυμικός πυρήνας, ο πυρήνας ακτινικής βάσης (RBF), και ο sigmoid πυρήνας, με τον κάθε πυρήνα να έχει διαφορετικά πλεονεκτήματα και μειονεκτήματα. Ενώ οι παραδοσιακοί αλγόριθμοι μηχανικής μάθησης που παρουσιάζονται παραπάνω έχουν τη δυνατότητα να είναι πολύ αποτελεσματικοί και συχνά οδηγούν σε πολύ ελπιδοφόρα αποτελέσματα, όταν το σύνολο δεδομένων ή ο στόχος-στόχος απαιτεί σύνθετες λύσεις, απαιτούνται πιο σύνθετες αρχιτεκτονικές για την επίτευξη αξιόπιστης και ακριβούς απόδοσης. Εμπνευσμένα από τον ανθρώπινο εγκέφαλο αναπτύχθηκαν τα **νευρωνικά δίκτυα**, κατασκευασμένα από μεμονωμένους νευρώνες και επίπεδα (συχνά ονομαζόμενα κρυφά επίπεδα). Κάθε νευρώνας λαμβάνει μία ή περισσότερες τιμές εισόδου, εφαρμόζει ένα σταθμισμένο άθροισμα ακολουθούμενο από μια μη γραμμική συνάρτηση ενεργοποίησης και μεταβιβάζει το αποτέλεσμα στο επόμενο επίπεδο [22].

$$a_i^{(l)} = g\left(\sum_{j=1}^{n} w_{ij}^{(l)} a_j^{(l-1)} + b_i^{(l)}\right) \tag{9}$$

όπου:

- $a_i^{(l)}$: η ενεργοποίηση του νευρώνα i στο επίπεδο l,
- $w_{ij}^{(l)}$: το βάρος που συνδέει τον νευρώνα j στο προηγούμενο επίπεδο $(l-1)$ με τον νευρώνα i,
- $b_i^{(l)}$: ο όρος μετατόπισης (bias) για τον νευρώνα i,
- $g(\cdot)$: μια μη γραμμική συνάρτηση ενεργοποίησης (π.χ. ReLU, sigmoid ή tanh),
- n: ο αριθμός των νευρώνων στο προηγούμενο επίπεδο.

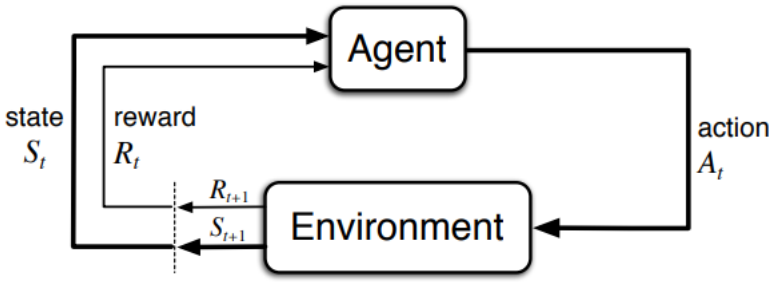
0.1.3 Ενισχυτική μάθηση και Βαθιά Ενισχυτική μάθηση

Στην προηγούμενη ενότητα, παρουσιάστηκαν οι προσεγγίσεις της εποπτευόμενης μάθησης, οι οποίες βασίζονται σε μεγάλο βαθμό σε στατικά, προ-επισημασμένα σύνολα δεδομένων και ουσιαστικά σε έναν επόπτη για να σχολιάζει είτε χειροκίνητα είτε με κάποιο προγραμματιζόμενο τρόπο κάθε καταχώρηση στη μεταβλητή-στόχο αυτού του συνόλου δεδομένων. Αυτό ενδέχεται να είναι πολύ δαπανηρό και πολλές φορές αδύνατο να γίνει σε περιβάλλοντα επιπέδου παραγωγής όπου η τιμή της μεταβλητής-στόχου είναι άγνωστη μέχρι να συμβεί το γεγονός ή αλλάζει συνεχώς με την πάροδο του χρόνου, γεγονός που σημαίνει ότι πρόκειται για ένα εξαιρετικά δυναμικό περιβάλλον. Για να ξεπεράσει αυτές τις προκλήσεις, η Ενισχυτική Μάθηση (RL) εισάγει μια διαφορετική προσέγγιση στα προβλήματα διαδοχικής λήψης αποφάσεων με τη χρήση των Διαδικασιών Απόφασης Markov (Markov Decision Processes (MDP)) οι οποίες περιγράφονται παρακάτω:

$$\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, P, R, \gamma \rangle$$

όπου:

- Ένα σύνολο καταστάσεων $s \in S$, το οποίο ενδέχεται να είναι διακριτό ή συνεχές.
- Ένα σύνολο ενεργειών $a \in A$ που έχει τη δυνατότητα να εκτελέσει ο πράκτορας, διακριτό ή συνεχές.
- Μια συνάρτηση ανταμοιβής
$$R : S \times A \to \mathbb{R}$$
η οποία αντιστοιχίζει έναν πραγματικό αριθμό ως ανταμοιβή σε κάθε ζεύγος κατάστασης–ενέργειας.
- Μια συνάρτηση μεταβάσεων
$$P : S \times A \times S \to [0, 1]$$
όπου $P(s'|s, a)$ εκφράζει την πιθανότητα (στη διακριτή περίπτωση) ή την πυκνότητα πιθανότητας (στη συνεχή περίπτωση) μετάβασης στην επόμενη κατάσταση s', δεδομένης της τρέχουσας κατάστασης s και της ενέργειας a.
- Ένας συντελεστής προεξόφλησης $\gamma \in [0, 1]$, ο οποίος καθορίζει τη σχετική σημασία των μελλοντικών ανταμοιβών σε σχέση με τις άμεσες.



Σχήμα 0.6. *Αλληλεπίδραση MDP, Πηγή: [5].*

Οι αλγόριθμοι ενισχυτικής μάθησης βασίζονται στην έννοια της συνάρτησης αξίας, η οποία εκφράζει πόσο ικανοποιητική είναι μια κατάσταση ή μια ενέργεια ως προς τη μακροπρόθεσμη αθροιστική ανταμοιβή η οποία ορίζεται ως:

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \\ &= R_{t+1} + \gamma \left(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots \right) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{10}$$

Τυπικά, μια πολιτική ορίζεται ως μια απεικόνιση από καταστάσεις σε πιθανότητες επιλογής κάθε δυνατής ενέργειας. Αν ο πράκτορας ακολουθεί την πολιτική π τη χρονική στιγμή t, τότε η $\pi(a|s)$ δηλώνει την υπό συνθήκη πιθανότητα ότι $A_t = a$, δεδομένου ότι $S_t = s$.

Ο τελικός στόχος είναι η μεγιστοποίηση της αναμενόμενης αθροιστικής προεξοφλημένης ανταμοιβής, μέσω της εύρεσης μιας βέλτιστης πολιτικής π^*. Η **συνάρτηση αξίας κατάστασης** (state-value function) υπό μια πολιτική π μετασχηματίζεται ως Bellman εξισώση και ορίζεται ως:

$$\begin{aligned} V^\pi(s) &= \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} P(s'|s,a) \left[R(s,a) + \gamma V^\pi(s') \right] \\ &= \mathbb{E}_{a \sim \pi(\cdot|s)} \left[\sum_{s' \in \mathcal{S}} P(s'|s,a) \left[R(s,a) + \gamma V^\pi(s') \right] \right] \\ &= \mathbb{E}_{a \sim \pi(\cdot|s)} \left[R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^\pi(s') \right] \end{aligned} \tag{11}$$

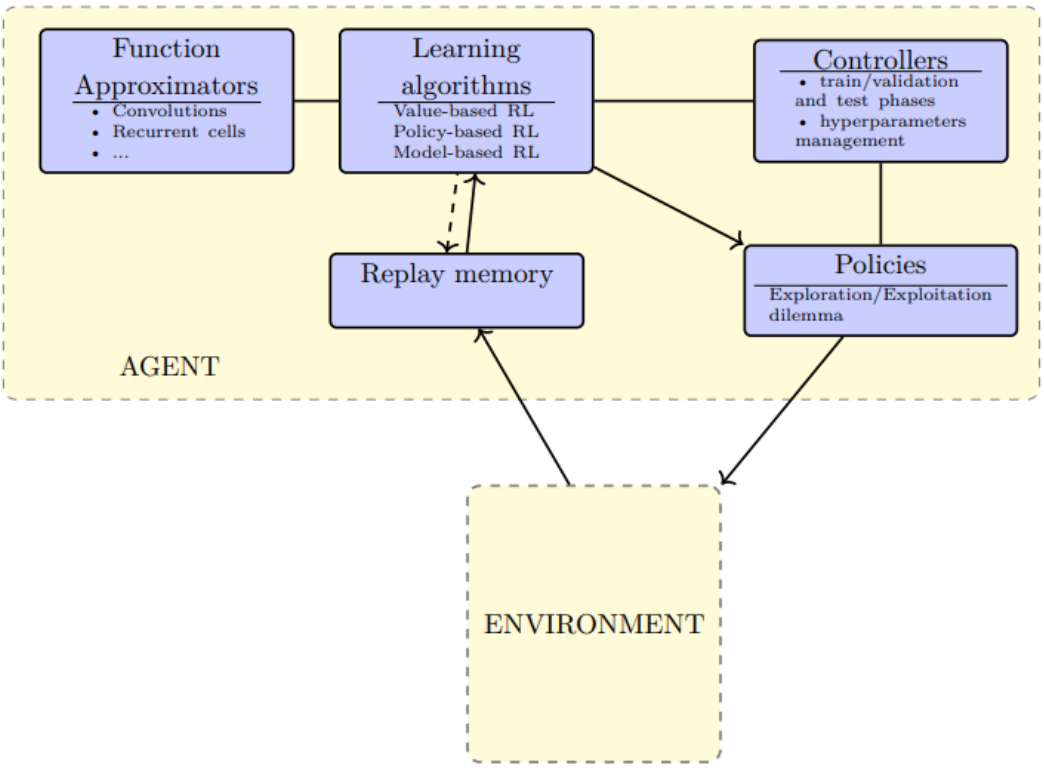
και μετρά την αναμενόμενη αθροιστική ανταμοιβή ξεκινώντας από την κατάσταση s, ακολουθώντας την πολιτική π.

Αντίστοιχα, η **συνάρτηση αξίας κατάστασης–ενέργειας** (action-value function) ορίζεται ως:

$$Q^\pi(s,a) = R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a) \sum_{a' \in \mathcal{A}} \pi(a' \mid s') Q^\pi(s',a') \tag{12}$$

και μετρά την αναμενόμενη ανταμοιβή ξεκινώντας από την κατάσταση s, εκτελώντας την

ενέργεια a, και έπειτα ακολουθώντας την πολιτική π. Με βάση τα παραπάνω οι αλγόριθμοι της Ενισχυτικής Μάθησης χωρίζονται σε τρεις κατηγορίες όπου η πρώτη κατηγορία βασίζεται στη μεγιστοποίηση κάποιας από τις συναρτήσεις αξίας, έπειτα η δεύτερη βασίζεται στην κατευθείαν εκτίμηση της πολιτικής και τέλος, η τρίτη βασίζεται στον συνδυασμό αυτών των δύο προηγούμενων τεχνικών. Παράλληλα, για την εκτίμηση των περίπλοκων αυτών συναρτήσεων υπό περίπλοκα περιβάλλοντα, χρησιμοποιούνται τα νευρωνικά δίκτυα τα οποία, όπως προαναφέρθηκαν έχουν τη δυνατότητα εκτίμησης συναρτήσεων υπό περίπλοκες συνθήκες. Οι αλγόριθμοι που βασίζονται σε αυτή τη τεχνική εκτίμησης ονομάζονται αλγόριθμοι Βαθιάς Ενισχυτικής Μάθησης και παρατίθενται παρακάτω:



Σχήμα 0.7. *Γενικό σχήμα Βαθιάς Ενισχυτικής Μάθησης, Πηγή: [6].*

Πιο συγκεκριμένα, οι αλγόριθμοι οι οποίοι χρησιμοποιήθηκαν για τη μελέτη της κατανομής πόρων σε συστήματα Kubernetes είναι οι παρακάτω:

Αρχικά **Trust Region Policy Optimization (TRPO)** [23] είναι ένας από τους πρώτους καινοτόμους αλγορίθμους που προτάθηκαν, βελτιώνοντας τις κλασικές μεθόδους policy gradient μέσω της εισαγωγής μιας έξυπνης συνθήκης που εξασφαλίζει σταθερή και μονοτονικά βελτιούμενη εκπαίδευση της πολιτικής. Οι παραδοσιακές μέθοδοι policy gradient ενδέχεται να πραγματοποιούν υπερβολικά μεγάλες ενημερώσεις, οδηγώντας σε απότομες αλλαγές στην πολιτική και συνεπώς προκύπτει ασταθής μάθηση, καθιστώντας τη σύγκλιση δύσκολη. Για την αντιμετώπιση αυτών των προβλημάτων, το TRPO προτείνει τη μεγιστοποίηση ενός εναλλακτικού στόχου (surrogate objective), ο οποίος είναι υπολογιστικά πιο εύκολος (βασισμένος στη μέθοδο importance sampling, που είναι τεχνική Monte Carlo), ενώ χρησιμοποιεί τη KL-divergence [24] για να εξασφαλίσει ότι η νέα πολιτική δεν αποκλίνει σημαντικά από την προηγούμενη, παραμένοντας εντός μιας trust region. Με αυτόν τον τρόπο εξασφαλίζεται

σταθερή και αξιόπιστη βελτίωση από την πολιτική $\pi_{\theta_{\text{old}}}$ στην πολιτική π_θ.

$$\max_{\theta} \quad \mathbb{E}_{s,a\sim\pi_{\theta_{\text{old}}}}\left[\frac{\pi_\theta(a \mid s)}{\pi_{\theta_{\text{old}}}(a \mid s)} A^{\pi_{\theta_{\text{old}}}}(s,a)\right] \tag{13}$$

$$\mathbb{E}_{s\sim\pi_{\theta_{\text{old}}}}\left[D_{\text{KL}}(\pi_{\theta_{\text{old}}}(\cdot \mid s) \,\|\, \pi_\theta(\cdot \mid s))\right] \le \delta \tag{14}$$

όπου το δ είναι το όριο της trust region, το οποίο ελέγχει τη μέγιστη επιτρεπόμενη KL-divergence μεταξύ της παλιάς και της νέας πολιτικής, και το $D_{\text{KL}}(\pi_{\theta_{\text{old}}} \,\|\, \pi_\theta)$ είναι η απόκλιση Kullback–Leibler που μετρά πόσο αποκλίνει η νέα πολιτική από την προηγούμενη. Το A^π είναι η advantage function (βλ. 3.24). Αν και το TRPO προσφέρει σταθερές ενημερώσεις πολιτικής μέσω του constraint στην KL-divergence, είναι υπολογιστικά ακριβό, καθώς απαιτεί conjugate gradient optimization και line search για την ικανοποίηση του περιορισμού.

Αντί για αυτό, προτάθηκε ένας πιο απλός αλγόριθμος, το **Proximal Policy Optimization (PPO)** [25], ο οποίος αντικαθιστά τον σκληρό περιορισμό στην KL-divergence με μια προσεγγιστική συνάρτηση που βασίζεται σε clipping:

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \tag{15}$$

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t\left[\min\left(r_t(\theta)\,A_t,\ \text{clip}(r_t(\theta),\,1-\epsilon,\,1+\epsilon)\,A_t\right)\right] \tag{16}$$

όπου ϵ είναι μια υπερπαράμετρος (συνήθως 0.2). Η αντικειμενική συνάρτηση αποτελείται από δύο όρους: ο πρώτος είναι το αρχικό surrogate objective, ενώ ο δεύτερος το τροποποιεί, περιορίζοντας το λόγο πιθανοτήτων r_t ώστε να παραμένει εντός του διαστήματος $[1-\epsilon, 1+\epsilon]$. Με αυτόν τον τρόπο αποφεύγονται μεγάλες αποκλίσεις στην ενημέρωση της πολιτικής.

Τέλος, μια ακόμα πιο απλοποιημένη προσέγγιση είναι ο **synchronous Advantage Actor-Critic (A2C)** [26], στον οποίο, όπως περιγράφηκε προηγουμένως, χρησιμοποιείται ένα δίκτυο για τον actor και ένα για τον critic (ή ένα κοινό δίκτυο, όπως στις προεπιλεγμένες υλοποιήσεις βιβλιοθηκών), και εισάγεται όρος εντροπίας στη συνάρτηση policy optimization:

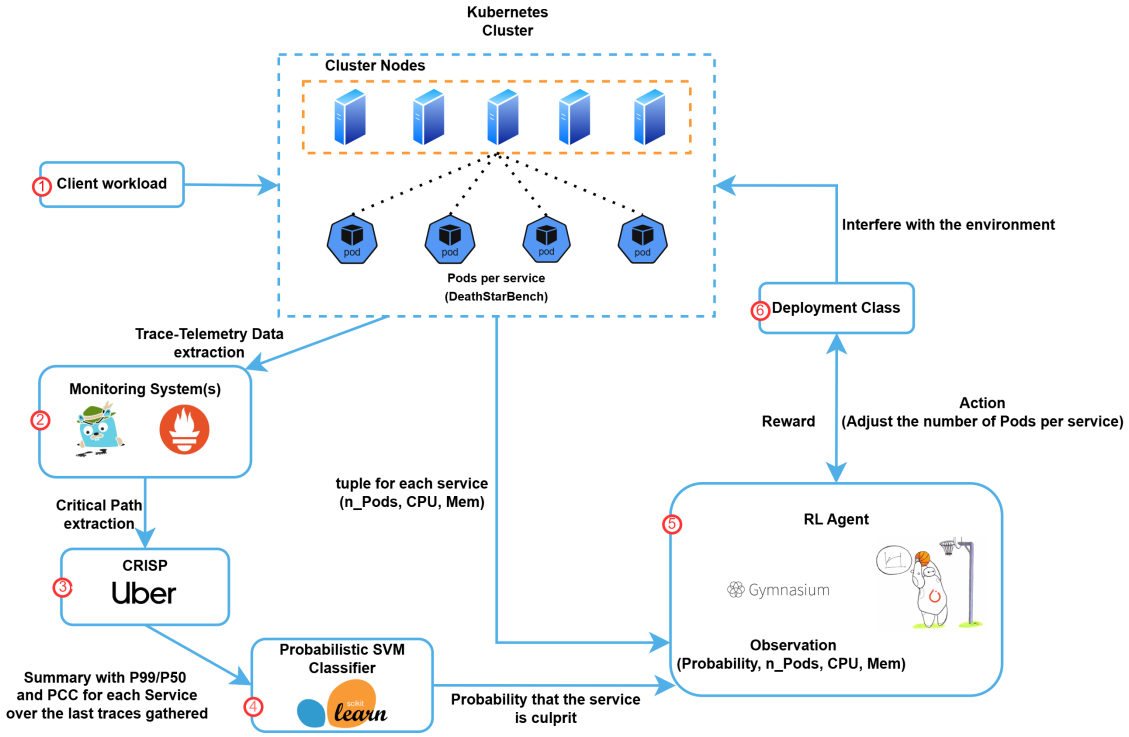
$$\nabla_\theta J_{\text{actor}}(\theta) = \mathbb{E}_t\left[\nabla_\theta \log \pi_\theta(a_t \mid s_t)\,A_t + \beta\,\nabla_\theta H(\pi(s_t;\theta))\right] \tag{17}$$

$$L_{\text{critic}}(\theta) = \mathbb{E}_t\left[\left(G_t - V_\theta(s_t)\right)^2\right] \tag{18}$$

όπου H είναι η εντροπία και η υπερπαράμετρος β ελέγχει τη βαρύτητα του όρου entropy regularization term. Το πρόβλημα παλινδρόμησης που επιλύεται μέσω του critic, συνδυάζει μια εκτίμηση της ανταμοιβής G_t με τη μέθοδο Temporal difference (υπάρχει η δυνατότητα να χρησιμοποιηθούν και άλλες μέθοδοι, όπως Monte Carlo και της Γενικευμένης Εκτίμησης Πλεονεκτήματος) και την προβλεπόμενη τιμή του δικτύου critic.

0.2 Αρχιτεκτονική

Αυτή η ενότητα παρέχει την πλήρη αρχιτεκτονική του προτεινόμενου συστήματος, προκειμένου να δείξει πώς τα διάφορα στοιχεία που αναλύθηκαν προηγουμένως αλληλεπιδρούν για τη διαχείριση και τη βελτιστοποίηση της κατανομής πόρων στο Kubernetes. Η αρχιτεκτονική αποτελείται από 6 κύρια επίπεδα τα οποία φαίνονται παρακάτω:



Σχήμα 0.8. *Προτεινόμενο σύστημα για τη κανομή των πόρων.*

0.2.1 Τα 3 πρώτα επίπεδα

Το **πρώτο επίπεδο** της αρχιτεκτονικής αποτελεί τον εξωτερικό φόρτο εργασίας, δηλαδή τους χρήστες ή συστήματα που εκτελούν αιτήματα προς τις μικροϋπηρεσίες του συστήματος που μελετάται. Για τη δοκιμή του συστήματος επιλέχθηκε η λειτουργία ComposePosts, μέσω της οποίας δημιουργούνται και δημοσιεύονται αναρτήσεις σε ένα εικονικό κοινωνικό δίκτυο. Για τη δημιουργία πολλαπλών χρηστών χρησιμοποιήθηκε ο HTTP workload generator wrk2 [27] και το αντίστοιχο Lua script compose-post.lua, όπως προτείνεται στο framework του DeathStarBench [8], το οποίο είναι μια ολοκληρωμένη σουίτα από benchmark ανοιχτού κώδικα που περιλαμβάνει πέντε ρεαλιστικές, end-to-end εφαρμογές cloud που έχουν κατασκευαστεί χρησιμοποιώντας μικροϋπηρεσίες. Κατασκευασμένη με ευρέως υιοθετημένα πλαίσια όπως το Apache Thrift [28] και το gRPC [29], η σουίτα μοντελοποιεί αντιπροσωπευτικές υπηρεσίες, όπως ένα κοινωνικό δίκτυο, μια πλατφόρμα ηλεκτρονικού εμπορίου (e-commerce), έναν ιστότοπο ανασκόπησης και ροής μέσων, ένα ασφαλές τραπεζικό σύστημα και μια υπηρεσία συντονισμού drone που βασίζεται στο IoT. Σε αντίθεση με προηγούμενα benchmark που επικεντρώνονται σε απλές ή μονοεπίπεδες υπηρεσίες, το DeathStarBench καταγράφει την πολυπλοκότητα και τις αλληλεξαρτήσεις των μικροϋπηρεσιών του πραγματι-

κού κόσμου σε κλίμακα, με κάθε εφαρμογή να περιλαμβάνει δεκάδες μικροϋπηρεσίες που έχουν κατασκευαστεί και γραφτεί σε ένα μείγμα γλωσσών προγραμματισμού.

Το **δεύτερο επίπεδο** αφορά τα συστήματα παρακολούθησης. Το Prometheus συλλέγει μετρικές χρήσης (π.χ. CPU, μνήμη,) από κάθε μικροϋπηρεσία, επιτρέποντας την παρακολούθηση της απόδοσης και την έγκαιρη ανίχνευση συμφόρησης ή σφαλμάτων. Παράλληλα, το Jaeger καταγράφει ίχνη (traces) που αναπαριστούν τη ροή των αιτήσεων μεταξύ των μικροϋπηρεσιών και συμβάλλει ιδιαίτερα στη κατασκευή γράφων εξαρτήσεων, αποκαλύπτοντας τη δομή και τις σχέσεις καλέσαντος-καλούμενου (caller–callee) μεταξύ των υπηρεσιών. Τα δεδομένα αυτά είναι κρίσιμα για την κατανόηση της συμπεριφοράς του συστήματος, την εντοπισμό σημείων συμφόρησης, και αποτελούν τη βάση για την ανάλυση της κρίσιμης διαδρομής[1] που ακολουθεί.

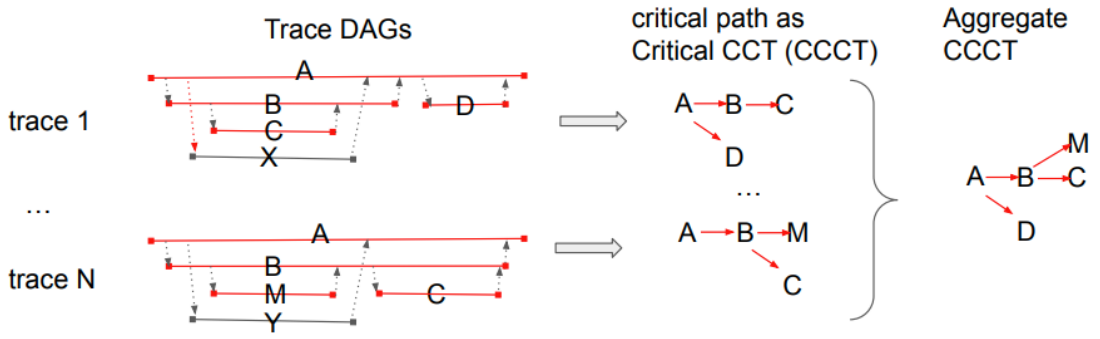
Αφού συλλεχθούν τα ίχνη, μεταβιβάζονται στο **τρίτο επίπεδο**, όπου χρησιμοποιείται το εργαλείο CRISP, το οποίο είναι ένα προηγμένο εργαλείο που υπολογίζει αποτελεσματικά την πολυπλοκότητα των σύγχρονων συστημάτων μικροϋπηρεσιών, απομονώνοντας την κρίσιμη αλυσίδα διαδρομών των κλήσεων υπηρεσίας, η απόδοση των οποίων καθορίζει άμεσα τη συνολική καθυστέρηση ενός αιτήματος. Η ουσία του CRISP έγκειται στην ικανότητά του να χειρίζεται αποτελεσματικά όλες τις κρίσιμες διαδρομές σε συστήματα μεγάλης κλίμακας και να φέρνει στην επιφάνεια χρήσιμες πληροφορίες σχετικά με το λειτουργικό περιβάλλον που βασίζεται στο cloud. Αυτό το εργαλείο αξιοποιεί τη δομή των ιχνών Jaeger για να βρει και να υπολογίσει την κρίσιμη διαδρομή με τον ακόλουθο ψευδοαλγόριθμο:

```
def CP(root):
  path = [root]
  if len(root.child) == 0:
    return path
  children = sortDescendingByEndTime(root.children)
  lfc = children[0]
  path.extend(CP(lfc))
  for c in children[1:]:
    if happensBefore(c, lfc):
      path.extend(CP(c))
      lfc = c
  return path
```

Σχήμα 0.9. *Ψευδοαλγόριθμος εξαγωγής κρίσιμης διαδρομής, Πηγή: [7].*

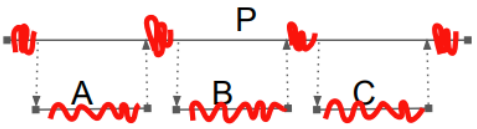
Το πλαίσιο του CRISP επιτρέπει επιπλέον την οπτικοποίηση του συνολικού Critical Calling Context Tree (CCCT), δηλαδή τη συγχώνευση των κρίσιμων διαδρομών των τελευταίων N ιχνών:

[1] Υποθέτοντας ένα σταθμισμένο κατευθυνόμενο ακυκλικό γράφο (DAG) G(V,E) με V κορυφές και E ακμές και την αρχική κορυφή να είναι το S και την τελική το Z, μια διαδρομή μέγιστου βάρους από το S στο Z σε ένα γράφημα εξαρτώμενο από εργασίες G(V,E) ονομάζεται **κρίσιμη διαδρομή** και το G ενδέχεται να περιέχει περισσότερες από μία από αυτές[30].

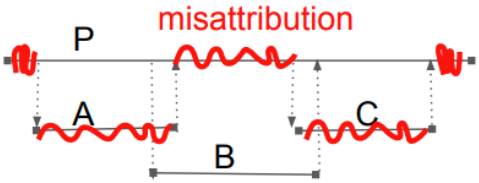


Σχήμα 0.10. *Συγχώνευση κρίσιμων διαδρομών (CCCT), Πηγή: [7].*

Καθώς κάθε κόμβος περιλαμβάνει ποσοτικά δεδομένα, και όλες οι κλήσεις ξεκινούν από ένα κοινό σημείο, οι διαδρομές έχουυν τη δυνατότητα να συγχωνευτούν σε ένα δέντρο συμφραζομένων. Το CCCT αποτυπώνει ποιοι δρόμοι εμφανίζονται πιο συχνά ως κρίσιμοι, με βάρη στους κόμβους να αντιστοιχούν στο άθροισμα των βαρών κάθε μονοπατιού. Συγκεκριμένα με αυτήν την ανάλυση, το πλαίσιο του CRISP παρέχει πολλά ποσοστά όπως τα εκατοστημόρια P50, P95, P99. Τέλος, το ιδιαίτερα σημαντικό πρόβλημα της μετατόπισης του ρολογιού (clock drift) που συχνά παρουσιάζεται σε τέτοια συστήματα αντιμετωπίζεται με την ψευδοσυνάρτηση happensBefore, όπως φαίνεται στον ψευδοαλγόριθμο παραπάνω. Η οπτική απεικόνιση τέτοιων προβλημάτων φαίνεται παρακάτω:



Σχήμα 0.11. *Ιδανικά ίχνη - οι κόκκινες γραμμές δείχνουν την κρίσιμη διαδρομή, Πηγή: [7]*

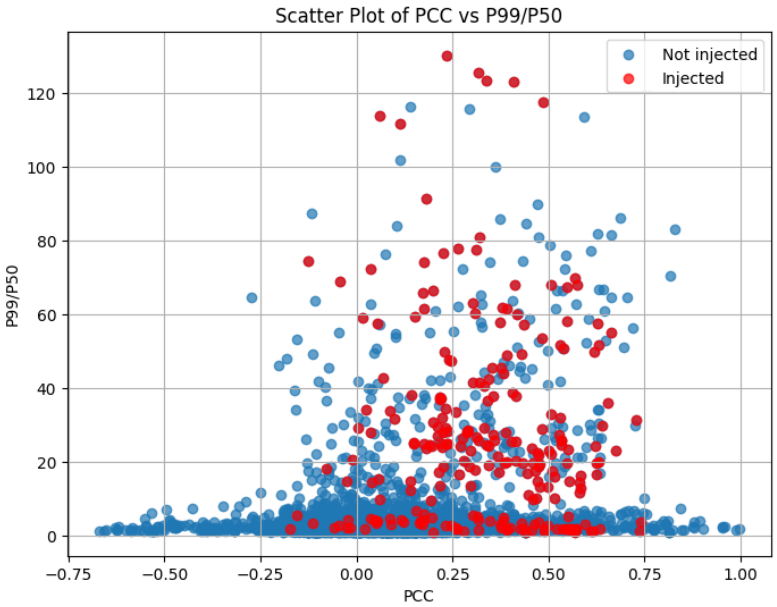


Σχήμα 0.12. *Μη ιδανικά ίχνη λόγω χρονικής απόκλισης, Πηγή: [7]*

0.2.2 Τα 3 τελευταία επίπεδα

Για την αξιοποίηση των στατιστικών προτύπων που προκύπτουν από τις κρίσιμες διαδρομές κάθε υπηρεσίας, επιλέχθηκε η τεχνική επιβλεπόμενης μηχανικής μάθησης με Support Vector Machine (SVM) στο **τέταρτο επίπεδο**, λόγω της ικανότητάς να γενικεύει σε περιβάλλοντα υψηλής διαστασιμότητας ακόμη και με απώλεια μεγάλων συνόλων δεδομένων. Στόχος της συγκεκριμένης τεχνικής είναι η ταξινόμηση των πιο πιθανολογούμενων υπαιτίων εντός της κρίσιμης διαδρομής, η οποία θα συμβάλλει θετικά στην έπειτα ανάλυση της Ενισχυτικής μάθησης. Για να επιτευχθεί αυτό, ακολουθώντας την ανάλυση που πραγματοποιήθηκε στο paper του FIRM [31], εξάγονται δύο βασικά χαρακτηριστικά, το Relative Importance, που υπολογίζεται μέσω του συντελεστή συσχέτισης Pearson[32] μεταξύ καθυστέρησης υπηρεσίας και συνολικής καθυστέρησης (γνωστό και ως explained variance [33]), και το Congestion Intensity, δηλαδή ο λόγος καθυστέρησης του εκατοστημορίου P99 προς το P50, το οποίο δείχνει την ικανότητα διαχείρισης φορτίου από το συγκεκριμένο Service. Τα δεδομένα

που δίνονται στο SVM προέρχονται από ελεγχόμενα πειράματα με χρήση του chaos-mesh [34], όπου εφαρμόστηκαν διάφορες καταπονήσεις σε διάφορα στοιχεία (π.χ. CPU stress) σε τυχαία Pods, ενώ παράλληλα τα ίχνη καταγράφτηκαν από το Jaeger σε συνθήκες σταθερού φόρτου (π.χ. 6 RPS για 5 λεπτά). Οι παρεμβάσεις αυτές βασίστηκαν στην τεχνική Chaos Engineering, η οποία χρησιμοποιείται για την προσομοίωση ρεαλιστικών σεναρίων καταπόνησης. Κάποια από τα παραπάνω ίχνη φαίνονται στο παρακάτω Σχήμα:



Σχήμα 0.13. *Διάγραμμα διασποράς ιχνών με παρεμβάσεις.*

Όπως φαίνεται στο παραπάνω διάγραμμα, η χρήση γραμμικού μοντέλου για την ταξινόμηση των δεδομένων τέτοιου συστήματος είναι αρκετά δύσκολη, γεγονός που δικαιολογεί τη χρήση πυρήνων kernels.Ταυτόχρονα, επειδή η έξοδος του ταξινομητή ενσωματώνεται αργότερα στο παρατηρήσιμο περιβάλλον του πράκτορα ενισχυτικής μάθησης, αντί για δυαδική ταξινόμηση (0 ή 1), χρησιμοποιήθηκε ως βελτίωση η προσέγγιση πιθανολογικής ταξινόμησης, με πιθανότητα για το αν μια μικροϋπηρεσία είναι υπαίτια. Έτσι, ακόμη και μερικώς υπεύθυνες υπηρεσίες, δηλαδή με μικρή πιθανότητα, λαμβάνουν πόρους από τον πράκτορα αναλόγως. Για να επιτευχθεί αυτή η ανάλυση με την πιθανολογική ταξινόμηση, εφαρμόστηκε **calibration** στον SVM ταξινομητή, μετατρέποντας τις εξόδους του σε πιθανότητες στο διάστημα $[0, 1]$. Το calibration επιτυγχάνεται με παλινδρόμηση, χρησιμοποιώντας το λογιστικό μοντέλο του Platt [35] όπως φαίνεται παρακάτω:

$$p(label = 1 \mid f_i) = \frac{1}{1 + \exp(Af_i + B)}, \tag{19}$$

όπου label είναι η πραγματική ετικέτα και f_i η έξοδος του αρχικού (un-calibrated) ταξινομητή για το δείγμα i. Οι παράμετροι A και B προκύπτουν με μέγιστη πιθανοφάνεια (maximum likelihood). Για να εφαρμοστεί η τεχνική σε ήδη εκπαιδευμένο μοντέλο, χρησιμοποιήθηκαν οι κλάσεις FrozenEstimator και CalibratedClassifierCV από το scikit-learn

[36]. Η πιθανολογική ταξινόμηση εφαρμόζεται σε κάθε μικροϋπηρεσία ξεχωριστά και η τελική πιθανότητα ενσωματώνεται στο παρατηρήσιμο χώρο του πράκτορα Ενισχυτικής μάθησης.

Όπως αναφέρθηκε σε προηγούμενα κεφάλαια, η Ενισχυτική Μάθηση (RL) είναι κατάλληλη για συστήματα λήψης αποφάσεων, καθώς επιτρέπει τη μάθηση μέσω εμπειρίας και τη συνεχή βελτίωση της απόδοσης σε δυναμικά περιβάλλοντα όπως το cloud. Έχοντας καλύψει τη θεωρητική βάση των RL agents και αναλύσει πρακτικά τα προηγούμενα επίπεδα, το επόμενο βήμα είναι η πρακτική ενσωμάτωση της RL στο περιβάλλον του Kubernetes. Για να αλληλεπιδρά ο RL agent (**πέμπτο επίπεδο**) με το Kubernetes, δημιουργήθηκε μια προσαρμοσμένη κλάση περιβάλλοντος σύμφωνα με το πρότυπο της βιβλιοθήκης Gymnasium [37]. Αυτό σημαίνει ότι η κλάση υλοποιήθηκε με τις βασικές μεθόδους:

- **__init__()**: Κληρονομεί από gymnasium.Env και ορίζει τα απαραίτητα:
  - **observation_space**: Τύπος παρατήρησης (π.χ. Box, Discrete).
  - **action_space**: Ενέργειες που ενδέχεται να εκτελεστούν, εδώ ορίστηκε ως MultiDiscrete[num_services, num_actions] για τροποποίηση του αριθμού των Pods ανά υπηρεσία.
  - Εσωτερικές μεταβλητές κατάστασης και μετρικές.
- **reset()**: Επαναφέρει το περιβάλλον στην αρχική του κατάσταση και επιστρέφει υποχρεωτικά σε μορφή tuple:
  - **observation**: Αρχική παρατήρηση.
  - **info**: Επιπρόσθετες πληροφορίες (προαιρετικό).
- **step()**: Εκτελεί την ενέργεια και προχωρά το περιβάλλον κατά ένα βήμα επιστρέφοντας υποχρεωτικά σε μορφή tuple:
  - **observation**: Η παρατήρηση από το περιβάλλον.
  - **reward**: Η αριθμητική ανταμοιβή που υπολογίστηκε είτε με δυναμική είτε με στατική συνάρτηση.
  - **terminated**: True εάν τελείωσε με επιτυχία το επεισόδιο.
  - **truncated**: True εάν τελείωσε το επεισόδιο λόγω κάποιου περιορισμού.
  - **info**: Επιπρόσθετη πληροφορία.

Σημαντικό είναι οι `observation_space` και `action_space` να είναι συμβατοί με τις εξόδους του `reset()` και τις εισόδους του `step()`. Όλες οι υποχρεωτικές μέθοδοι (συμπεριλαμβανομένων των `render()`, `close()`) περιγράφονται στο documentation του Gymnasium. Η αρχική δομή του περιβάλλοντος και της κλάσης ανάπτυξης βασίστηκε στο paper gym-hpa [38], η οποία είχε υλοποιηθεί με παλαιότερη έκδοση της Gymnasium (gym). Στη συνέχεια, προσαρμόστηκε στην έκδοση 1.1.1 ώστε να υποστηρίζει το DeathStarBench και τις τεχνικές της παρούσας εργασίας.

Παράλληλα, τα μοντέλα RL (PPO, TRPO, A2C) υλοποιήθηκαν με τις σταθερές και αξιόπιστες βιβλιοθήκες stable-baselines3 [39] και stable-baselines3-contrib [40], έκδοσης 2.6.0, με τις ακόλουθες υπερπαραμέτρους για κάθε μοντέλο:

Table 1. *Υπερπαράμετροι υλοποίησης του αλγορίθμου PPO μέσω του Stable-Baselines3*

Παράμετρος	**Περιγραφή**	**Πειραματική Τιμή**
`policy`	Policy model	MlpPolicy
`env`	Gymnasium environment (can be vectorized)	SocialNetwork
`learning_rate`	Step size for optimizer	0.0003
`n_steps`	Steps per environment per update	150
`batch_size`	Minibatch size	64
`n_epochs`	Number of passes per update	10
`gamma`	Discount factor	0.99
`gae_lambda`	GAE parameter	0.95
`clip_range`	Policy loss clipping parameter	0.2
`clip_range_vf`	Value function clipping	None
`normalize_advantage`	Normalize advantages	True
`ent_coef`	Entropy bonus coefficient	0.0
`vf_coef`	Value function loss coefficient	0.5
`max_grad_norm`	Gradient clipping norm	0.5
`use_sde`	State-Dependent Exploration	False
`sde_sample_freq`	SDE noise resampling frequency	-1
`rollout_buffer_class`	Custom rollout buffer class	None
`rollout_buffer_kwargs`	Rollout buffer arguments	None
`target_kl`	KL divergence early stopping	None
`stats_window_size`	Stats averaging window size	100
`tensorboard_log`	Path for TensorBoard logs	tensorboard_log_path
`policy_kwargs`	Policy keyword arguments	None
`verbose`	Verbosity level	1
`seed`	Random seed	None
`device`	Training device	auto
`_init_setup_model`	Build model on creation	True

Table 2. *Υπερπαράμετροι υλοποίησης του αλγορίθμου TRPO μέσω του Stable-Baselines3 contrib*

Παράμετρος	**Περιγραφή**	**Πειραματική Τιμή**
`policy`	Policy model	MlpPolicy
`env`	Gymnasium environment	SocialNetwork
`learning_rate`	Step size for optimizer	0.0003
`n_steps`	Steps per environment per update	150
`batch_size`	Minibatch size	128
`gamma`	Discount factor	0.99
`gae_lambda`	GAE parameter	0.95
`ent_coef`	Entropy bonus coefficient	0.0
`vf_coef`	Value function loss coefficient	0.5
`max_kl`	Maximum KL divergence for updates	0.01
`cg_damping`	Conjugate gradient damping factor	0.1
`cg_max_steps`	Maximum conjugate gradient iterations	10
`line_search_coef`	Coefficient for line search	0.8
`n_cpu_tf_sess`	Number of CPU threads for TensorFlow session	1
`normalize_advantage`	Normalize advantages	True
`tensorboard_log`	Path for TensorBoard logs	tensorboard_log_path
`policy_kwargs`	Policy keyword arguments	None
`verbose`	Verbosity level	1
`seed`	Random seed	None
`device`	Training device	auto
`_init_setup_model`	Build model on creation	True

Table 3. *Υπερπαράμετροι υλοποίησης του αλγορίθμου A2C μέσω του Stable-Baselines3*

Παράμετρος	**Περιγραφή**	**Πειραματική Τιμή**
`policy`	Policy model	MlpPolicy
`env`	Gymnasium environment	SocialNetwork
`learning_rate`	Step size for optimizer	0.0007
`n_steps`	Steps per environment per update	150
`gamma`	Discount factor	0.99
`gae_lambda`	GAE parameter	1.0
`ent_coef`	Entropy bonus coefficient	0.0
`vf_coef`	Value function loss coefficient	0.5
`max_grad_norm`	Gradient clipping norm	0.5
`rms_prop_epsilon`	RMSProp optimizer epsilon	1e-5
`use_rms_prop`	Whether to use RMSProp optimizer	True
`normalize_advantage`	Normalize advantages	False
`tensorboard_log`	Path for TensorBoard logs	tensorboard_log_path
`policy_kwargs`	Policy keyword arguments	None
`verbose`	Verbosity level	1
`seed`	Random seed	None
`device`	Training device	auto
`_init_setup_model`	Build model on creation	True

Η συνάρτηση ανταμοιβής που χρησιμοποιήθηκε για την εκπαίδευση των πρακτόρων με στόχο την αργή και σωστή εκπαίδευση είναι η ακόλουθη:

$$\mathrm{R} = \sum_{d \in D} [a \cdot SLO_d + (1 - a) \cdot Align_d] \tag{20}$$

όπου

$$SLO_d = 1 - w_d \tag{21}$$

$$Align_d = \frac{1}{1 + |p_d - r_d|} \tag{22}$$

$$D = \text{set of deployments} \tag{23}$$

$$p_d = \text{number of pods for deployment } d \in D \tag{24}$$

$$r_d^{(\text{cpu})} = \text{desired number of replicas for deployment } d \in D \tag{25}$$

$$= \left\lceil p_d \cdot \frac{\text{cpu\_usage}}{\text{cpu\_target\_usage}} \right\rceil \text{(same for memory)} \tag{26}$$

$$\tag{27}$$

$$w_d = \text{critical weight of deployment } d \in D \tag{28}$$

$$a \in [0, 1] \text{ is the weighting factor} \tag{29}$$

$$\tag{30}$$

$$\text{cpu\_target\_usage} = p_d \cdot \text{cpu\_target} \tag{31}$$

$$\text{mem\_target\_usage} = p_d \cdot \text{mem\_target} \tag{32}$$

$$\text{cpu\_target} = \text{threshold} \cdot \text{cpu\_request} \tag{33}$$

$$\text{threshold} = 0.75 \text{ (same as KHPA)} \tag{34}$$

$$\text{cpu\_request} = \text{initial CPU request of deployment.} \tag{35}$$

Για να διασφαλιστεί ότι οι agents αλληλεπιδρούν με το πραγματικό σύστημα Kubernetes μέσω της προσαρμοσμένης κλάσης περιβάλλοντος, το **έκτο επίπεδο**, το οποίο αποτελείται από μια άλλη κλάση με όνομα DeploymentStatus, γεφυρώνει το κενό μεταξύ των δύο εξασφαλίζοντας τη σωστή επικοινωνία μέσω του Kubernetes API. Οι κύριες μέθοδοι που ορίζονται μέσα σε αυτή την κλάση είναι οι εξής:

- **__init__()**: Αρχικοποιεί τις απαραίτητες πληροφορίες που πρέπει να περιλαμβάνονται για κάθε Service, όπως cpu_request, cpu_limit, max_pods, min_pods κ.ά., καθώς και την τιμή critical_weight, η οποία υποδηλώνει την πιθανότητα να είναι υπαίτιο, όπως αναφέρθηκε προηγουμένως. Οι αρχικές τιμές ορίζονται ξεχωριστά για κάθε Service, όπως φαίνεται παρακάτω:

Table 4. *Αρχική διαμόρφωση πόρων των μικροϋπηρεσιών του SocialNetwork Kubernetes*

Service	CPU (αιτούμενο/όριο)(MB)	RAM (αιτούμενο/όριο)(MB)
compose-post-service	100 / 300	100 / 300
home-timeline-redis	200 / 300	200 / 300
home-timeline-service	100 / 300	100 / 300
jaeger	100 / 300	600 / 800
media-frontend	100 / 300	100 / 300
media-memcached	200 / 300	200 / 300
media-mongodb	200 / 300	200 / 300
media-service	100 / 300	100 / 300
nginx-thrift	200 / 300	200 / 300
post-storage-memcached	200 / 300	200 / 300
post-storage-mongodb	200 / 300	200 / 300
post-storage-service	100 / 300	100 / 300
social-graph-mongodb	200 / 300	200 / 300
social-graph-redis	200 / 300	200 / 300
social-graph-service	100 / 300	100 / 300
text-service	100 / 300	100 / 300
unique-id-service	100 / 300	100 / 300
url-shorten-memcached	200 / 300	200 / 300
url-shorten-mongodb	200 / 300	200 / 300
url-shorten-service	100 / 300	100 / 300
user-memcached	200 / 300	200 / 300
user-mongodb	200 / 300	200 / 300
user-service	100 / 300	100 / 300
user-mention-service	100 / 300	100 / 300
user-timeline-mongodb	200 / 300	200 / 300
user-timeline-redis	200 / 300	200 / 300
user-timeline-service	100 / 300	100 / 300

- **update_obs_k8s()**: Καλείται κυρίως πριν τον υπολογισμό της ανταμοιβής στη μέθοδο step() για να εξασφαλιστούν οι σωστές τιμές για την παρατήρηση του περιβάλλοντος. Εκτελεί ερωτήματα προς το Prometheus και ανακτά τις απαραίτητες μετρικές που θα χρησιμοποιηθούν στη συνέχεια για τον υπολογισμό των επιθυμητών Pod replicas για κάθε Service.

- **update_replicas()**: Καλείται μέσα στη μέθοδο update_obs_k8s(). Ο βασικός σκοπός

είναι ο υπολογισμός των επιθυμητών replicas για κάθε Service με βάση τις μετρικές που ανακτήθηκαν προηγουμένως.

- **update_deployment()**: Ενημερώνει το Deployment μέσω του Kubernetes API και στη συνέχεια καλεί τη μέθοδο patch_deployment() για την εφαρμογή της ενημέρωσης.

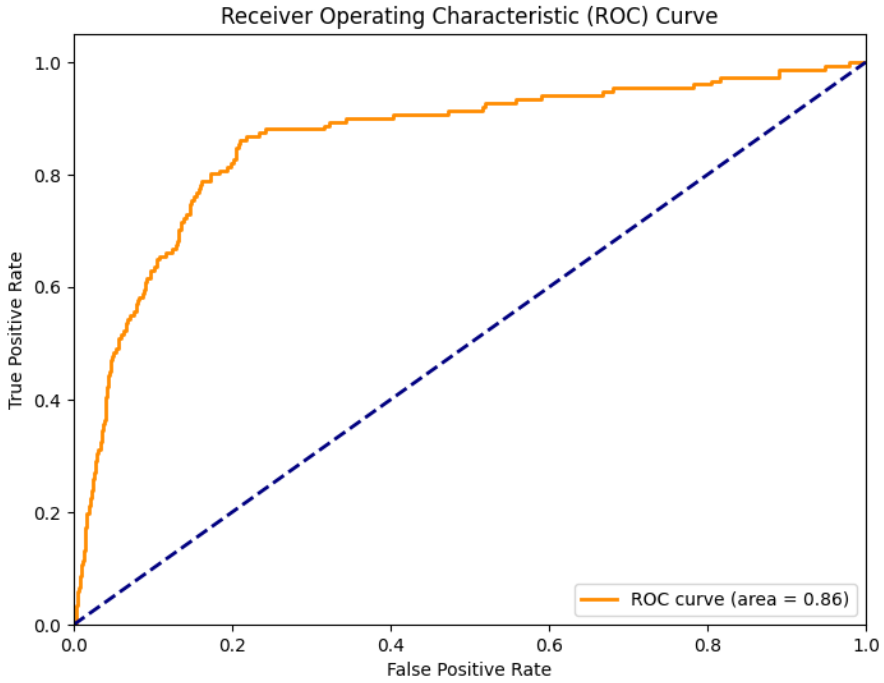
- **patch_deployment()**: Εφαρμόζει αλλαγές στο Deployment μέσω του Kubernetes API.

- **deploy_pod_replicas()**: Καλείται μέσα από το προσαρμοσμένο περιβάλλον από τον Agent όταν η ενέργεια είναι η προσθήκη replica Pods σε κάποιο Service. Καλεί τη update_deployment() για την υλοποίηση της ενέργειας.

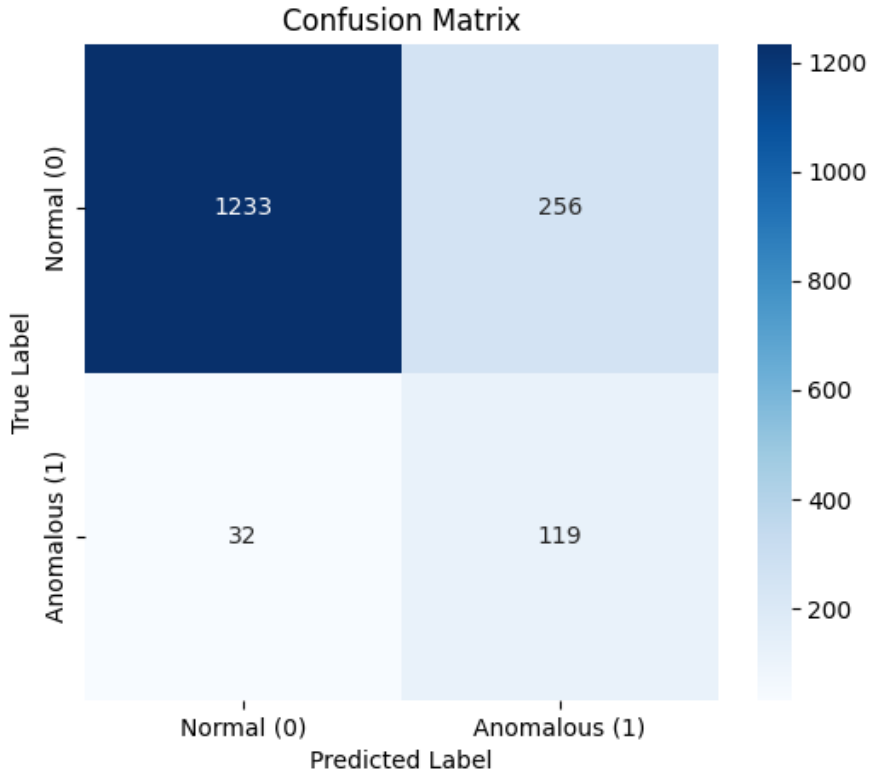
- **terminate_pod_replicas()**: Καλείται μέσα από το προσαρμοσμένο περιβάλλον από τον Agent όταν η ενέργεια είναι η αφαίρεση replica Pods από κάποιο Service. Καλεί τη update_deployment() για την υλοποίηση της ενέργειας.

0.3 Αξιολόγηση μελέτης

Σε αυτό το κεφάλαιο παρουσιάζονται τα αποτελέσματα ξεχωριστά για κάθε κομμάτι του σχήματος αλλά και συνολικά με βάση κάποιες σημαντικές μετρικές αξιολόγησης όπως είναι η από άκρο-σε-άκρο καθυστέρηση του συστήματος ή ο μέσος όρος διαχειριζομένων Pods στο σύστημα. Ξεκινώντας την αξιολόγηση της μελέτης, αρχικά παρουσιάζεται η επίδοση μιας σημαντικής ενότητας αυτής της διπλωματικής εργασίας, που είναι ο ταξινομητής SVM. Ο συγκεκριμένος ταξινομητής εκπαιδεύτηκε σε καλά επισημασμένα δεδομένα με τη βιβλιοθήκη που αναφέρθηκε προηγουμένως και ταυτόχρονα, για τον σκοπό αυτό, τα πειράματα διεξήχθησαν με λίγα δεδομένα, προκειμένου να αποφευχθεί η υπερπροσαρμογή (overfit), καθώς είναι ένα από τα γνωστά προβλήματα αυτών των ταξινομητών. Τα αποτελέσματα αξιολόγησης παρουσιάζονται κυρίως χρησιμοποιώντας την καμπύλη ROC και τον πίνακα σύγχυσης, τα οποία παρέχουν συμπληρωματικές πληροφορίες για την αποτελεσματικότητα του ταξινομητή. Η καμπύλη ROC απεικονίζει την ισορροπία μεταξύ ευαισθησίας (sensitivity) και ειδικότητας (specificity) σε διαφορετικά κατώφλια. Ταυτόχρονα, ο πίνακας σύγχυσης παρέχει μια σαφή εικόνα των αποτελεσμάτων ταξινόμησης, με ιδιαίτερη έμφαση στην απουσία ανώμαλων δεδομένων που κατηγοριοποιούνται, λανθασμένα, ως φυσιολογικά. Αυτή η ιδιότητα είναι κρίσιμη για την αξιοπιστία του συνολικού συστήματος, καθώς τα ψευδώς θετικά δεδομένα θα είχαν τη δυνατότητα ενδεχομένως να βλάψουν το σύστημα και επίσης να παραπλανήσουν τον πράκτορα ενισχυτικής μάθησης.



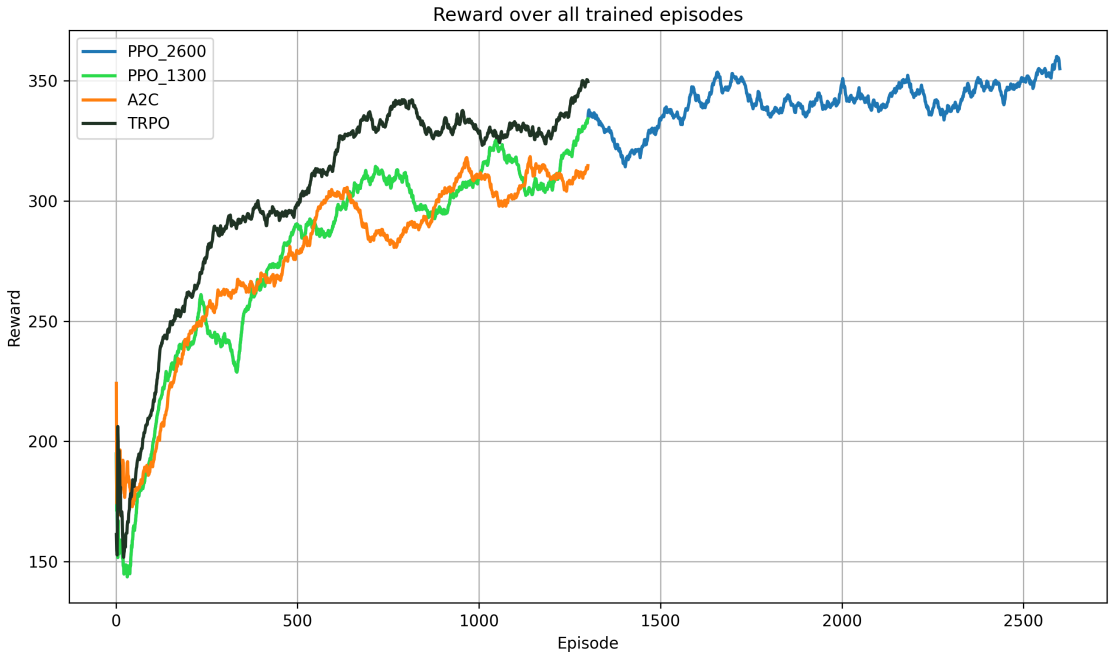
Σχήμα 0.14. *Καμπύλη ROC του ταξινομητή.*



Σχήμα 0.15. *Πίνακας σύγχυσης του ταξινομητή.*

Σε αυτό το σημείο, ο ταξινομητής έχει εκπαιδευτεί και έχει επιτύχει ικανοποιητική απόδοση προκειμένου να χειριστεί κατάλληλα και να επισημάνει τα κρίσιμα στοιχεία του κρίσιμου μονοπατιού και με αυτόν τον τρόπο να βοηθήσει τον πράκτορα ενισχυτικής μάθησης στη σωστή απόφαση διάθεσης κατάλληλων πόρων στα κατάλληλα Services. Μετά από αυτό το σημείο, ο ταξινομητής μετατράπηκε σε πιθανολογικός όπως περιγράφτηκε σε προηγούμενα κεφάλαια.

Στη συνέχεια, η διαδικασία εκπαίδευσης του πράκτορα ενισχυτικής μάθησης πραγματοποιήθηκε χρησιμοποιώντας το cluster Kubernetes που αναφέρθηκε προηγουμένως στο benchmark του SocialNetwork του DeathStarBench. Κάθε πράκτορας εκπαιδεύτηκε για 1300 επεισόδια και μέγιστο 20 βήματα ανά επεισόδιο λόγω του υψηλού κόστους (χρονικά) εκπαίδευσης στο πραγματικό cluster. Πιο συγκεκριμένα, το κόστος εκπαίδευσης των πρακτόρων PPO, A2C, TRPO ήταν περίπου 3.8, 4.3, 4.2 ημέρες αντίστοιχα. Όπως και με την εκπαίδευση, οι αρχικές συνθήκες του περιβάλλοντος διατηρήθηκαν τυχαίες, δίνοντας έμφαση και ελέγχοντας την ικανότητα τόσο των πρακτόρων όσο και του KHPA να προσαρμόζονται και να γενικεύονται στο δεδομένο πρόβλημα. Για να τονιστεί η σημασία της αύξησης της εκπαίδευσης των πρακτόρων RL σε τέτοια πολύπλοκα συστήματα, ο πράκτορας PPO εκπαιδεύτηκε περαιτέρω για 1300 επιπλέον επεισόδια, φτάνοντας τα συνολικά επεισόδια εκπαίδευσης σε 2600 επεισόδια και φτάνοντας τον χρόνο εκπαίδευσης σε συνολικά 8.11 ημέρες.



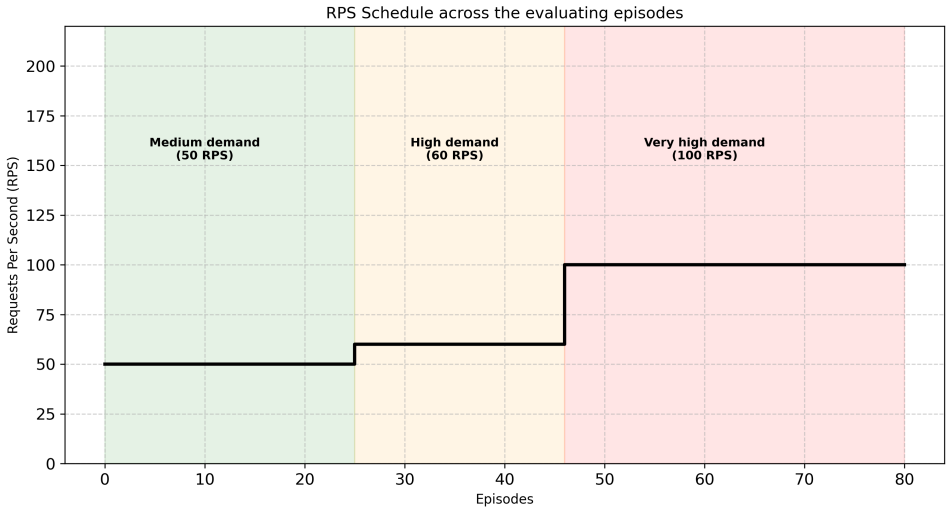
Σχήμα 0.16. *Αποτελέσματα εκπαίδευσης του πράκτορα ενισχυτικής μάθησης.*

Τα αποτελέσματα της εκπαίδευσης του πράκτορα ενισχυτικής μάθησης παρουσιάζονται παραπάνω με ομαλοποιημένες καμπύλες εκπαίδευσης χρησιμοποιώντας παράθυρο κινητού μέσου όρου 100 επεισοδίων για να δοθεί έμφαση στην τάση μάθησης.

Αρχικά, όλοι οι πράκτορες αρχίζουν να μαθαίνουν με χαμηλές τιμές ανταμοιβής και βελτιώνονται σταδιακά μέχρι να ολοκληρωθεί η εκπαίδευση. Ο πράκτορας TRPO δείχνει σαφώς την πιο γρήγορη και σταθερή βελτίωση σε όλη τη φάση εκπαίδευσης, ξεπερνώντας

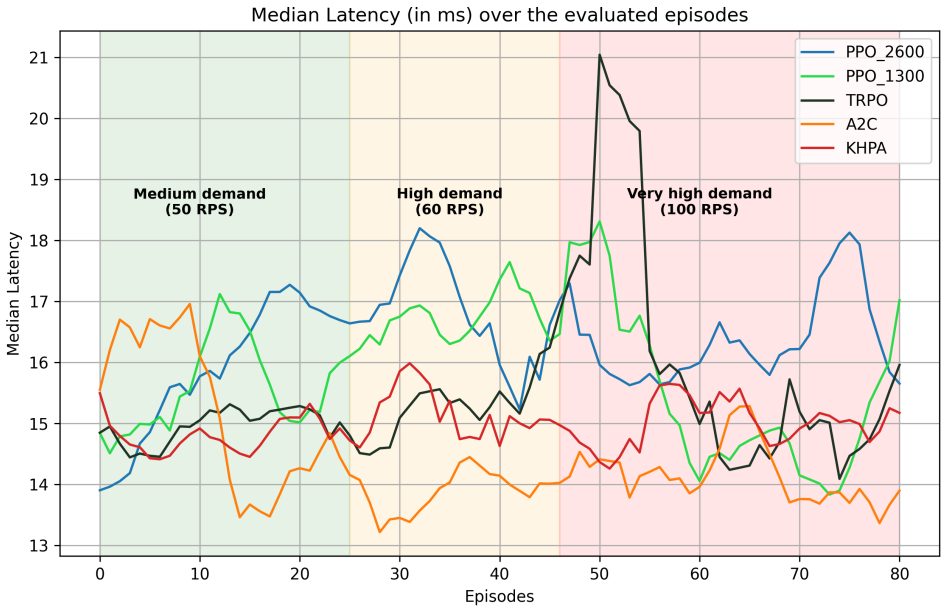
σταθερά τους άλλους πράκτορες. Ταυτόχρονα, ο πράκτορας PPO ακολουθεί μια παρόμοια καμπύλη μάθησης με τον TRPO, αλλά με πολλές διακυμάνσεις, ειδικά γύρω από το σημείο επεισοδίου 200-400. Τέλος, ο πράκτορας A2C ακολουθεί μια σταθερή καμπύλη μάθησης με μια μικρή πτώση γύρω από τα σημεία 600-800 και 1000-1100.

Παράλληλα, το μοτίβο φόρτου εργασίας που χρησιμοποιήθηκε με σκοπό τη δοκιμή του προαναφερθέντος σχήματος στο cluster Kubernetes, αποτελείται από τη γεννήτρια http ανοιχτού βρόχου wrk2, την ίδια που χρησιμοποιήθηκε για την εκπαίδευση των μοντέλων, με το ακόλουθο μοτίβο φόρτου εργασίας:



Σχήμα 0.17. *Μοτίβο φόρτου εργασίας σε ολόκληρη τη φάση αξιολόγησης.*

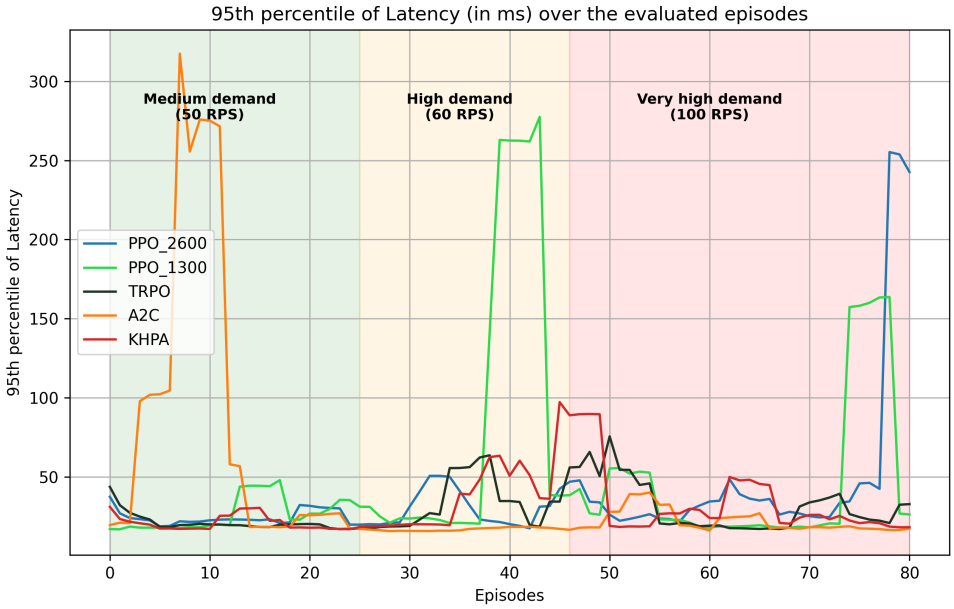
Το μοτίβο φόρτου εργασίας που χρησιμοποιήθηκε κατά τη φάση αξιολόγησης σχεδιάστηκε για να αντικατοπτρίζει ρεαλιστικές και απαιτητικές συνθήκες συστήματος. Συγκεκριμένα, τα αρχικά στάδια της αξιολόγησης υπέβαλαν το σύστημα σε μεσαία φορτία αιτημάτων, προσομοιώνοντας αρχικές περιόδους που συνήθως παρατηρούνται σε πραγματικές συνθήκες εός συστήματος, αλλά και το τελευταίο μέρος της αξιολόγησης παρουσιάζει μια υψηλότερη ζήτηση από τους χρήστες, υποθέτοντας επίσης ότι μετά από κάποιο χρονικό διάστημα υπάρχει αύξηση του φόρτου εργασίας στο σύστημα. Στην επόμενη ενότητα, υπάρχει μια αξιολόγηση των πρακτόρων χρησιμοποιώντας πολυάριθμες μετρήσεις όπως πολλά ποσοστά καθυστέρησης και επίσης τον αριθμό των Pods που αναπτύχθηκαν. Το πρώτο αντικατοπτρίζει τον πραγματικό αντίκτυπο από την πλευρά του χρήστη και το δεύτερο αντικατοπτρίζει το κόστος διατήρησης της προαναφερθείσας απόδοσης.



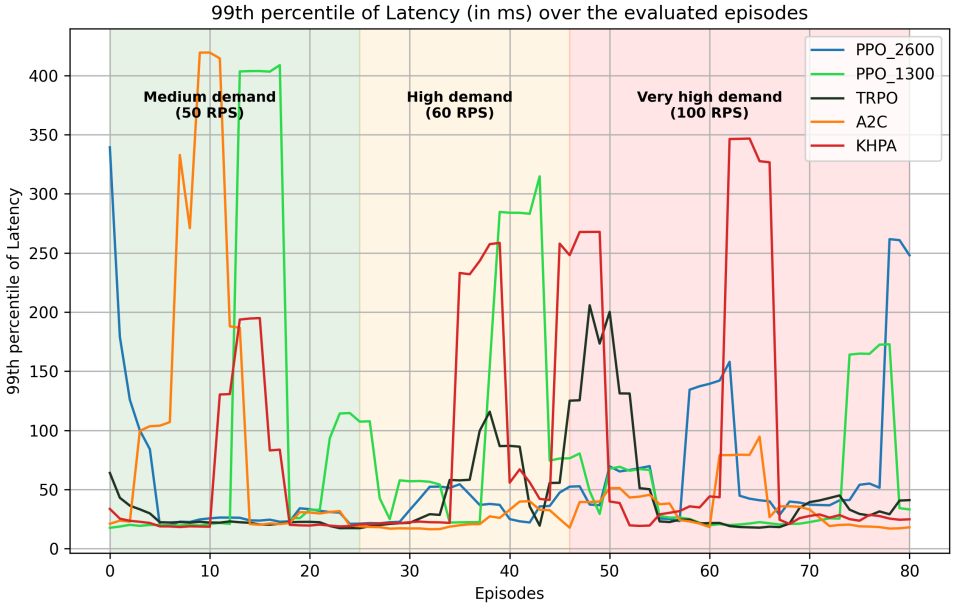
Σχήμα 0.18. *Διάμεσος καθυστέρησης σε ολόκληρη τη φάση αξιολόγησης.*

Τα αποτελέσματα των δοκιμών των πρακτόρων που φαίνονται στο παραπάνω διάγραμμα, ιδιαίτερα σε σύγκριση με την απόδοση του KHPA στην αρχική κατάσταση, είναι εύλογα λόγω του χαμηλού αριθμού επεισοδίων εκπαίδευσης και του χώρου υψηλής διάστασης του χώρου δράσης, όπως αναφέρεται και στο paper του FIRM ([31], σελίδα-12), όπου παρόμοια προβλήματα παρουσιάστηκαν υπό τέτοιες συνθήκες εκπαίδευσης και ιδιαίτερα σε τόσο χαμηλό, επεισοδιακά, χρόνο εκπαίδευσης. Αρχικά, φαίνεται ότι όλοι οι πράκτορες συμπεριφέρονται με τρόπο που ξεπερνούν εύλογα την απόδοση του KHPA στις μετρικές καθυστέρησης. Πιο συγκεκριμένα, ο πράκτορας A2C ξεκινά με αυξημένη καθυστέρηση, αλλά κοντά στο $10^{ο}$ επεισόδιο, μειώνει την καθυστέρηση κάτω από τη γραμμή βάσης του KHPA. Αυτό αναδεικνύει πως ο πράκτορας χρειάζεται χρόνο για να προσαρμοστεί στις πραγματικές συνθήκες του συστήματος και παρουσιάζει καλές ενέργειες όταν πραγματοποιηθεί αυτό. Ο πράκτορας TRPO φαίνεται να ταιριάζει με την απόδοση της γραμμής βάσης του KHPA σε όλο το στάδιο αξιολόγησης. Η σημαντική αύξηση που παρατηρήθηκε γύρω από την αλλαγή στο φόρτο εργασίας δικαιολογείται κάπως λόγω της ξαφνικής αλλαγής από 60 RPS σε 100 RPS. Τέλος, ο πρώτος πράκτορας PPO που εκπαιδεύτηκε για 1300 επεισόδια, αρχικά φαίνεται ότι δυσκολεύεται να προσαρμοστεί στις μετρήσεις του συστήματος όσον αφορά την καθυστέρηση, αλλά γύρω στο 55ο επεισόδιο αρχίζει να συμπεριφέρεται καλύτερα από τη γραμμή βάσης του KHPA. Επίσης, ο δεύτερος και πιο εκπαιδευμένος πράκτορας PPO δυσκολεύεται να δώσει καλά αποτελέσματα στις μετρήσεις μέσης καθυστέρησης, αν και αυτό εύκολα δικαιολογείται στα παρακάτω διαγράμματα των αναπτυγμένων Pod. Είναι αξιοσημείωτο ότι καθώς οι πράκτορες προσαρμόζονται στην αλλαγή του φόρτου εργασίας, η απόδοση γίνεται όχι μόνο συγκρίσιμη με την αρχική τιμή, αλλά είναι εύκολα παρατηρήρισμο ότι ξεπερνά την αρχική τιμή του KHPA, ειδικά από τον πράκτορα A2C. Όλα τα αποτελέσματα που σχετίζονται με την καθυστέρηση που φαίνονται στο παραπάνω διάγραμμα θα δικαιολογηθούν σε μεταγε-

νέστερο στάδιο, όπου θα παρουσιαστεί το κύριο διάγραμμα που απεικονίζει την ανάπτυξη των Pods εντός του Kubernetes cluster. Ταυτόχρονα, υπάρχει σαφής ένδειξη ότι ακόμη και με χαμηλό αριθμό επεισοδίων εκπαίδευσης, οι πράκτορες έχουν καταφέρει να μάθουν πολύτιμα μοτίβα, βοηθώντας στη μέση απόδοση καθυστέρησης από άκρο-σε-άκρο του συστήματος Kubernetes. Στη συνέχεια της ανάλυσης αυτού του κεφαλαίου και της απόδοσης των πρακτόρων, παρουσιάζονται τα διαγράμματα 95^{ου} και 99^{ου} εκατοστημορίου.

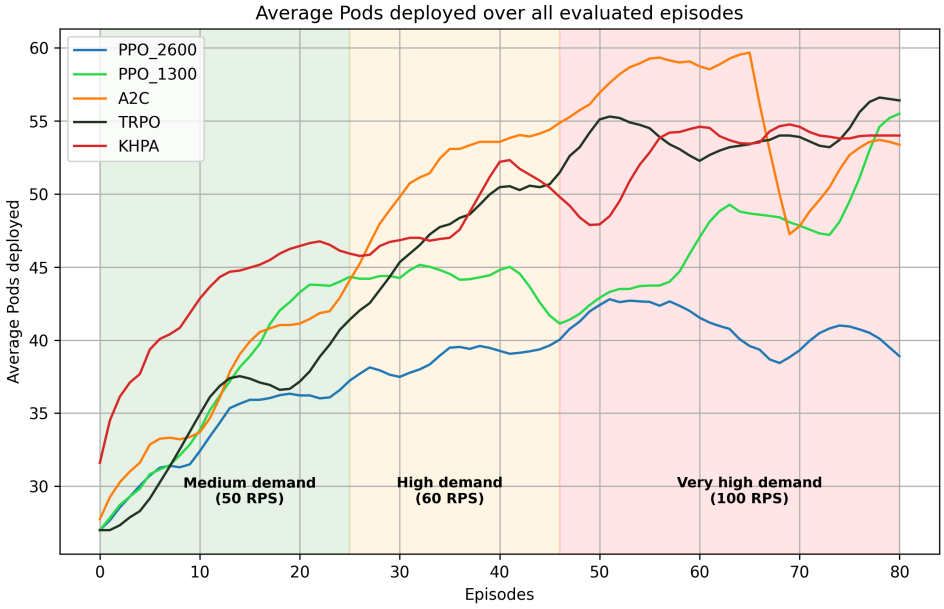


Σχήμα 0.19. *95^{ο} εκατοστημόριο της καθυστέρησης σε ολόκληρη τη φάση αξιολόγησης.*



Σχήμα 0.20. *99^{ο} εκατοστημόριο της καθυστέρησης σε ολόκληρη τη φάση αξιολόγησης*

Στο διάγραμμα του 95^{ου} εκατοστημορίου, παρατηρείται ότι τόσο ο πράκτορας A2C όσο και ο PPO που έχουν εκπαιδευτεί στα 1300 επεισόδια, παρουσιάζουν κάποιες ξαφνικές αιχμές κατά τα πρώτα στάδια της αξιολόγησης, υποδεικνύοντας την αδυναμία αποτελεσματικής διαχείρισης του μεγαλύτερου μέρους της ουράς αιτημάτων ανά πάσα στιγμή. Εξαιρώντας τα προαναφερθέντα σημεία, όλοι οι πράκτορες ξεπερνούν το KHPA σε πολλά σημεία του σταδίου αξιολόγησης. Τέλος, στο διάγραμμα του 99^{ου} εκατοστημορίου καθυστέρησης, το KHPA και όλοι οι πράκτορες έχουν αυξημένο αριθμό αιχμών καθυστέρησης, υπογραμμίζοντας την πρόκληση διατήρησης της βέλτιστης απόδοσης σε όλη την ουρά υπό ακραίες συνθήκες. Σημαντική βελτίωση αναδεικνύεται από τον πιο εκπαιδευμένο πράκτορα PPO, ο οποίος έχει μικρό αριθμό αιχμών υψηλής καθυστέρησης. Αυτό υποδηλώνει ότι, με επαρκή εκπαίδευση, ο πράκτορας PPO είναι σε θέση να χειρίζεται τις ουρές αιτημάτων πιο αποτελεσματικά και να διατηρεί χαμηλότερους χρόνους απόκρισης υπό μεταβαλλόμενες συνθήκες φόρτου. Για την περαιτέρω αξιολόγηση της αποδοτικότητας των πόρων, ο κύριος στόχος ανταμοιβής των πρακτόρων, ο μέσος αριθμός αναπτυχθέντων Pods και η γραμμή βάσης παρουσιάζονται στο ακόλουθο διάγραμμα:



Σχήμα 0.21. *Μέσος όρος αναπτυγμένων Pods σε ολόκληρη τη φάση αξιολόγησης.*

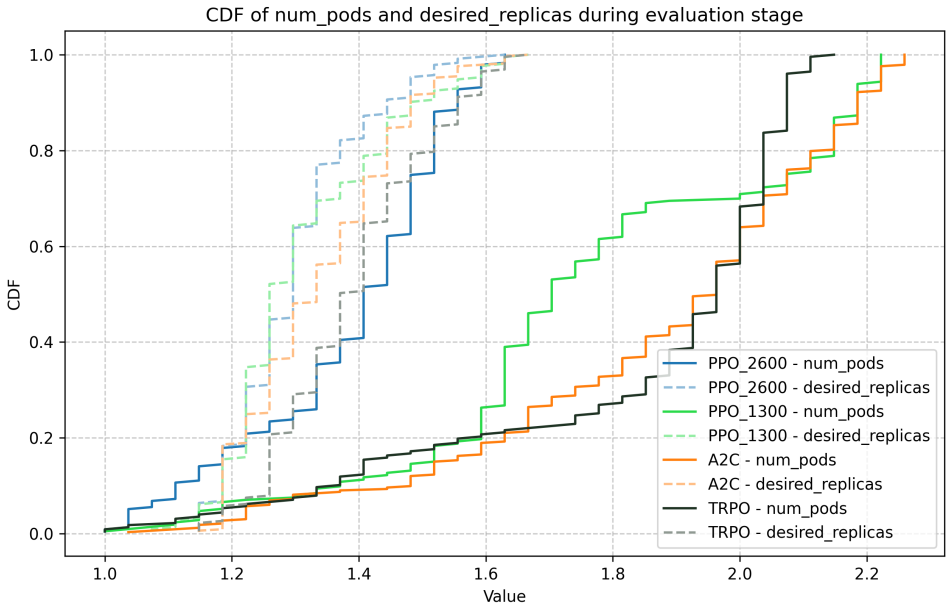
Το πρώτο διάγραμμα που αναλύει τον αριθμό των Pods που έχουν αναπτυχθεί δείχνει τον τρόπο με τον οποίο κάθε πράκτορας προσαρμόζει τη συμπεριφορά κλιμάκωσής του σε απόκριση στην αλλαγή του φόρτου εργασίας. Είναι εύκολα παρατηρησιμο ότι όλοι οι πράκτορες σε ολόκληρη την πρώτη περίοδο των 50 RPS έχουν αναπτύξει λιγότερα Pods από το KHPA. Αυτό είναι ένα σημαντικό πλεονέκτημα, καθώς λιγότερα Pods μεταφράζονται άμεσα σε χαμηλότερο κόστος από την πλευρά του διακομιστή. Σε μεταγενέστερο στάδιο, ο πράκτορας A2C υπερπρομηθεύει πόρους σε απόκριση στην αλλαγή του φόρτου εργασίας, επιδεικνύοντας προσαρμοστική συμπεριφορά κλιμάκωσης αλλά μια μικρή σπατάλη πόρων καθώς ξεπερνά τη γραμμή βάσης του KHPA. Είναι σημαντικό να σημειωθεί ότι στα τελευτα-

ία 12 επεισόδια του σταδίου αξιολόγησης, ο πράκτορας προσαρμόζεται στο περιβάλλον και μειώνει τον αριθμό των Pods που έχουν αναπτυχθεί κάτω από τη γραμμή βάσης του KHPA, υποδεικνύοντας βραδύτερη απόδοση προσαρμογής. Ταυτόχρονα, ο πράκτορας TRPO δείχνει καλύτερη ικανότητα κατανομής στο πρώτο στάδιο, αλλά σε μεταγενέστερα στάδια ευθυγραμμίζει τις αποφάσεις του με τη γραμμή βάσης του KHPA. Τέλος, και οι δύο πράκτορες PPO επιδεικνύουν εξαιρετικές δυνατότητες κατανομής πόρων, ευθυγραμμίζοντας αποτελεσματικά τις αποφάσεις παροχής τους με τη ζήτηση φόρτου εργασίας. Είναι σημαντικό να σημειωθεί εδώ ότι ο πιο εκπαιδευμένος πράκτορας PPO επιτυγχάνει απόδοση 22.55% λιγότερων Pods που έχουν αναπτυχθεί στο σύστημα, με αντάλλαγμα 9.16% υψηλότερη μέση καθυστέρηση. Όλες οι μετρήσεις και τα αποτελέσματα από το αξιολογημένο στάδιο συνοψίζονται στον παρακάτω πίνακα:

Table 5. *Μέση διαφορά μετρικών μεταξύ Πρακτόρων και KHPA (Επεισόδια 0-80).*

Metric	**PPO_2600**	**PPO_1300**	**A2C**	**TRPO**
Pod Usage				
Relative (%)	22.55%	10.30%	-0.74%	2.57%
Absolute	10.77	5.25	-0.05	2.03
p50 Latency				
Relative (%)	-9.16%	-6.09%	4.89%	-0.72%
Absolute (ms)	-1.369	-0.931	0.807	-0.120
p95 Latency				
Relative (%)	-14.64%	-6.05%	4.46%	-1.13%
Absolute (ms)	-2.770	-1.131	0.983	-0.251
p99 Latency				
Relative (%)	-12.29%	-8.17%	6.78%	-4.69%
Absolute (ms)	-2.516	-2.350	1.757	-1.182

Το τελευταίο διάγραμμα της Συνάρτησης Αθροιστικής Κατανομής (CDF) τόσο των επιθυμητών αντιγράφων όσο και των πραγματικών αναπτυγμένων Pod που φαίνεται παρακάτω υποδεικνύει μια ακόμη μετρική βέλτιστης κατανομής πόρων. Αυτή η απόδοση δίνει μια ισχυρή ένδειξη ότι με περισσότερη εκπαίδευση ο πράκτορας όχι μόνο έχει την ικανότητα να διαθέσει λιγότερα Pod, αλλά και προς τη σωστή κατεύθυνση όσον αφορά τη χρήση της CPU και της μνήμης. Οι υπόλοιποι πράκτορες, ο PPO με εκπαίδευση 1300 επεισοδίων, ο A2C και ο TRPO επιτυγχάνουν μια όχι και τόσο ικανοποιητική συμπεριφορά, εκτός του PPO_1300 καθώς φαίνεται να κατευθύνεται προς τη σωστή κατεύθυνση σύγκλισης. Η διαφορά στις κατανομές δείχνει πόσο κοντά βρίσκονται οι πράκτορες στην επίτευξη βέλτιστης απόδοσης σε ολόκληρη τη φάση αξιολόγησης.



Σχήμα 0.22. *Αθροιστική Συνάρτηση Κατανομής (CDF) των Pods σε ολόκληρη τη φάση αξιολόγησης.*

Όπως φαίνεται στο παραπάνω διάγραμμα, οι πράκτορες που έχουν εκπαιδευτεί για 1300 επεισόδια δεν έχουν την ικανότητα να προσαρμόσουν την ικανότητα κατανομής βέλτιστα τους ώστε να ταιριάζει με την κατανομή των επιθυμητών αντιγράφων. Αντίθετα, ο πράκτορας PPO που έχει εκπαιδευτεί για 2300 επεισόδια έχει εξαιρετική ικανότητα να κατανέμει πόρους σε ευθύ αναλογία με την επιθυμητή κατανομή αντιγράφων, γεγονός που αναδεικνύει τη σημασία της εκπαίδευσης σε τέτοια πολύπλοκα συστήματα.

0.4 Συμπεράσματα και μελλοντικές επεκτάσεις

Το πιο σημαντικό συμπέρασμα που προέκυψε από αυτή τη διπλωματική εργασία είναι ότι η δημιουργία ενός αποτελεσματικού και ακριβούς πράκτορα ενισχυτικής μάθησης που είναι σε θέση να κατανέμει σωστά τους πόρους δυναμικά σε ένα cluster Kubernetes είναι ένα εγγενώς πολύπλοκο και απαιτητικό έργο. Η εξαιρετικά δυναμική φύση των περιβαλλόντων που βασίζονται σε μικροϋπηρεσίες, με πολύπλοκες και εξαιρετικά δυναμικές αλληλεξαρτήσεις, εισάγει υψηλή μεταβλητότητα που πρέπει να ληφθεί υπόψη από τον πράκτορα στην εκπαίδευση και τις δοκιμές.Επίσης, οι online αλγόριθμοι ενισχυτικής μάθησης αποδεικνύονται ακριβοί, όσον αφορά την εκπαίδευση, σε συστήματα όπως το Κυβερνετες, με τρόπο που χρειάζονται μέρες για να εκπαιδευτεί ένας πράκτορας μόνο για λίγα επεισόδια, γεγονός που υποδηλώνει ότι αυτός ο τύπος εκπαίδευσης ενδέχεται να μην είναι ο καλύτερος. Αυτή η διπλωματική εργασία αξιοποίησε την κρίσιμη διαδρομή που εξάγεται από τα ίχνη που ανακτώνται μέσω του Jaeger και εκπαίδευσε έναν ταξινομητή SVM χρησιμοποιώντας αυτές τις πληροφορίες, προκειμένου να ταξινομήσει τις ανιχνευμένες υπηρεσίες ως υπαίτιες υπηρεσίες. Σε επέκταση, η δυαδική έξοδος μετασχηματίστηκε σε πιθανολογική έξοδο χρησιμοποιώντας το calibration του ταξινομητή, επιτρέποντας τη λήψη ασφαλέστερων αποφάσεων

όπου οι υπηρεσίες κατατάσσονται με βάση την κρισιμότητά τους και όχι με βάση τη δυαδική τους ταξινόμηση ως υπαίτιες ή όχι. Στο επόμενο στάδιο, οι εμπλουτισμένες πληροφορίες που είχαν προηγουμένως εξαχθεί, προστέθηκαν στη συνέχεια στην είσοδο του πράκτορα RL προσαρμόζοντας τους πόρους με βάση όχι μόνο την επαναληπτική μάθηση, αλλά και τη σημασία κάθε υπηρεσίας. Τα ολοκληρωμένα αποτελέσματα δείχνουν ότι το πρόβλημα της κατανομής πόρων σε τόσο πολύπλοκα, κατανεμημένα συστήματα είναι πράγματι ένα πραγματικά πολύπλοκο έργο. Η δημιουργία ενός πράκτορα τεχνητής νοημοσύνης που είναι ικανός να λαμβάνει αποφάσεις με αντίκτυπο απαιτεί προσεκτική ενσωμάτωση πολλαπλών στοιχείων, που κυμαίνονται από τα εργαλεία παρατηρησιμότητας και την ικανότητά τους να καταγράφουν τις σωστές μετρήσεις στον κατάλληλο χρόνο, αλλά και τα μοντέλα μηχανικής μάθησης και ενισχυτικής μάθησης που πρέπει να εκπαιδεύονται με ακρίβεια με αρκετές εκτελέσεις επεισοδίων, με ελάχιστη μεροληψία (bias) και τις σωστές αποφάσεις δράσης που θα επηρεάσουν αργότερα το προαναφερθέν σύστημα.

Παρά τις προκλήσεις αυτές, η προτεινόμενη διαδικασία έχει δείξει πολλά υποσχόμενα αποτελέσματα που ξεπερνούν το κλασσικό εργαλείο του Kubernetes το KHPA. Ο πράκτορας με τον μεγαλύτερο αριθμό εκπαιδευμένων επεισοδίων (PPO_2600) έχει επιτύχει σχετική διαφορά 22,55% λιγότερης χρήσης Pod σε αντάλλαγμα για 9,16% περισσότερη μέση καθυστέρηση ή 12,29% περισσότερο 99^{o} εκατοστημόριο καθυστέρησης στο Kubernetes cluster. Αυτό μεταφράζεται σε 10,77 λιγότερα Pods σε αντάλλαγμα για 1,369 (ms) περισσότερη μέση καθυστέρηση ή 2,516 (ms) περισσότερο 99^{o} εκατοστημόριο καθυστέρησης σε όλο το στάδιο αξιολόγησης που αποτελείται από 81 επεισόδια με δυναμικό φόρτο εργασίας που κυμαίνεται από 50 έως 100 RPS. Ακόμη και οι υπόλοιποι πράκτορες που έχουν εκπαιδευτεί για λιγότερα επεισόδια πέτυχαν εξαιρετικά αποτελέσματα σε ολόκληρο το στάδιο αξιολόγησης, όπως φαίνεται στον πίνακα 5.

Στην συνέχεια παρουσιάζονται κάποιες μελλοντικές επεκτάσεις αυτής της διπλωματικής εργασίας, οι οποίες θα επεκτείνουν και θα βελτιώσουν τα αποτελέσματα που παρουσιάστηκαν. Αρχικά, είναι πολύ σημαντικό να διασφαλιστεί ότι το σύστημα είναι κατασκευασμένο με ανθεκτικότητα και με τον σωστό τρόπο, ώστε αργότερα να δημιουργηθεί έναν πράκτορας που θα αλληλεπιδρά ορθά με αυτό. Για παράδειγμα, η δημιουργία μιας ξεχωριστής βάσης δεδομένων, όπως η Cassandra ή η ElasticSearch, για την αποθήκευση και διατήρηση των δεδομένων για τον Jaeger και όχι στη μνήμη, όπως υιοθετήθηκε σε αυτή τη διπλωματική εργασία, θα αποφύγει την σημαντική επιβάρυνση που προέκυψε κατά την προσπάθεια ανάκτησης των προαναφερθέντων δεδομένων (μέσω αιτημάτων http) προκειμένου να τροφοδοτηθεί ο αλγόριθμος CRISP για να εξαχθεί αργότερα η κρίσιμη διαδρομή από αυτά τα ίχνη. Επίσης, με τη ρύθμιση ενός Kafka pipeline μαζί με τη βάση δεδομένων, εξασφαλίζεται η βέλτιστη προεπεξεργασία των δεδομένων, με σκοπό τη βέλτιση εκπαίδευση των μοντέλων μηχανικής μάθησης. Ως συνέπεια, ο χρόνος εκπαίδευσης των πρακτόρων αναμένεται να μειωθεί με κρίσιμο τρόπο, επιτρέποντας στους προγραμματιστές όχι μόνο να εκπαιδεύσουν τους πράκτορες για περισσότερα επεισόδια, αλλά και να δοκιμάσουν διαφορετικές προσεγγίσεις και καινοτομίες.

Ταυτόχρονα, μια σημαντική βελτίωση είναι στον αλγόριθμο ανίχνευσης ακραίων τιμών ή όπως περιγράφτηκε σε προηγούμενα κεφάλαια, στον εντοπισμό του κρίσιμου στοιχείου. Όπως φαίνεται στη βιβλιογραφία, μέθοδοι όπως αυτοί που βασίζονται στην πυκνότητα έχουν

τη δυνατότητα να χρησιμοποιηθούν για την ανίχνευση ακραίων τιμών, για παράδειγμα η Χωρική Ομαδοποίηση Εφαρμογών με Θόρυβο (DBSCAN) που βασίζεται στην πυκνότητα είναι ένας εξαιρετικός αλγόριθμος που βρίσκει βασικά δείγματα υψηλής πυκνότητας και επεκτείνει συστάδες (clusters) από αυτές, ουσιαστικά ανιχνεύοντας σημεία εκτός αυτών των συστάδων και επισημαίνοντας τα ως ακραίες τιμές.

Μια ακόμη βελτίωση, η οποία θα βελτιώσει τη συνολική απόδοση ολόκληρου του pipeline, βασίζεται αποκλειστικά στον πράκτορα RL και την εκπαίδευσή του. Όπως περιγράφεται και φαίνεται σε αυτή τη διπλωματική εργασία, η εκπαίδευση του πράκτορα RL σε τέτοια πολύπλοκα συστήματα είναι αναμφισβήτητα η πιο κρίσιμη πτυχή για να ταιριάξει και τελικά να ξεπεράσει τις βασικές στρατηγικές του Kubernetes ή παρόμοιων συστημάτων, αλλά αποτελεί επίσης μια σημαντική πρόκληση, όπως φαίνεται στο κεφάλαιο αξιολόγησης, που σημαίνει ότι χρειάζονται μέρες για να εκπαιδευτεί ένας πράκτορας, για έναν μικρό αριθμό επεισοδίων. Μια ιδέα για την αύξηση της ταχύτητας της εκπαίδευσης είναι η χρήση μοντέλων Γενετικής Τεχνητής Νοημοσύνης (Generative Artificial Intelligence) για την προσομοίωση και την αναδημιουργία των κατανομών του χώρου κατάστασης-δράσης, δημιουργώντας τεχνητά νέα δεδομένα εκπαίδευσης που είναι πολύ κοντά στην πραγματικότητα και με αυτόν τον τρόπο η εκπαίδευση του πράκτορα RL θα βελτιωθεί σημαντικά.

Ακόμη, μια σημαντική βελτίωση που ενδέχεται να βελτιώσει τη συνολική απόδοση είναι η χρήση μιας δομής ενισχυτικής μάθησης πολλαπλών πρακτόρων, αναπτύσσοντας πολλούς πράκτορες RL που μοιράζονται την κατάσταση παρατήρησης ολόκληρου του συμπλέγματος Kubernetes. Αντί να βασίζεται σε έναν κεντρικό πράκτορα για τη διαχείριση όλων των διαφορετικών πόρων, μια εγκατάσταση πολλαπλών πρακτόρων κατανέμει το φόρτο εργασίας ή την ευθύνη των ενεργειών σε πολλούς πράκτορες με τη δυνατότητα να πραγματοποιούν διαφορετικές ενέργειες σε διαφορετικές υπηρεσίες, ενισχύοντας την τοπική λήψη αποφάσεων, με τις παρατηρήσεις του cluster να μοιράζονται μεταξύ των πρακτόρων από ενέργειες που πραγματοποιούνται από τους διαφορετικούς πράκτορες ξεχωριστά ή ακόμη και συνεργαζόμενοι.

Τέλος, μια σημαντική βελτίωση που θα βελτιώσει τη συνολική απόδοση ολόκληρου του pipeline είναι η χρήση ενός νευρωνικού δικτύου γραφημάτων (GNN) και περισσότερων τεχνικών που βασίζονται σε γραφήματα, σε συνδυασμό με τον πράκτορα ενισχυτικής μάθησης. Σε επέκταση αυτής της εργασίας, η κρίσιμη διαδρομή ή το κρίσιμο DAG που εξάγεται από τα ίχνη Jaeger, ακόμα και ολόκληρο το γράφημα που αποτελείται από τα ίχνη Jaeger έχει τη δυνατότητα να τροφοδοτηθεί σε αυτό το μοντέλο GNN. Με αυτόν τον τρόπο, η κατανόηση του περιβάλλοντος από τους πράκτορες θα ενισχυθεί σημαντικά, βελτιώνοντας τις αποφάσεις κατανομής πόρων που λαμβάνονται από αυτούς τους πράκτορες.

Chapter 1

Introduction-Problem Statement

1.1 Motivation and Problem Statement

In recent years, cloud computing has transformed the way applications are developed and deployed. Modern large-scale datacenters host an increasing number of popular cloud-based services that impact nearly every aspect of human activity. In order to satisfy the many complex objectives that different users have among the different services of their providers, a common way was to build a large, often complex architecture, called monolithic, that integrates every function, or in general block of code, into a single, unified codebase and deployed as one executable.

Although this technique seems fine, in many ways it induces very high complexity as the number of different services gets bigger and thus it can be frustrating and many times impossible to deal with. In more recent years, developers have shifted to more modular, decentralized and scalable approaches. Among the many architectural paradigms emerging from this transformation, microservices have gained widespread popularity due to their modularity, scalability, and resilience [14].

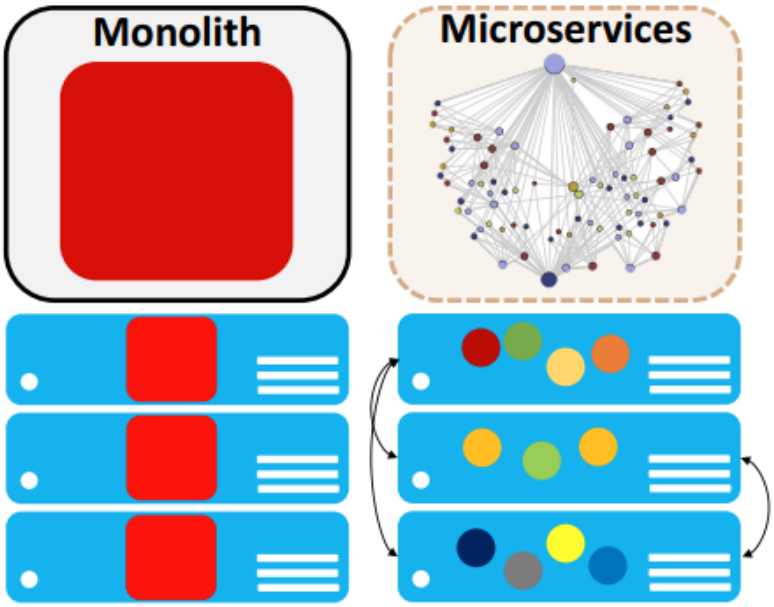


Figure 1.1. *Monolith vs Microservices architecture, Source: [8].*

However, as microservices become more complex and grow in size, managing their given resources dynamically becomes increasingly complex. Resource allocation is a cornerstone of operational efficiency in many fields, ranging from telecommunications and supply chain management to energy systems ,healthcare and finally in large cloud-based systems. The challenge lies in determining how to allocate limited resources optimally to achieve specific objectives. Traditional methods, such as linear programming or rule-based heuristics, while effective in static and deterministic settings, can struggle to adapt to the complexity and variability of modern systems and the high demand in user requests.

Resource allocation in microservices presents a unique challenge due to the architecture's dynamic and distributed nature. Microservices, which consist of loosely coupled services, each with distinct functionalities, experience fluctuating workloads and varying resource demands based on user interactions and inter-service communications. This variability complicates the task of distributing very important computational resources such as CPU and memory. Traditional static or heuristic allocation methods such as static thresholds as implemented by solutions like the Horizontal Pod Autoscaler (HPA) [41] and Vertical Pod Autoscaler (VPA) [42], struggle to adapt to these changing conditions, leading to potential inefficiencies, such as over-provisioning (waste) or under-utilization of resources leading to bottlenecks.

Reinforcement learning (RL) offers a promising approach to this problem by allowing for dynamic, data-driven resource allocation decisions. In this context, RL models the microservices environment as a series of states, each reflecting current resource utilization, workload, and performance metrics like latency and throughput. Actions in this framework involve decisions about adjusting resource allocations or scaling services up or down. The RL agent's objective is to learn a policy that optimizes a reward function, which typically combines goals such as minimizing response times, maximizing resource utilization, and reducing operational costs.

Also, supervised or unsupervised machine learning models can help the aforementioned RL agent to generalize more quickly and also gain impactful insight information about the residing microservices it has to deal with. The application of RL in microservices resource allocation provides several advantages. It enables adaptive responses to workload changes, optimizing resource usage in real-time, and considers the complex dependencies between services, leading to more holistic and effective allocation strategies. Moreover, RL automates the decision-making process, reducing the need for constant manual intervention and tuning. However, this approach brings up many rough challenges. The implementation and training of the RL agents can introduce computational overhead, and during the learning phase, the RL agent may make suboptimal decisions as it explores various strategies, making the whole process really slow.

Despite these challenges, RL has shown great potential in large-scale cloud-based platforms, handling them the ability to adapt and manage resource allocation dynamically, ensuring efficient operation even under varying demand conditions. By leveraging reinforcement learning and machine learning in general, organizations can achieve a more responsive and cost-effective management of their microservices environments, paving the way for improved performance and scalability.

1.2 Related Work

This section explores the work done at the literature, pointing out great frameworks that explore the problem of resource allocation with various techniques. The explored work is divided into threshold-based controllers, statistical controllers-models and finally ML-based controllers-models. This literature review is essential in the design of this thesis's pipeline with the objective of optimal resource allocation in Kubernetes clusters.

First, Threshold-based techniques constitute one of the most widely deployed SLO protection mechanisms in cloud systems. Popular platforms (e.g. Amazon, K8s) rely on threshold-based scaling policies on metrics such as CPU and memory pressure and queue depth. When a metric crosses the predefined threshold, specific actions are triggered, altering the formation of the cloud by adjusting the resource provisioning (e.g. increase the CPU or memory request). This approach is relatively easy to implement as it does not require any type of training or data acquisition. However, threshold rules require manual tuning, react poorly to non-stationary or in changing workloads as seen in many papers([43]).

At the same time, another technique that is used in the literature is the statistical-based one. This technique uses statistical models or analytical models to anticpate QoS fluctuations and keep the predefined SLAs/SLOs intact. A great example of this approach is Ursa [44] in which the authors propose a performance model for mapping microservice SLAs to resources with decomposition of end-to-end latency, mapping the per-service latency to resources and finally use a resource optimization model to calculate an efficient way to allocate resources for each microservice using solvable mixed-integer programming (MIP) models.

On the other hand, ML-based techniques aim to build a model that is able to predict the right resource allocation to the cluster under varying or specific workloads based on metrics such as CPU , memory or even workload traffic or network traffic. This approach can be really robust in dynamic environments due to the adaptation of the model via training.

A great example of this approach is Sinan [45], in which the authors created a model-based approach with Convolutional Neural Networks (CNNs) and boosted trees to predict the end-to-end latency and give the probability of a QoS violation. This approach leverages ML and the correlation with time, to identify the impact of dependencies on end-to-end performance and make efficient resource allocation decisions.

Also, in FAST23 [46] the authors propose an intelligent ML-based scheduler which learns the correlation between architectural hints (e.g. IPC, cache misses, memory footprint, etc.), scheduling solutions and the QoS demands. They adopt a multi-model collaborative learning approach, where multiple ML models are employed in order to work collaboratively to anticipate and predict QoS variations-fluctuations. Trained on runtime data, this ensemble sustains SLOs at higher and dynamic workloads.

Another great example is FIRM [31] which is a fine-grained, intelligent cluster management framework designed to optimize the performance and efficiency of microservice-based cloud applications by leveraging hardware heterogeneity and real-time performance

insights. Recognizing that microservices exhibit diverse resource demands and interdependencies, FIRM introduces the idea of acquiring critical paths from the dependency graph of all microservices. Based on that, the focus shifts to a smaller group of microservices, giving us the ability to not waste time and resources in order to fix problems that are not the direct cause of the loss of target. Lastly, it makes great use of machine learning, and reinforcement learning in order to automate the resource allocation efficiently and rapidly.

Lastly, a great approach to solve the problem of resource allocation is Seer [47] which combines ML-based prediction with queueing theory-based signals. The authors of Seer trained a pipeline that consists of Convolutional Neural Networks and Long Short-Term Memory (LSTM) networks in order to reduce the dimensionality of the original dataset and at the same time capture load patterns effectively, guided by queueing theory intuition and metrics reducing latency with a short window to act.

Table 1.1. *Related Work.*

Framework	Method	Signals	Key idea	Dependencies
K8s HPA [41]	Threshold	CPU, mem, queue depth	Fixed rules trigger scaling	Metric only
Ursa [44]	Statistical	Arrivals, service rates, per-svc targets	SLO decomposition with MIP	Per-service analytic decomposition
Sinan [45]	ML	CPU/mem, traffic, time/dep feats	CNN + boosted trees predict tail	Explicit inter-service graph
FAST'23 (OSML) [46]	ML	Architectural hints & QoS demand	Multi-model collab. scheduler	Learned interference across co-located svcs
FIRM [31]	ML+RL	Telemetry data	Contention localization + RL reprovision	Critical-path extracted manually
Seer [47]	ML+Queue	Traces, per-svc queue depth	DL forecasting for early action	Trace-derived initiators / causal hints

1.3 Proposed Solution and Outline of Thesis

The proposed solution presented in this thesis is inspired by FIRM's [31] approach, which has shown promising results in resource allocation. However, to better suit the specific requirements of this study, we introduce several modifications that adapt and extend the original model. These adaptations are intended to address the acquirement of critical path more effectively, training of the SVM model more effectively with a significant modification of the output transforming it to probabilistic and finally feeding the aforementioned results into online Reinforcement Learning Agent of recent RL modules in order to improve the management of the total resources of Kubernetes and cloud cluster as a whole.

This thesis is composed out of three main parts, with Part I named Theoretical background providing the main theoretical information about the technologies being utilized in this thesis, giving the readers the ability to have a clear idea of the referenced concepts. More specifically, this part is composed from the following chapters:

- **Chapter 2**: Presents the essential theoretical information about Kubernetes including its architecture, core components and finally the integration with monitoring

tools such as Jaeger and Prometheus.

- **Chapter 3**: Provides the essential theoretical foundations of Machine learning and its components. More specifically, starting with supervised and unsupervised learning, covering the main algorithm used in this thesis, while later introducing neural networks with its underlying structures and their role as powerful tools for approximations. Finally, an extensive analysis about Reinforcement learning covering its underlying mathematical structure and in later stage introducing the deep neural network addition in order to improve the value and policy based methods.

Continuing with Part II named Implementation, focus shifts from theoretical applications to the practical implementation of the previous technologies. More specifically, this part details the integration of all Kubernetes components with the Machine learning algorithms previously mentioned, divided into the two following chapters:

- **Chapter 4**: In this chapter, the entire architecture of the proposed pipeline is examined in detail, outlining the details of its structure with the parameters used to test the accuracy and effectiveness of the pipeline.
- **Chapter 5**: The results of both the training and testing of the aforementioned pipeline will be presented.

Finally, Part III contains one chapter (Chapter 6) that summarizes the conclusions of this thesis and provides suggestions for future work in order to increase the accuracy and efficiency of the proposed pipeline.

Part I

Theoretical Preliminaries

Chapter 2

Kubernetes and related technologies

In this Part, we provide comprehensive information on the technologies referenced throughout this thesis. The focus is to understand fundamental concepts and technologies about the underlying system that this thesis is dedicated to. Starting, in this chapter the basic concepts of Containers, Microservices and Docker platform are going to be analyzed. Also, the basic ideas and components of Kubernetes are going to be extensively analyzed, giving great details about the underlying algorithms used in Kubernetes for resource allocation and self-healing. Furthermore, in this chapter the fundamentals of Monitoring in microservices system will be introduced, emphasizing in tools like Jaeger which is a mandatory tracking tool in such systems. Also, Prometheus, with its added services, is going to be analyzed in order to close the chapter having all the basic concepts of microservices and its core ideas.

2.1 Containers and Microservices

Containers are a standard unit and form of operating system-level virtualization that allow applications to run in isolated environments, integrating the application code along with all its dependencies in order to run properly without the need of a hypervisor. This integration ensures that the software behaves consistently across different computing environments, from development and testing to production. Unlike virtual machines, which require a separate operating system for each instance and essentially a hypervisor, containers share the kernel of the host machine-system, making them significantly more lightweight and efficient in terms of resource consumption and startup time (Fig.2.1).

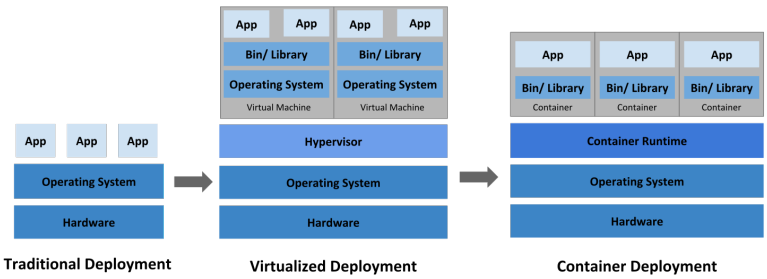


Figure 2.1. *Traditional approach vs VM vs Container, Source: [1].*

At the same time, containers are fundamental in the microservice architecture [48] that developers are adopting recently. More specifically, applications like front-end or back-end, are separated into containers that each play a different and individual role, while each of them are loosely coupled, microservices can easily communicate. Each microservice communicates with each other via a specific protocol, for example gRPC [29] or HTTP [49].

2.2 Docker

Docker [50] is one of the most widely used containerization platforms that has popularized and standardized the use of containers in the software industry. With the use of Docker, user can host a large number of containers to his host machine without the need of setting up different VMs as described above. Docker simplifies application deployment, reduces conflicts between software versions, and optimizes resource utilization by sharing the host OS kernel. However, while Docker is good at managing individual containers or small groups of services, it lacks native support for handling complex, distributed systems at scale, such as orchestrating container lifecycles, managing service discovery, and ensuring high availability.

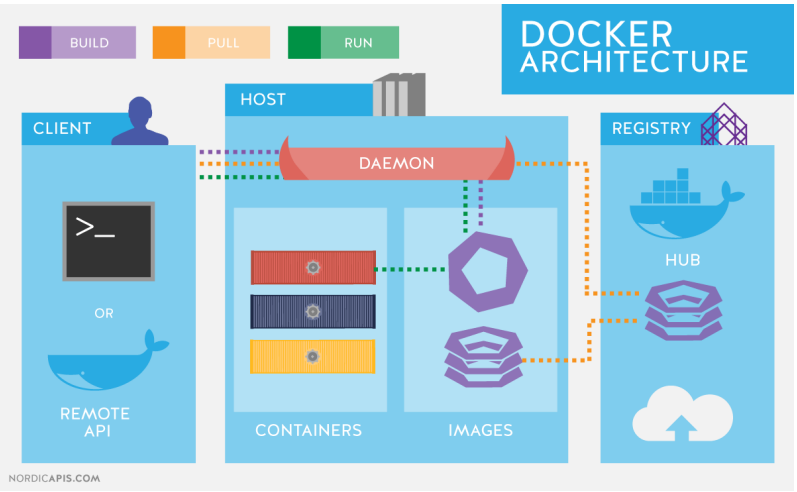


Figure 2.2. *Docker architecture, Source: [9].*

2.3 Kubernetes

Kubernetes [1] (K8s) is an open-source container orchestration system for automating deployment, scaling, and management of containerized applications. First developed by Google in 2014 and later on 2015 was launched open source, Kubernetes has become the industry standard as it offers numerous advantages on container-based cloud environments, starting with a few robust features like:

- **Self-healing**: Automatically replaces or restarts failed containers.
- **Rolling updates**: Gradually updates applications with minimal disruption.

- **Automated failover**: Redirects traffic or workloads to healthy instances to ensure high availability.
- **Automatic bin packing**: You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto nodes to make the best use of resources.
- **Horizontal and Vertical scaling**: Automatically adjusts the number of Pods in a deployment (Horizontal) or the resource limits of individual Pods (Vertical) based on observed metrics such as CPU usage, RAM usage, network traffic, workload intensity, and many more, ensuring optimal performance and efficient use of resources.

2.3.1 Kubernetes cluster components

In this subsection, the essential components are provided in order to create a Kubernetes cluster (system). Starting of with some basic terms for Kubernetes:

- **Nodes**: A node may be a virtual or physical machine, depending on the cloud cluster. Each node is managed by the control plane or the master node and contains the services necessary to run Pods.
- **Pods**: Pods are the smallest deployable units of computing that you can create and manage in Kubernetes. A Pod is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled, and run in a shared context. A Pod models an application-specific "logical host" by containing one or more application containers. These containers are relatively tightly coupled, with shared storage and network resources, and a specification on how to run them. [1].
- **Services**: In a cluster, Pods can be terminated or restarted unexpectedly, making them incredibly unreliable. Therefore, due to the highly dynamic nature of the Pods, Services are created to be a more abstracted layer than Pods, making them essential for the exposure of the Pods over the network. In that way, the developer does not have to rely on the dynamic IP each Pod has, and therefore each Service object defines a logical set of endpoints (usually these endpoints are Pods) along with a policy about how to make those pods accessible. In short, Service manages access to the Pods.
- **Deployment**: Last but not least, Deployment is essential for managing the lifecycle of Pods, handling tasks like scaling, rolling updates, and self-healing.

A Kubernetes cluster consists of a control plane, often called master-node, and a set of worker-nodes referred as nodes. Worker nodes host Pods that are the components of the application workload and the same time Control plane is responsible for the appropriate operation of the worker nodes and the Pods that are being scheduled to them. In production environments, a good practice is that the Control Plane often has many copies across

the cluster, offering high availability and fault tolerance to the cloud. Core components of the cluster, starting of with the Control-Plane components:

- **kube-apiserver**: The core component server that exposes the Kubernetes HTTP API.
- **etcd**: Consistent and highly-available key value store database that stores all API server data.
- **kube-scheduler**: Looks for Pods not yet bound to a node, and assigns each Pod to a suitable node. Via the Round-Robin algorithm scheduler selects one node and tries to evaluate if it is suitable for Pod scheduling. Factors taken into account for scheduling decisions include: individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines.

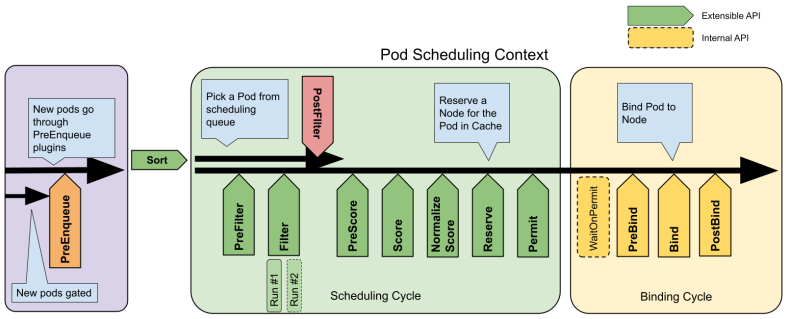


Figure 2.3. *Kubernetes Scheduling framework, Source: [1].*

- **kube-controller-manager**: Runs controllers to implement Kubernetes API behavior.
- **cloud-controller-manager (optional)**: Integrates with underlying cloud provider structure. For example providers can be AWS [15], Microsoft Azure [16] or Google cloud [17].

Continuing with the Node components that run on every worker node, maintaining running pods and providing the Kubernetes runtime environment :

- **kubelet**: Ensures that Pods are running properly without any error, including their containers.
- **kube-proxy (optional)**: Maintains network rules on nodes to implement Services.
- **Container runtime**: Software responsible for running containers. The most common one is Docker (Docker-engine now), following by Containerd and CRI-O.

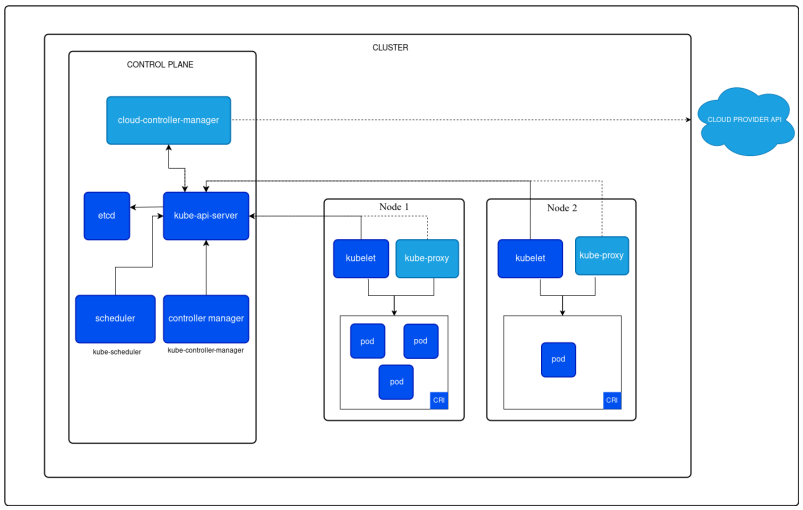


Figure 2.4. *Kubernetes cluster architecture, Source: [1].*

2.3.2 Kubernetes Scaling

In this subsection, the basic techniques of scaling in Kubernetes are briefly analyzed. Although Kubernetes system can indeed self-heal and restart broken Pods, that is not enough to ensure that the cluster is operating in optimized environment settings. That is why the concept of autoscaling resources is present and helps achieve the best conditions for the cluster. In Kubernetes, you can scale a workload depending on the current demand of resources. This allows your cluster to react to changes in resource demand more elastically and efficiently.

- **Horizontal scaling**: One of the most important approaches of autoscaling resources in Kubernetes is Horizontal scaling, which involves increasing or decreasing the number of Pod replicas to match workload demand and at the same time distribute the workload across the current Pod replicas. In this way, applications can handle traffic spikes or reduce resource usage during low-demand periods. While this method does increase the total resource consumption, it enables greater parallelism, and resilience against failures, making it especially effective for applications designed to operate as multiple independent instances such as stateless Services. The negative aspect of this approach is that it increases the total resource consumption and also, sometimes can be wrong to apply this kind of scale in some microservices due to their stateful nature. Stateful Services maintain data or session information that must remain consistent across requests, making it challenging to simply add replicas without careful coordination. For example, databases or caching layers require specialized mechanisms like replication, sharding to ensure data integrity and consistency when scaled horizontally and at the same time must satisfy the CAP theorem [51].

- **Vertical scaling**: In contrast, instead of increasing or decreasing the number of Pod replicas of the specific service, Kubernetes is able to give more resources to the Pod, with the most common ones being the CPU and memory, in order to meet the

current demands of the workload. This approach is suited best for stateful Services. Although vertical scaling can effectively improve performance for such workloads, it has inherent limitations, such as maximum hardware capacity and the risk of creating a single point of failure but also the risk of allocating less resources than needed which may cause bugs or even out of memory errors.

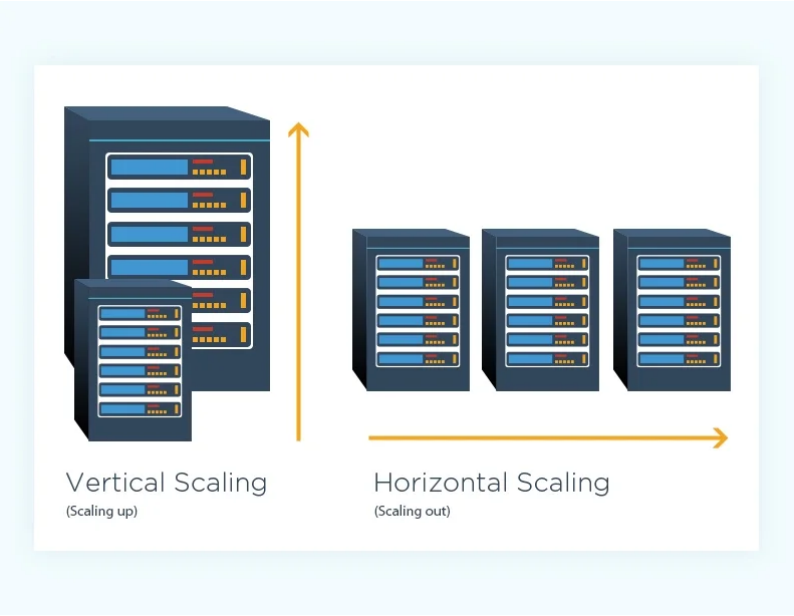


Figure 2.5. *Vertical and Horizontal scaling techniques, Source: [2].*

- **Event driven Autoscaling**: Event driven autoscaling is a CNCF [19] graduated project enabling you to scale your workloads based on the number of events to be processed, for example the amount of messages in a queue.
- **Node scaling**: In very rare situations, Node scaling is also possible in Kubernetes. If Pod scaling is not enough to distribute the workload effectively, adding more Nodes to the cluster can eventually also help to achieve this goal.

2.3.3 Horizontal Pod Autoscaler (HPA)

Although all of the aforementioned techniques have their distinct advantages, for this thesis, the chosen use and baseline of performance is the default HPA of Kubernetes. As mentioned before, the most common practice is that based on some kind of resource, for example CPU usage or RAM usage, the HPA instructs the workload resource (the Deployment, StatefulSet, or other similar resource) to scale up or down. Basically, HPA is a controller that tries to transform the current state into the desired one via the metrics-server which is an API that needs to be launched separately to acquire the desired metrics. The default algorithm that HPA uses in order to achieve the desired number of replicas is the following:

$$\text{desiredReplicas} = \text{ceil}\left[\text{currentReplicas} \times \frac{\text{currentMetric}}{\text{targetMetric}}\right]$$

Based on the above algorithm and a configurable tolerance (usually 10%) that applies on the base scale ratio, HPA makes the decision to scale the replicas. It is very important to note that this algorithm takes into account all Pods with Ready state and the ones with a deletion timestamp set (objects with a deletion timestamp are in the process of being shut down / removed) are ignored, and all failed Pods are discarded. Also, due to technical constraints and initialization noise of the Pods (really high at the initial state) , Pods with CPU target metric are initially labeled as "not yet ready" and then in a certain amount of time, that is also configurable, they transition to state Ready. This configuration flag is named horizontal-pod-autoscaler-initial-readiness-delay and its default value is 30 seconds. At the same time, another flag that tries to avoid any CPU value during the warmup stage (especially useful for example Java apps) is horizontal-pod-autoscaler-initial-readiness-delay with default value of 5 minutes.

2.4 Monitoring tools

2.4.1 Jaeger

Jaeger [3] is an open-source, end-to-end distributed tracing system originally developed by Uber to monitor and troubleshoot complex microservice architectures. It helps developers understand how requests flow through a system by collecting and visualizing trace data, which includes timing information for operations across multiple services. Jaeger enables the identification of performance bottlenecks, root cause analysis of errors, and optimization of service dependencies by providing detailed insights into request latency and service interactions. It integrates well with modern observability stacks and supports standards like OpenTracing [52] and OpenTelemetry [53] .

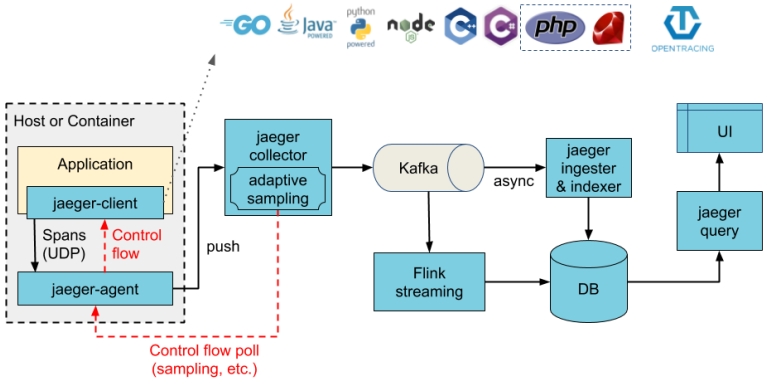


Figure 2.6. *Jaeger architecture, Source: [3].*

Figure 2.6 illustrates the core architecture of Jaeger. It consists of jaeger-client, jaeger-agent, jaeger-collector, jaeger-query and optionally jaeger-ingester. Each of the aforementioned components is essential in the Jaeger tracing system to track and troubleshoot complex microservices architectures in the following way:

- **jaeger-client**: Jaeger clients are language-specific implementations of the OpenTracing API. They can be used to instrument applications for distributed tracing either manually or with a variety of existing open source frameworks, such as Flask, Dropwizard, gRPC, and many more, that are already integrated with OpenTracing. The traces that are being fetched consist of spans for each operation in the underlying application. In that way, a directed acyclic causal graph is created and essentially models the causal relationship between each microservice, with each microservice being a node and an operation being the edge. This can be very useful for a developer not only to debug and understand the underlying system but also to use this kind of information for various machine learning cases.

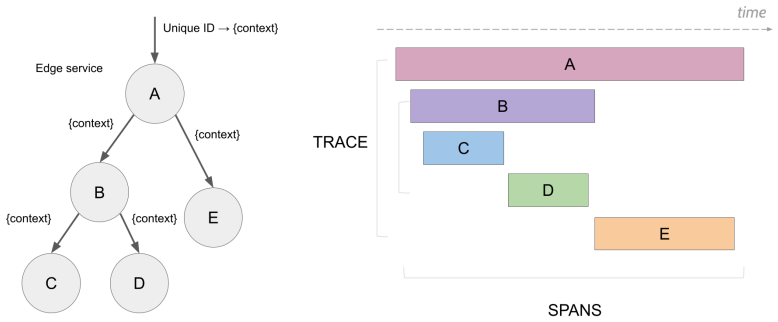


Figure 2.7. *Traces, Spans and the Causal graph in Jaeger, Source: [3].*

- **jaeger-agent**: Jaeger agent is a network daemon that essentially listens and waits for the client to send tracing data via the UDP protocol [54]. Once collected, the agent sends them to the collector for further analysis.

- **jaeger-collector**: Jaeger collector receives the traces as mentioned before and processes them through a specific pipeline, consisting of validation, indexing, transformation and then persists them into a Jaeger storage. Jaeger storage is a pluggable component and in the simple case it can be an in-memory infrastructure or, in the more complex case, can be one of the currently supported databases of Cassandra [55], ElasticSearch [56] or Kafka [57].

- **jaeger-query**: Jaeger query is essential to acquire the data from the database (in-memory or not) and display them to the hosted UI.

- **jaeger-ingester (optional)**: In order to use a complex backend to store the data from jaeger-agent and jaeger-collector, a jaeger-ingester must be present, this way data can be read from a Kafka topic and then stored to a Cassandra or ElasticSearch database.

2.4.2 Prometheus, Grafana and Alertmanager

Prometheus is an open-source systems monitoring and alerting toolkit originally built at SoundCloud [18] but since 2016 has joined CNCF [19] as the second hosted project, after Kubernetes [1]. Prometheus collects and stores its metrics as time series data, i.e.

metrics information is stored with the timestamp at which it was recorded, alongside optional key-value pairs called labels [4]. It helps developers to track important metric from their systems with very high accuracy and gain visibility of their behavior. Tracking metrics such as CPU, RAM, network traffic and many more, handles the opportunity for various advanced analyses and optimizations.

Figure 2.8 illustrates the core architecture of Prometheus. It consists of the main components, Prometheus server which scrapes and stores time series data, exposing the very powerful PromQL query language that enables users to filter, aggregate, and analyze metrics with great flexibility, Grafana which helps with the visualization of the scraped metrics, Alertmanager which can be set accordingly in order to alert the developer based on defined conditions. Lastly, Pushgateaway for supporting short-lived jobs to deliver metrics that also integrates well with Grafana creating live and dynamic dashboards.

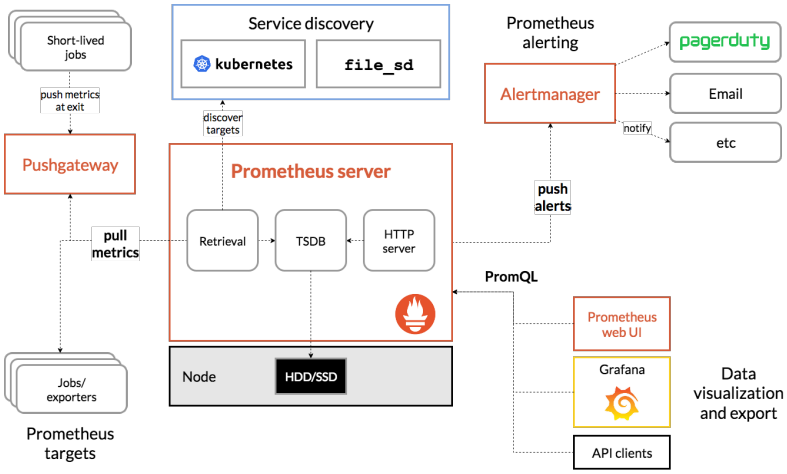


Figure 2.8. *Prometheus architecture, Source: [4].*

Chapter 3

Machine learning

In this chapter, we provide comprehensive information on the technologies referenced throughout this thesis. The focus is to understand fundamental concepts and technologies relevant to resource allocation with the use of reinforcement learning and machine learning in general, which are the core essence of this thesis. Starting, machine learning topics such as supervised learning, unsupervised learning and reinforcement learning are also going to be extensively analyzed. More specifically, deep reinforcement learning as the core feature will be presented, and at the same time, algorithms of that nature such as Proximal Policy Optimization (PPO), will be introduced. Furthermore, in the supervised and unsupervised learning section the core algorithm of SVM will be analyzed due to the criticality in this thesis.

Machine learning [20] is a field of artificial intelligence that enables systems to automatically learn patterns and make decisions or predictions based on data, without being explicitly programmed for specific tasks. It relies on statistical and computational techniques to extract insights from small or large datasets and at the same time mitigate the complexity of the task needed. In order for these systems to learn patterns, mathematical algorithms (often called model) over many loops, called epochs, are being applied to iteratively adjust the model's parameters. In this way, each model based on the task and the data they are iteratively applied tends to learn specific patterns and finally makes future predictions. The data aforementioned are randomly split into training and validation sets, with the training data usually being the 80 percent of the whole dataset and the validation data being the rest (assuming the simplest form of split). Training data are used to let the model learn specific patterns from input to output, usually by minimizing a loss function, the loss function being a mathematical expression that quantifies the difference between a model's predicted output and the actual ground truth. On the other hand, validation data are used to evaluate how well the model has learnt the general relationship between input and output or simply the function that expresses output in terms of input. Here, terms like overfitting [58] appear, referring to the situation where a model learns the training data too well, including its noise and minor fluctuations, rather than capturing the underlying general patterns and function that connects the output(s) to input(s). As a result, while the model performs exceptionally well on the training data, it tends to score poorly on validation or test data, indicating poor generalization. This discrepancy is often reflected in evaluation metrics such as accuracy, precision, recall, or

loss, where the model's performance on unseen data significantly drops. Overfitting is a common challenge in machine learning and can be mitigated through various techniques such as cross-validation, regularization, early stopping, and by using more training data or simpler models.

3.1 Supervised and Unsupervised learning

Supervised learning [59] is a prominent machine learning paradigm in which algorithms are trained on labeled datasets, enabling them to predict or classify outcomes by identifying patterns and relationships between input features and associated target variables. This approach relies on explicit guidance in the form of annotated examples, where the model iteratively adjusts its parameters to minimize the values between predicted and actual values, as called before a loss function.

Table 3.1. *Common loss functions in supervised learning [12]*

Loss Function	**Mathematical Expression**	**Use Case**
Mean Squared Error (MSE)	$\mathcal{L}_{\text{MSE}} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$	Regression
Mean Absolute Error (MAE)	$\mathcal{L}_{\text{MAE}} = \frac{1}{n}\sum_{i=1}^{n}\lvert y_i - \hat{y}_i\rvert$	Regression
Binary Cross-Entropy (BCE)	$\mathcal{L}_{\text{BCE}} = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right]$	Binary classification
Categorical Cross-Entropy (CCE)	$\mathcal{L}_{\text{CCE}} = -\sum_{i=1}^{n}\sum_{j=1}^{C} y_{ij}\log(\hat{y}_{ij})$	Multi-class classification
Hinge Loss	$\mathcal{L}_{\text{hinge}} = \sum_{i=1}^{n}\max(0, 1 - y_i \cdot \hat{y}_i)$	Support Vector Machines (SVMs)

Also, in order to avoid overfit as described before, many times a regularization term is added in the previous loss functions as shown in Table 3.2.

Table 3.2. *Common regularization (penalty) terms in supervised learning [13]*

Penalty Type	**Mathematical Expression**	**Purpose**
L1 Regularization (Lasso)	$\mathcal{R}_{\text{L1}} = \lambda\sum_{j=1}^{m}\lvert w_j\rvert$	Encourages sparsity by driving weights toward zero
L2 Regularization (Ridge)	$\mathcal{R}_{\text{L2}} = \lambda\sum_{j=1}^{m} w_j^2$	Penalizes large weights, helps reduce overfitting
Elastic Net	$\mathcal{R}_{\text{EN}} = \lambda_1\sum_{j=1}^{m}\lvert w_j\rvert + \lambda_2\sum_{j=1}^{m} w_j^2$	Combines L1 and L2 for balance between sparsity and smoothness

Common algorithms employed within supervised learning include linear regression, logistic regression, decision trees, support vector machines, and artificial neural networks, each with distinct advantages and limitations depending on data characteristics and application contexts.

On the other hand, unsupervised learning [60] is a critical paradigm in computer science and machine learning that deals with identifying patterns, structures, or intrinsic relationships within unlabeled data, without explicit guidance or predefined output labels. Unlike supervised learning, unsupervised algorithms such as clustering methods (e.g., k-means, hierarchical clustering), dimensionality reduction techniques (e.g., principal component analysis, autoencoders), and anomaly detection models rely solely on input features, seeking to uncover hidden patterns or groupings inherent to the dataset. This approach enables the exploration of data in scenarios where labeling is costly, impractical, or unavailable. In figure 3.1 the basic categories of machine learning are briefly shown with the appropriate algorithms.

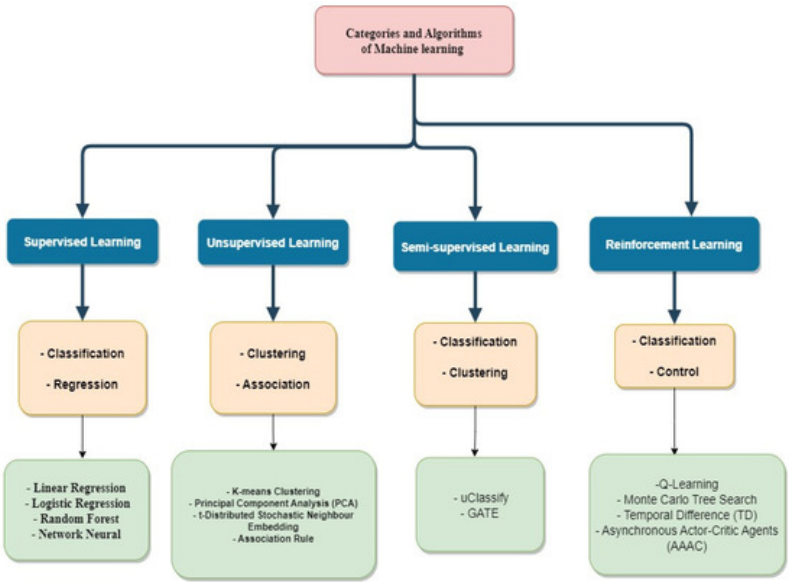


Figure 3.1. *Machine learning categories, Source: [10].*

3.1.1 Support Vector Machines (SVMs)

Support Vector Machines (SVMs) [21] are supervised learning algorithms primarily used for classification (and regression in the form of Support Vector Regression). The core idea is to find a hyperplane in a high-dimensional feature space that separates data points of different classes with the maximum possible margin (often called "street"). The data points closest to this hyperplane are called support vectors, as they define the boundary and thus the linearly separable problem is described as:

$$\min_{w,\,b} \quad \frac{1}{2}\|w\|^2 \tag{3.1}$$

$$y_i(w^\top x_i + b) \geq 1, \qquad i = 1, \ldots, N \tag{3.2}$$

where N is the total number of data points with $x_i \in \mathbb{R}^d$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, N$.

In the non-linearly separable case, two modifications are introduced to the original SVM formulation:

- **Slack variables** $\xi_i \geq 0$ to allow misclassification (soft margin).
- **Kernel mapping** $\phi(x)$ to transform data into a higher-dimensional feature space where linear separation is possible.

A mapping is defined, in order to solve the following optimization problem:

$$\phi : \mathbb{R}^d \to \mathcal{H} \tag{3.3}$$

$$\min_{w,b,\xi} \quad \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N} \xi_i \tag{3.4}$$

$$y_i(w^\top \phi(x_i) + b) \geq 1 - \xi_i, \quad i = 1, \ldots, N \tag{3.5}$$

$$\xi_i \geq 0, \quad i = 1, \ldots, N \tag{3.6}$$

where $C > 0$ controls the trade-off between margin width and misclassification. Instead of computing $\phi(x)$ explicitly, the kernel trick is used, replacing the dot product $\phi(x_i)^\top \phi(x_j)$ with a kernel function:

$$K(x_i, x_j) = \phi(x_i)^\top \phi(x_j) \tag{3.7}$$

Common choices of kernels include the polynomial kernel, radial basis function (RBF) kernel, and sigmoid kernel.

3.2 Neural Networks

While traditional machine learning algorithms shown above can be very effective and often lead to very promising results, when the dataset or the target objective requires complex solutions, more complex architectures are required in order to reach reliable and accurate performance. Inspired by the human brain (Figure 3.2) neural networks were developed, made from individual neurons and layers (often called hidden layers). Each neuron receives one or more input values, applies a weighted sum followed by a nonlinear activation function, and passes the result to the next layer [22].

$$a_i^{(l)} = g\left(\sum_{j=1}^{n} w_{ij}^{(l)} a_j^{(l-1)} + b_i^{(l)}\right) \tag{3.8}$$

where:

- $a_i^{(l)}$ is the activation of neuron i in layer l,
- $w_{ij}^{(l)}$ is the weight connecting neuron j in the previous layer $(l-1)$ to neuron i,
- $b_i^{(l)}$ is the bias term for neuron i,
- $g(\cdot)$ is a nonlinear activation function (e.g., ReLU, sigmoid, or tanh),
- n is the number of neurons in the previous layer.

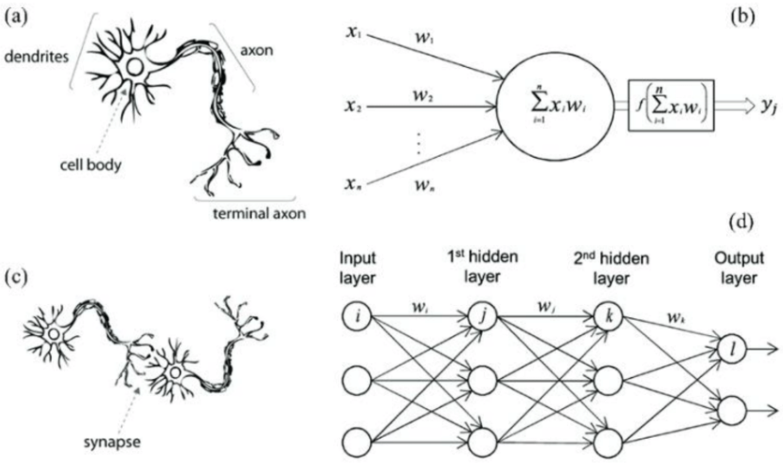


Figure 3.2. *A biological neuron in comparison to an artificial neural network. (a) Brain neuron, (b) Artificial neuron, (c) Neuron and biological synapse, (d) Artificial neural network. Source: [11].*

This layered architecture allows neural networks to learn complex, hierarchical patterns in data by progressively extracting features at different levels of abstraction. More specifically, the network previous shown is called feed-forward network (FFN) or multilayer perceptron (MLP) that consists only from neurons and layers. These architectures often include deeper networks with many hidden layers (deep learning networks). In order to evaluate these networks, similar techniques are employed as in simpler models. However, due to the increased complexity arising from the large number of neurons and layers, a specialized optimization method known as Backpropagation [22] is used. This technique efficiently computes gradients by propagating the error backward through the network, allowing for effective adjustment of weights during the training process. In many cases instead of just using simple neurons, alternative network architectures exist that combines inputs or even neurons, e.g. Convolutional neural networks (CNN) or Recurrent neural networks (RNN).

3.3 Reinforcement learning

In the previous sections, supervised and unsupervised learning approaches were presented, both of which rely heavily on static, pre-labeled datasets and essentially a supervisor to either manually or with some programmable way annotate each entry to the target variable of that dataset. That can be very costly and many times impossible to do in production level environments that the value of the target variable is unknown until the event happens or is constantly changing with time meaning it is a highly dynamic environment. In such settings, the outcome of an action often depends not only on the current state but also on the sequence of prior decisions, creating a complex dependency that traditional data-driven methods struggle to handle. To overcome these challenges, Reinforcement learning (RL) introduces a different approach to sequential decision-making kind of problems.

3.3.1 Markov Decision Processes (MDP) and Bellman equations

In the context of this category of problems, it is necessary to appropriately introduce the framework of a Markov Decision Process (MDP) [61] that is formal to decision-making problems in dynamic environments. Markov Models are stochastic models created to describe non-deterministic processes with the main advantage being that they are memoryless, meaning they only depend on the current state and not the previous ones. In that way, having enough information about the current state and a set of actions, the transition ,via the agent, to the next step is possible. An MDP is defined as:

$$\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, P, R, \gamma \rangle$$

where:

- A set of states $s \in S$, can be discrete or continuous.
- A set of actions $a \in A$ the agent can execute, can be discrete or continuous.
- A reward function
$$R : S \times A \to \mathbb{R}$$
which assigns a scalar reward to each state–action pair.
- A transition function
$$P : S \times A \times S \to [0, 1]$$
where $P(s'|s, a)$ denotes the probability (discrete case) or probability density (continuous case) of transitioning to the next state s' from state s after taking action a.
- A discount factor $\gamma \in [0, 1]$, which specifies how future rewards are weighted relative to immediate rewards.

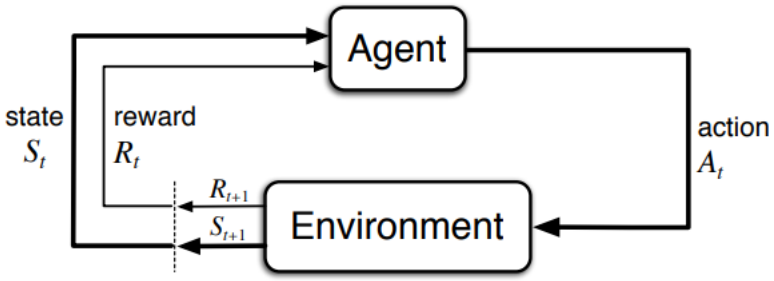


Figure 3.3. *MDP interaction, Source: [5].*

In reinforcement learning, the purpose or goal of the agent is formalized in terms of a special signal, called the reward, passing from the environment to the agent. At each time step, the reward is a simple number. Informally, the agent's goal is to maximize the total amount of reward it receives. This means maximizing not immediate reward, but cumulative reward in the long run [5]:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3.9}$$

The extraction of the recursive form of equation (3.9) is essential in dynamic programming and further analysis of reinforcement learning:

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \\ &= R_{t+1} + \gamma \left(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots \right) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{3.10}$$

Now, in order to continue, it is necessary to define certain functions that estimate how good it is for the agent to be in a particular state or to take a specific action in that state. Initially, it is obvious that future rewards depend on the actions the agent will perform. That is why it is necessary to define a term called policy denoted as π .Formally, a policy is a mapping from states to probabilities of selecting each possible action. If the agent is following policy π at time t, then $\pi(a|s)$ denotes the conditional probability that $A_t = a$ given $S_t = s$. Ultimately, the objective is to maximize the expected cumulative discounted reward by finding an optimal policy π^*.

The state-value function under a policy π is defined as:

$$V^{\pi}(s) = \mathbb{E}_{\pi} \left[G_t \mid S_t = s \right] \tag{3.11}$$

and the action-value function of taking an action a in state s under a policy π is defined as:

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi} \left[G_t \mid S_t = s, A_t = a \right] \tag{3.12}$$

It is important to note that in this form, these equations cannot be used properly. By using the recursive property in eq. 3.10, the Bellman equations are defined:

The Bellman equation for the state-value function $V^{\pi}(s)$ [5] is:

$$\begin{aligned} V^{\pi}(s) &= \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} P(s'|s, a) \left[R(s, a) + \gamma V^{\pi}(s') \right] \\ &= \mathbb{E}_{a \sim \pi(\cdot|s)} \left[\sum_{s' \in \mathcal{S}} P(s'|s, a) \left[R(s, a) + \gamma V^{\pi}(s') \right] \right] \\ &= \mathbb{E}_{a \sim \pi(\cdot|s)} \left[R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) V^{\pi}(s') \right] \end{aligned} \tag{3.13}$$

Essentially, this equation averages over all the possibilities, weighting each by its probability of occurring.

Similarly, the Bellman equation for the action-value function $Q^{\pi}(s, a)$ is :

$$Q^{\pi}(s,a) = R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a) \sum_{a' \in \mathcal{A}} \pi(a' \mid s') Q^{\pi}(s',a') \tag{3.14}$$

Now, in order to find the optimal solution for the policy function π^* the Bellman optimality equations are introduced:

For the optimal state-value function $V^*(s)$, the Bellman optimality equation is:

$$V^*(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P(s'|s,a) \left[R(s,a) + \gamma V^*(s') \right] \tag{3.15}$$

Similarly, the optimal action-value function $Q^*(s,a)$ satisfies:

$$Q^*(s,a) = \sum_{s' \in \mathcal{S}} P(s'|s,a) \left[R(s,a) + \gamma \max_{a'} Q^*(s',a') \right] \tag{3.16}$$

These equations form the basis for many reinforcement learning algorithms, including dynamic programming, Monte Carlo methods, and temporal difference learning but most importantly approximate solution methods which are more applicable in this thesis due to the complexity of the action and more importantly the state space.

3.3.2 Reinforcement learning algorithms

Now, having established the baseline theory of reinforcement learning with MDPs and Bellman equations, the introduction to RL continues with the main algorithms and the categorization of them, which is split mainly into two parts. The first part is the **model-based** algorithms, in which methods rely on planning as their primary component, which means the agent already knows the model-environment and its probabilities of every action and state, therefore it tries to create an optimal plan to achieve its goal with the maximization of the cumulative discounted reward G_t. As is obvious, in systems like Kubernetes it is very hard to know the model-environment and its probabilities to begin with. For that reason alone, the analysis of this thesis is solely focused in the second part of the RL algorithms, with the second part consisting of **model-free** algorithms in which methods primarily rely on learning, meaning the agent does not know the model-environment itself, as we defined it, but can gain access to its properties via experience and interaction with it.At the same time, a fundamental challenge in reinforcement learning is the **exploration–exploitation** dilemma. At every decision point, the agent must choose between exploiting its current knowledge to maximize immediate reward or exploring less known actions that might lead to higher long-term cumulative discounted reward. Exploitation leverages the best known policy so far, ensuring stability and short-term performance, while exploration seeks additional information about the environment, which is essential for improving the policy over time. Having briefly analyzed the main differences between the two parts of reinforcement learning and the importance of exploration–exploitation

dilemma, it is essential to focus deeply into the modern model-free algorithms which generally, can be further split into on-policy and off-policy. On-policy algorithms attempt to evaluate or improve the policy that is used to make decisions, whereas off-policy algorithms evaluate or improve a policy different from that used to generate the data:

- **Value-based Algorithms**: Value-based algorithms mainly focus on the estimation of value functions rather than direct policy optimization. These algorithms aim to compute the state-value function $V^{\pi}(s)$ or the action-value function $Q^{\pi}(s, a)$ and derive optimal behavior implicitly by selecting the actions that maximize these value estimates, typically through greedy or ϵ-greedy policies. Algorithms such as Q-Learning [62] and State–action–reward–state–action (SARSA) [5] follows this approach, iteratively updating value estimates based on the Bellman equations. The advantage of value-based methods lies in their conceptual simplicity and proven convergence properties in discrete state and action spaces. For example, the most well known value-based algorithm in reinforcement learning is Q-learning, which is an off-policy algorithm which seeks the optimal action-value function $Q^{*}(s, a)$ satisfying the Bellman equation:

$$Q^{*}(s, a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q^{*}(S_{t+1}, a') \,\middle|\, S_t = s, A_t = a\right]. \tag{3.17}$$

  In practice, Q-Learning iteratively approximates $Q^{*}(s, a)$ using the following update rule at each time step t:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + a\left[R_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a') - Q(s_t, a_t)\right], \tag{3.18}$$

  where $a \in (0, 1]$ is the learning rate. The aforementioned algorithm follows the fundamental logic of Temporal-Difference bootstrapping [5].

  However, value-based algorithms can face challenges when applied to environments with high dimensional or continuous action spaces such as Kubernetes systems, due to a nearly infinite search space, and employing improvements like discretizing the action space may produce sub-optimal solutions [63].

- **Policy-based Algorithms**: Policy-based algorithms on the other hand, primarily focus on the direct estimation and optimization of the policy π, or to be more precise a parameterized policy $\pi_{\partial}(a|s, \partial)$, with the use of linear or non-linear functions. The following method named Policy-gradient, is an essential method used in policy-based algorithms in RL. As mentioned in previous sections, the policy is a function that outputs an action(s) based on the current state, but when stochasticity is involved, this is done probabilistically. In order to achieve that, the policy parameter ∂ is updated in the direction of the gradient of the objective function $J(\partial)$ using gradient ascent:

$$\partial_{t+1} \leftarrow \partial_t + a\nabla_{\partial} J(\partial_t) \tag{3.19}$$

where a denotes the learning rate and $\nabla_{\partial} J(\partial)$ is given by the policy gradient theorem [5]:

$$\nabla_{\partial} J(\partial) = \mathbb{E}_{\pi_{\partial}} \left[\nabla_{\partial} \ln \pi_{\partial}(a|s)\, Q^{\pi_{\partial}}(s,a) \right] \tag{3.20}$$

where $Q^{\pi_{\partial}}(s,a)$ here is the action-value function under policy π_{∂}. The problem that occurs here is that in the current point the estimation involves only the policy π_{∂} and thus it helps us compute only the gradient of $ln\pi_{\partial}$, still not any estimation for action-value function $Q^{\pi_{\partial}}(s,a)$. In order to surpass this problem REINFORCE [5] algorithm was introduced, trying to estimate the action-value function with the use of Monte-Carlo estimators, thus at the end of each episode an approximation of the action-value function is derived by averaging the actual rewards observed after visiting state-action pairs, using the policy π_{∂}:

$$Q^{\pi}(s,a) \approx \frac{1}{N(s,a)} \sum_{i=1}^{N(s,a)} G_{t_i} \tag{3.21}$$

(3.22)

where G_{t_i} is the same as (3.9) and N(s,a) is the number of times state–action pair (s,a) appeared in the episode. Remember from (3.12), the action-value function is actually the expected value of the cumulative reward, thus using the estimator mentioned, one can approximate the expected value as the average over the derived samples. The same estimator can be used to estimate the state-value function $V^{\pi}(s)$, making the pure Value-based Algorithms simple to compute. By making all the aforementioned calculations the equation 3.19 is transformed accordingly:

$$\partial_{t+1} \leftarrow \partial_t + a\, \gamma^t\, G_t\, \nabla_{\partial} \ln \pi_{\partial}(A_t \mid S_t) \tag{3.23}$$

where A_t, S_t are the action and state set respectively. However, as a Monte Carlo method REINFORCE may be of high variance and thus produce slow learning. For this problem alone, the generalization of REINFORCE [5] method has occurred, instead of just using G_t, the problem of variance is reduced with a baseline function $b(s)$ as long as it does not vary with the learning parameter a. Introducing the next category of algorithms, the baseline function is set to the state-value function and the approximations made with Monte-Carlo are replaced with another method.

- **Actor-Critic Algorithms**: Algorithms that try to approximate both the value functions and the policy function are called Actor-Critic algorithms, with the actor focusing on learning the policy and the critic focusing on learning the value function (which can be either state or action value function). This hybrid approach combines the advantages of both value-based and policy-based algorithms, with the advantage of evaluating state–action pairs and the flexibility of policy-based methods in directly optimizing the policy. This interaction reduces the high variance typically found with pure policy gradient methods and simultaneously addresses the inefficiencies of purely value-based algorithms in handling continuous or high-dimensional action spaces. Continuing the analysis from the Policy-based algorithms, one obvious problem that occurs from that analysis is that, the updates happen after a full episode and not online. To overcome this issue, Actor-Critic architectures are introduced, with approximations done with the use of Temporal-Difference [5]. As such, the computation of the critic responsible of the value function (usually state-value) is done online, incrementally with lower variance as before and the same kind of bias in the actor as before. This way, instead of relying on the whole episodic G_t reward, the use of immediate reward is done. The final form of the algorithm and its generalization with eligibility traces are shown below respectively:

One-step Actor–Critic (episodic), for estimating $\pi_\theta \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
  Initialize S (first state of episode)
  $I \leftarrow 1$
  Loop while S is not terminal (for each time step):
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla \ln \pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

Figure 3.4. *Actor-Critic REINFORCE algorithm with temporal-difference, Source: [5].*

Actor–Critic with Eligibility Traces (episodic), for estimating $\pi_\theta \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: trace-decay rates $\lambda^{\boldsymbol{\theta}} \in [0,1]$, $\lambda^{\mathbf{w}} \in [0,1]$; step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
  Initialize S (first state of episode)
  $\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \mathbf{0}$ (d'-component eligibility trace vector)
  $\mathbf{z}^{\mathbf{w}} \leftarrow \mathbf{0}$ (d-component eligibility trace vector)
  $I \leftarrow 1$
  Loop while S is not terminal (for each time step):
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{z}^{\mathbf{w}} \leftarrow \gamma \lambda^{\mathbf{w}} \mathbf{z}^{\mathbf{w}} + \nabla \hat{v}(S,\mathbf{w})$
    $\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \gamma \lambda^{\boldsymbol{\theta}} \mathbf{z}^{\boldsymbol{\theta}} + I \nabla \ln \pi(A|S,\boldsymbol{\theta})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \mathbf{z}^{\mathbf{w}}$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \delta \mathbf{z}^{\boldsymbol{\theta}}$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

Figure 3.5. *Actor-Critic REINFORCE algorithm with eligibility traces, Source: [5].*

Finally, in many of these cases instead of using the Q-function, another function called advantage function is used, which quantity describes how good the action a is, as compared to the expected return when following directly policy π:

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s) \tag{3.24}$$

3.3.3 Deep Reinforcement learning algorithms

Having analyzed the foundational concepts of reinforcement learning, it is essential to outline that serious limitations arise when these classical methods are applied to environments with large or continuous state and action spaces. Traditional methods, which explicitly store value functions or policies, become computationally heavy and, at the same time, increasingly costly as the dimensionality of the problem arises, a phenomenon often referred to as the "curse of dimensionality" [64]. To address these challenges, an addition to this field is presented called Deep Reinforcement Learning, which integrates the classical methods aforementioned with deep neural networks, mitigating the problems described earlier. The ability of these networks to model complex and non-linear relationships enables deep reinforcement learning algorithms to approximate value functions or policies even in environments with large and highly dynamic state and action spaces. This makes them particularly suitable for system-level decision-making tasks, such as resource allocation, scheduling, and scaling, where the environment's dynamics are non-stationary and difficult to capture with traditional methods.

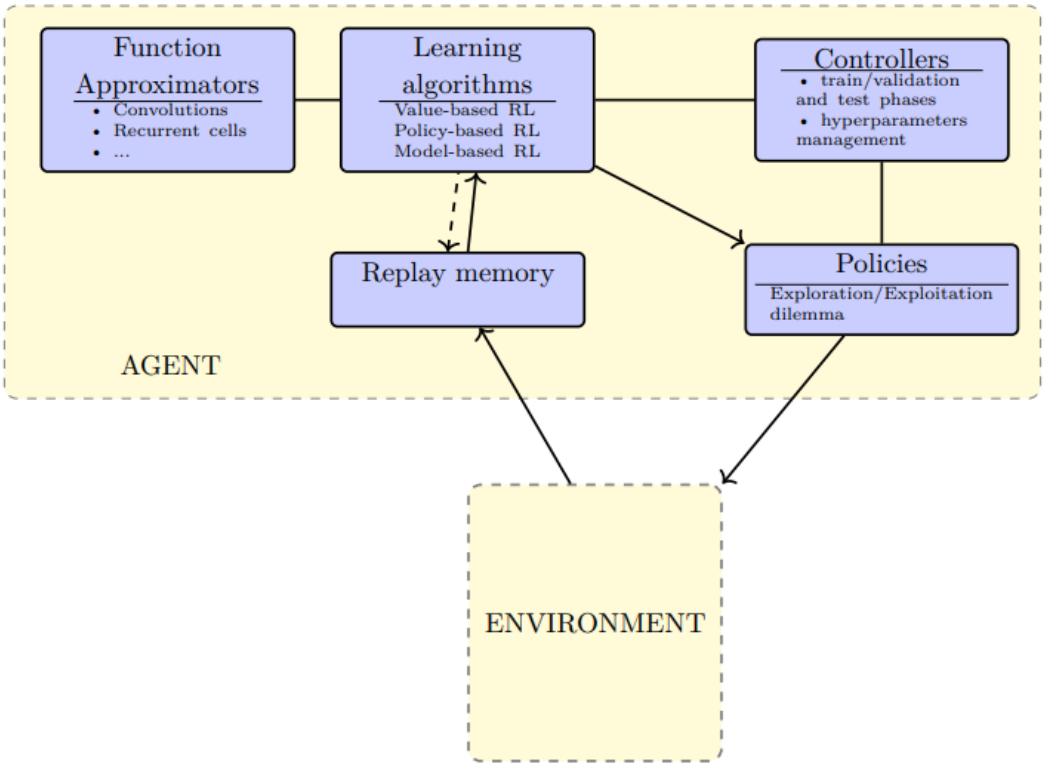


Figure 3.6. *General Deep Reinforcement learning schema, Source: [6].*

- **Value-based algorithms for Deep RL**: Building on the principles of the aforementioned classical value-based reinforcement learning, deep value-based methods leverage neural networks to directly approximate action-value functions in environments with large or continuous state spaces, instead of just using the classic tabular method. One of the first innovations in this type of algorithms was the **Deep Q-Network (DQN)** [65] which architecture is based on the classic tabular Q-learning algorithm:

$$y_t = r_t + \gamma \max_{a' \in \mathcal{A}} Q(s', a'; \theta^-), \tag{3.25}$$

where r_t is the reward received from $R(s, a, s')$, θ is the network parameters that directly estimates the Q function and its delayed version θ^- which is periodically synchronized with the online network with parameter θ . In fact, the delayed version is kept stable in order to have a stable target for the other network to achieve.The loss function is the mean squared error between the predicted Q-value and the target that must be reached, which in practice will be minimized by gradient descent :

$$\mathcal{L}(\theta) = \mathbb{E}_{(s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}} \left[(y_t - Q(s_t, a_t; \theta))^2 \right] \tag{3.26}$$

$$\nabla_\theta \mathcal{L}(\theta) = \mathbb{E}_{\mathcal{D}} \left[2 (Q(s, a; \theta) - y) \nabla_\theta Q(s, a; \theta) \right] \tag{3.27}$$

$$\theta \leftarrow \theta - \alpha 2 \mathbb{E}_{\mathcal{D}} \left[(Q(s, a; \theta) - y) \nabla_\theta Q(s, a; \theta) \right] \tag{3.28}$$

$$\theta^- \leftarrow \theta \qquad \text{(every } C \text{ steps, hard update)} \tag{3.29}$$

$$\theta^- \leftarrow \tau \theta + (1 - \tau) \theta^- \qquad \text{(every } C \text{ steps, soft update)} \tag{3.30}$$

Note: $(s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$ often called mini-batch, is sampled from the replay memory D, which is a special memory that keeps,the last N_{replay} experience gathered from the environment by following the $\epsilon - greedy$ policy[1]. When the replay memory has enough samples (e.g. after 1000 steps), a mini-batch is sampled and then the gradient descent in equation 3.26 is computed. The major advantage of the replay memory is that one mini-batch update has less variance compared to a single tuple update. Consequently, it provides the possibility to make a larger update of the parameters, while having an efficient parallelization of the algorithm [6].

- **Policy-based and Actor-Critic algorithms for Deep RL**: The policy-based algorithms can be extended further with the use of deep learning approaches such as neural networks in order to model non-linearities as part of both the policy function and the value function. The Actor-Critic approach described analytically earlier,

[1] it takes a random action with probability ϵ and follows the policy given by $\arg\max_{a \in \mathcal{A}} Q(s, a; \theta_k)$ with probability $1 - \epsilon$.

has the advantage of being simple, yet it is not computationally efficient as it uses a pure bootstrapping technique that is prone to instabilities and has a slow reward propagation backwards in time. The ideal architecture is one that can be both sample-efficient and computationally efficient. **Trust Region Policy Optimization (TRPO)** [23] is one of the first innovative algorithms that was introduced, improving upon standard policy gradient methods by introducing a clever condition to ensure stable and monotonic policy improvement. Traditional policy gradients can make overly large updates, causing drastic policy shifts and unstable learning making them hard to converge. In order to address that, TRPO suggests that maximizing a different, surrogate objective is computationally easier (derived from importance sampling which is a Monte-Carlo method) and using the KL-divergence [24] ensures that the policy doesn't change drastically but is kept within a Trust region. This way ensuring stable and robust change and improvement from policy $\pi_{\partial_{\text{old}}}$ to policy π_{∂}.

$$\max_{\partial} \quad \mathbb{E}_{s,a\sim\pi_{\partial_{\text{old}}}}\left[\frac{\pi_{\partial}(a \mid s)}{\pi_{\partial_{\text{old}}}(a \mid s)} A^{\pi_{\partial_{\text{old}}}}(s,a)\right] \tag{3.31}$$

$$\mathbb{E}_{s\sim\pi_{\partial_{\text{old}}}}\left[D_{\text{KL}}(\pi_{\partial_{\text{old}}}(\cdot \mid s) \parallel \pi_{\partial}(\cdot \mid s))\right] \leq \delta \tag{3.32}$$

where δ is the trust region threshold controlling the maximum allowed KL divergence between old and new policies, $D_{\text{KL}}(\pi_{\partial_{\text{old}}} \parallel \pi_{\partial})$ is the Kullback–Leibler divergence between the old policy $\pi_{\partial_{\text{old}}}$ and the new policy π_{∂}, measuring how much the new policy diverges from the old one, A^{π} is the advantage function (3.24).

While TRPO provides stable policy updates through a trust-region constraint on the average KL divergence, it is computationally expensive because it requires conjugate gradient optimization and a line search to enforce this constraint. Instead, a simpler approach was made through the **Proximal Policy Optimization (PPO)** [25] algorithm, in a way that it replaced the hard KL-divergence constraint with a clip approximation:

$$r_t(\partial) = \frac{\pi_{\partial}(a_t \mid s_t)}{\pi_{\partial_{\text{old}}}(a_t \mid s_t)} \tag{3.33}$$

$$L^{\text{CLIP}}(\partial) = \mathbb{E}_t\left[\min\left(r_t(\partial)\,A_t,\ \text{clip}(r_t(\partial),\ 1-\epsilon,\ 1+\epsilon)\,A_t\right)\right] \tag{3.34}$$

where ϵ is a hyperparameter, usually 0.2, the objective function has changed with two terms, the first one being the original surrogate objective, the second term modifies the surrogate objective by clipping the probability ratio, which removes the incentive for moving r_t outside of the interval $[1-\epsilon, 1+\epsilon]$. Finally, a simpler approach with a pure Actor-Critic algorithm is the **synchronous Advantage Actor-Critic (A2C)** [26] in which, as described before, combines a network for the actor and a network for the critic (in the default implementation of the library, one network is

used for both) with an entropy term added in the policy optimization function:

$$\nabla_{\partial} J_{\text{actor}}(\partial) = \mathbb{E}_t \left[\nabla_{\partial} \log \pi_{\partial}(a_t \mid s_t) A_t + \beta \nabla_{\partial} H(\pi(s_t; \partial)) \right] \tag{3.35}$$

$$L_{\text{critic}}(\partial) = \mathbb{E}_t \left[\left(G_t - V_{\partial}(s_t) \right)^2 \right] \tag{3.36}$$

where H is the entropy and the hyperparameter β controls the strength of the entropy regularization term. The regression problem being solved via the critic, combines an estimation of the cumulative reward G_t with Temporal difference (other estimations including Monte Carlo and Generalized Advantage Estimation can be used) and the predicted value of the critic network.

Part II

Implementation

Chapter 4

Architecture

In this section, the architecture of the entire thesis is extensively outlined. Initially, the Infrastructure's details are defined and in later stage every component of the proposed pipeline is analyzed, emphasizing each and every layer that was created and deployed. Lastly, all details about the deployed RL Agents of this case study are also analyzed.

4.1 Infrastructure setup

The Kubernetes cluster consists of 5 nodes with one being the master node and the rest being the worker nodes. Each of the node is equipped with a virtualized Intel Skylake processor exposing 4 vCPUs, each operating at approximately 2.2 GHz. It runs on a 64-bit architecture and supports virtualization (VT-x) along with SIMD instruction sets such as SSE. The processor includes 128 KiB of L1d cache in total (4 instances), 128 KiB of L1i cache in total (4 instances), 16 MiB of L2 cache in total (4 instances), and up to 64 MiB of L3 cache in total (4 instances). Each node has 15.61 GiB of RAM, with about approximately 14 GiB available. This hardware configuration is suitable for running autoscaling microservices in Kubernetes, as it provides the necessary processing power and memory capacity. Finally, each node runs on Ubuntu 22.04.5 LTS with Linux kernel version 5.15.0-144-generic. At this point it is important to note that all scripts were executed from the master node, including training of all RL and ML models, client traffic etc. , a choice made for simplicity reasons. Simultaneously, the Kubernetes cluster was provisioned using Kubernetes v1.30.3, with cluster management performed via kubectl v1.30.3 . At the same time, containerization performed using Docker v28.1.1 ensuring compatibility with Kubernetes deployments and the underlying aforementioned system. These details are critical, as variations in tool versions can influence cluster behavior, API availability, and overall system performance, particularly when integrating machine learning pipelines that rely on resource allocation.

4.2 Core components of the proposed solution

This section provides the complete architecture in order to illustrate how the various components previously analyzed interact to manage and optimize resource allocation in the Kubernetes cluster. The architecture is composed of 6 main layers which are

illustrated briefly on figure 4.1.

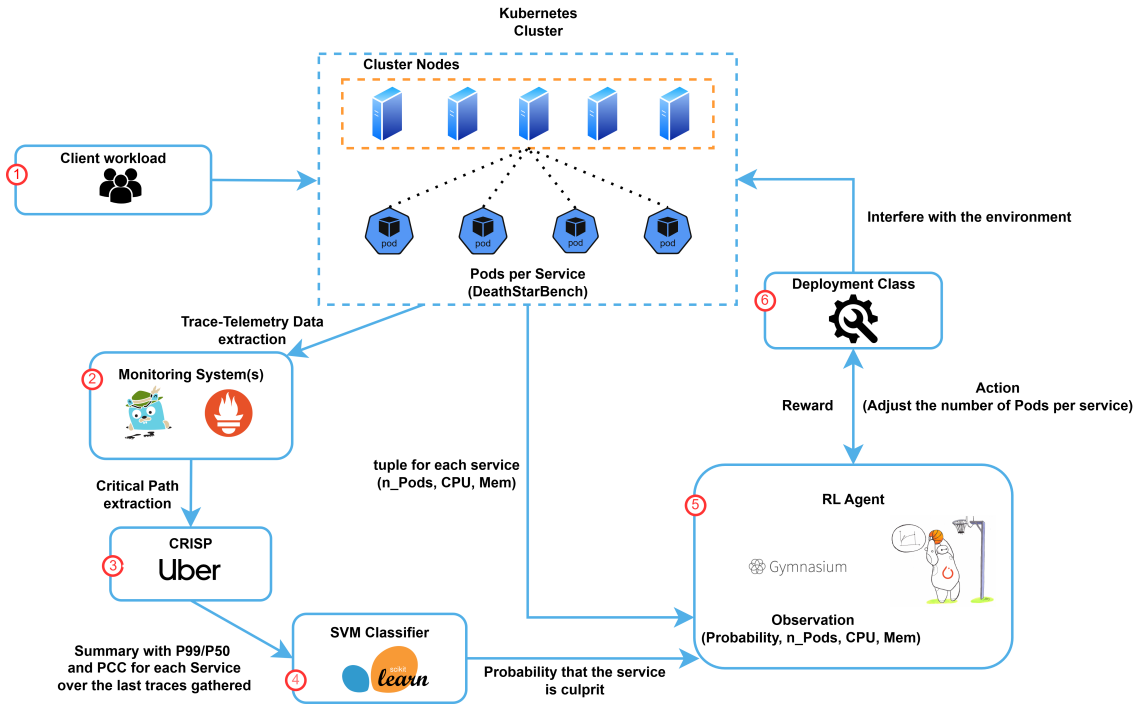


Figure 4.1. *Proposed pipeline for resource allocation.*

Before analyzing each layer extensively, it is important to note at this point that the microservices benchmark used in this thesis, is taken from DeathStarBench [8] which is a comprehensive open-source benchmark suite that includes five realistic, end-to-end cloud applications constructed using microservices. Built with widely adopted frameworks such as Apache Thrift [28]and gRPC [29], the suite models representative services including a social network, an e-commerce platform, a media review and streaming site, a secure banking system, and an IoT-based drone coordination service. Unlike previous benchmarks that focus on simple or single-tier services, DeathStarBench captures the complexity and interdependencies of real-world microservices at scale, with each application comprising dozens of microservices built and written in a mix of programming languages.

The benchmark is specifically designed to evaluate the broad system-level implications of microservices across the cloud and edge computing stack. DeathStarBench reveals the limitations of traditional cluster management strategies, showing how microservice dependencies can lead to backpressure, cascading Quality of Service (QoS) violations, and delayed recovery from performance degradation. Additionally, it explores the implications of deploying microservices in serverless environments, highlighting trade-offs between cost, scalability, and latency. Through its modular, extensible, and heterogeneous design, it provides a powerful foundation for studying the performance, efficiency, and scalability challenges introduced by the microservices paradigm in modern cloud and edge systems with the help of static policies such as heuristics or even with the help of machine learning techniques. The specific architecture chosen is called SocialNetwork which simulates a

social network with unidirectional follow relationships, implemented with loosely-coupled microservices, communicating with each other via Thrift RPCs and its architecture is depicted below:

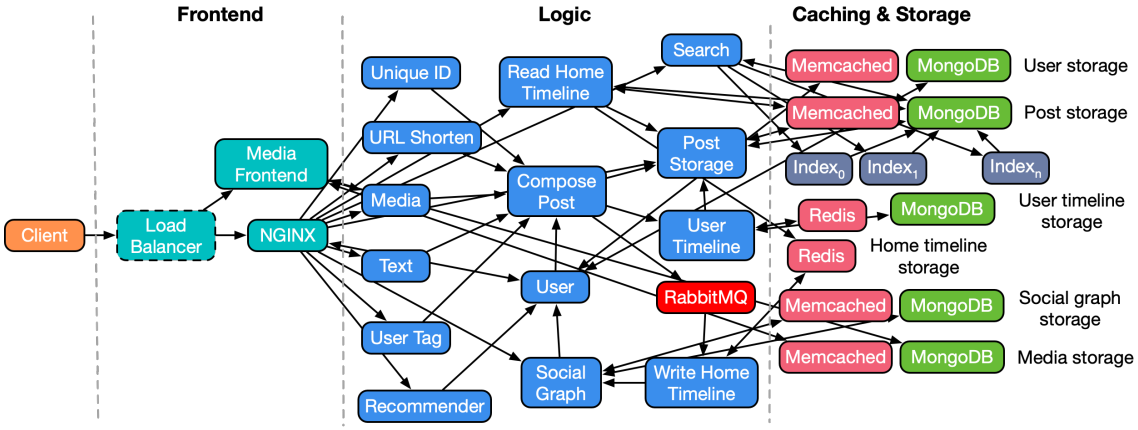


Figure 4.2. *SocialNetwork benchmark architecture,Source: [8].*

4.2.1 First and second layer (Workload and Monitoring)

Continuing with the core architecture of this thesis, the **first layer** consists of the client workload, which represents the end users or the external systems that query the cloud microservices. More specifically, the operation selected to be used in order to stress the system is named ComposePosts which creates and publishes posts into the social network. To effectively create multiple users, each using the aforementioned operation, the http workload generator wrk2 [27] was leveraged as the author of DeathStarBench recommended and also used the provided Lua [66] script, named compose-post.lua, in order to create and publish posts accurately. The **second layer** corresponds to the monitoring systems, which are tasked with collecting traces and telemetry data and presenting performance metrics from the various microservice instances in this architecture. More specifically, Prometheus is responsible for collecting metrics such as the usage of CPU and memory from each Service, and at the same time, Jaeger is responsible of capturing traces that form the internal dependence graph of each microservice to another. This graph is particularly useful for next layers. The further in depth analysis of this layer was done in previous chapter.

4.2.2 Third layer (Critical Path Analysis with CRISP)

After capturing and then processing the aforementioned traces, the data are passed into the **third layer**, consisting of CRISP [7] which is an advanced tool that computes efficiently the complexity of modern microservice systems by isolating the critical path[1] chain of service calls whose performance most directly determines the overall latency of a request. By transforming Jaeger traces into computational DAGs, CRISP identifies syn-

[1]Assuming a weighted directed acyclic graph (DAG) G(V,E) with V vertices and E edges and the starting Vertex being S and the final being Z, a maximal-weight path from S to Z in a task-dependency graph G(V,E) is called a **critical path** and G may contain more than one of them[30].

chronization points and walks each trace in reverse to discover the longest dependency path. When the same method is applied across large volumes of trace data, it aggregates these paths into very important visualizations as flame graphs, heatmaps, and text summaries that spotlight which services consistently contribute to delays. This focused approach drastically reduces noise and highlights opportunities for targeted optimization, whether in service code, infrastructure decisions, or anomaly detection pipelines. At large scale, CRISP also supports two complementary analysis modes: a top-down tool for endpoint specific issues and a bottom-up view to track which internal APIs have system-wide impact . Its real world deployment at Uber traces demonstrated significant operational benefits, identifying hidden bottlenecks, guiding infrastructure choices, and enhancing anomaly detection.

The essence of CRISP lies in its ability to efficiently handle all critical paths in large-scale systems and bring to the surface useful insights about the working cloud-based environment. This tool leverages the structure of the Jaeger traces to find and compute the critical path with the following pseudo-algorithm:

```
def CP(root):
  path = [root]
  if len(root.child) == 0:
    return path
  children = sortDescendingByEndTime(root.children)
  lfc = children[0]
  path.extend(CP(lfc))
  for c in children[1:]:
    if happensBefore(c, lfc):
      path.extend(CP(c))
      lfc = c
  return path
```

Figure 4.3. *Critical path extraction pseudo-algorithm,Source: [7].*

At the same time, the structure of CRISP's framework works in a way that can capture and present also the merge of all the critical paths from the last N traces, which is named Critical Calling Context Tree (CCCT):

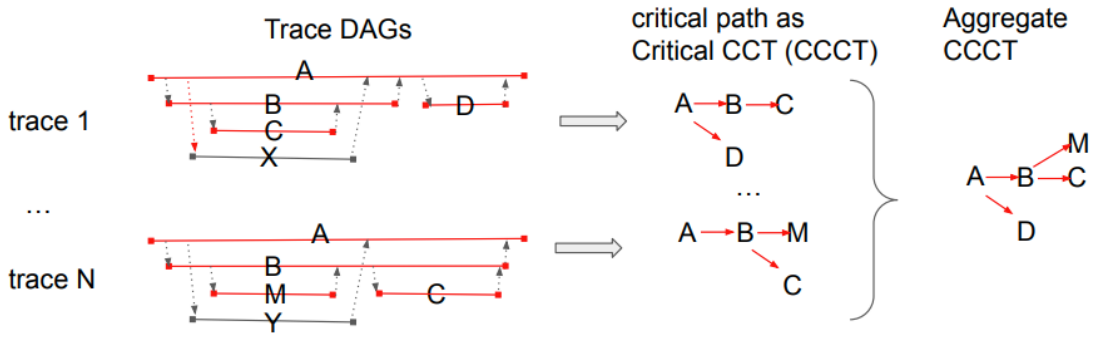


Figure 4.4. *CCCT aggregation,Source: [7].*

Since every node on the critical path encodes quantitative information of its called structure, and since all call paths originate from a common root, the endpoint under investigation, this allows all call paths to be merged into a calling context tree by identifying their common prefixes. The aggregate CCCT essentially summarizes all call paths leading to critical path nodes in all traces, thus it captures the quantitative aspect by associating higher weights to those call paths that are often on the critical path. The weights of the nodes in such a tree would be the summation of the weights of the individual call paths. Specifically with this analysis, CRISP's framework provides numerous percentiles such as P50, P95, P99. Last but not least, the clock drift problem shown in the figures below, is dealt with the happensBefore pseudo-function as it is shown in the algorithm 4.3 above.

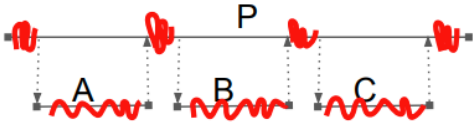


Figure 4.5. *Ideal traces for a parent with three serialized children executions, red lines indicate the critical path, Source: [7]*

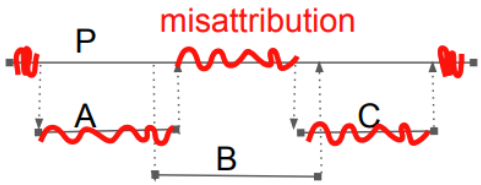


Figure 4.6. *Actual traces due to the clock drift, red lines indicate the critical path, Source: [7]*

4.2.3 Fourth layer (Probabilistic SVM classifier)

In order to leverage the statistical insights and the repetitive patterns that form from each critical path and specifically from each Service, Support Vector Machine was chosen in the **fourth layer**, from the various Machine Learning supervised techniques, with the objective to derive these patterns and more notably, to accurately classify the most probable culprit among the components residing in the critical path. This choice was chosen due to its effectiveness in high dimensional data and also its ability to generalize well in the absence of data. To achieve this kind of generalization, important and representative features must be extracted from all the services that are included into the critical path.

The idea of the original paper [31] was that the first good representation of the data, called **Relative Importance**, is defined as the Pearson correlation coefficient [32] of the

total latency and the individual latency of the Service. This way, the strength in the variance of the total latency correlated with the individual variance of the Service, known as explained variance[33], is measured accordingly. At the same time, the second good representative feature of the data, called **Congestion Intensity**, is defined as the ratio of the 99th percentile of latency over the 50th percentile of latency (also called median) of the Service. This feature simply measures the ability to handle requests with the given allocated resource. It is important to note that the 99th percentile was selected to specifically focus on tail latency, which highlights the worst-case performance experienced by a small fraction of requests in the queue.

Simultaneously, in order for the SVM to effectively classify performance culprits along the critical path, it must be provided with a well-structured and informative dataset that represents the Kubernetes system accurately, but also with controlled labeling of the culprits (supervised problem). For these reasons, experiments were designed with controlled performance injection of a random Pod at each experiment. The controlled performance injections were conducted using Chaos-Mesh [34], a powerful tool that enables the developer to conduct various controlled performance injections e.g. CPU, memory, network and even node failure injections. For this thesis only CPU injections were conducted for simplicity purposes, while a pipeline was created simulating normal conditions with a low workload of a constant number of requests per second for a controlled number of time, e.g. 6 RPS for 5 minutes and a randomly chosen Pod to be injected with high usage of CPU. The assumption made here is that injecting the CPU or any other metric of the specific Pod, represents the behavior of a Pod or in general a Service that needs more resources allocated. Such techniques are called Chaos Engineering which are essential for simulating the stress conditions the system is going to encounter with real-world scenarios [67]. Injecting and then gathering the Jaeger traces can be plotted as follows:

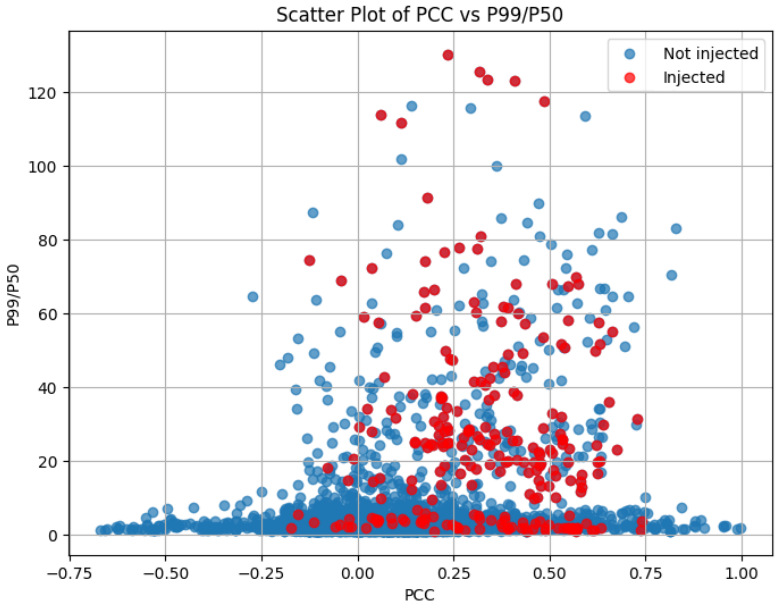


Figure 4.7. *Traces of injected data.*

As can be seen in the plot 4.7, deriving a linear pattern that satisfies the classification objective can be really hard in this case, so the use of kernels is justified. Also, because the output prediction of the classifier will be included in the observation of the RL Agent, an idea that is implemented in this thesis is that instead of using binary classification (0 or 1), a more reliable and safe way to derive the culprit from the critical path is to assign a probability that one Service is the culprit. In that way, even if a Service is partially causing a problem, still its resources will be allocated. To achieve this kind of probabilistic approach, the calibration of the SVM classifier was performed. Calibrating a classifier consists of fitting a regressor (called a calibrator) that maps the output of the classifier to a calibrated probability in [0,1]. The implementation of the SVM classifier and its calibration performed with the machine learning framework scikit-learn [36]. The regressor used in this thesis was the sigmoid regressor which was based Platt's logistic model [35]:

$$p(label = 1 \mid f_i) = \frac{1}{1 + \exp(Af_i + B)}, \tag{4.1}$$

where label is the true label of sample i and f_i is the output of the un-calibrated classifier for sample i. A and B are real numbers to be determined when fitting the regressor via maximum likelihood. Also, in order to use the aforementioned technique for an already fitted classifier, FrozenEstimator Class was used in combination with CalibratedClassifierCV. Given all the previous knowledge and frameworks, probabilistic SVM classification is made for each Service separately and then the output probability is fed on the observation space of the RL Agent.

4.2.4 Fifth layer and Sixth layer (RL Agent and Deployment class)

As described in earlier chapters, RL is the way to go in decision-making environments to achieve adaptive and optimal control, as it enables agents to learn from experience and continuously improve performance in the presence of dynamic interactions from users to the cloud environment. Having analyzed the theoretical foundations of RL agents and at the same time having analyzed all the previous layers practically, the final step is to examine the practical integration of RL with the Kubernetes environment. In order for the RL agent (**fifth layer**) to interact with the Kubernetes environment, it is essential to create a custom environment Class with structure that follows the specifications of the Gymnasium [37] interface, meaning the Methods inside the Class must be designed accordingly:

- **Initialization (`__init__()`)**: The first mandatory requirement is that the environment class should inherit from gymnasium.Env. Also, at the initialization stage the following must be defined:
  - **observation_space**: a gymnasium.spaces object (e.g., Box, Discrete) describing the format of the observations.
  - **action_space**: a gymnasium.spaces object describing available and valid actions. For simplicity reasons in this thesis the chosen action space included

only the modification of the number of Pods per service, a MultiDiscrete[num_services, num_actions] object was initialized, e.g. here num_actions is adding and subtracting Pods from the Service.

  - Internal variables for representing the state of the environment and additional metrics if needed.

- **reset()**: Resets the environment to its initial state. Also, must return a tuple with:

  - **observation**: initial state as defined by observation_space.
  - **info**: dictionary for auxiliary information, can be empty.

- **step()**: Executes the given action and advances the environment by one timestep. Also, must return a tuple with:

  - **observation**: The observation at the end of the step.
  - **reward**: The numerical reward calculated, either with a dynamic function or a static one.
  - **terminated**: True if episode ended successfully.
  - **truncated**: True if episode ended due to a limit.
  - **info**: auxiliary diagnostic information, can be empty.

An important note here is that the observation_space and action_space definitions must align with actual outputs from reset and inputs to step. All methods `__init__()`, reset(), step() and optionally render(), close() with their mandatory structure are further described in the Gymnasium documentation. The initial structures of the custom environment and the deployment class were based on the implementation of gym-hpa [38] authors, which was compatible with the old version of Gymnasium (formerly called gym). Later, they were altered and aligned considering the newest version of Gymnasium (version 1.1.1), the different benchmark used for the microservices architecture (DeathStarBench) and different techniques in this thesis. At the same time, the RL models (PPO, TRPO, A2C) were implemented using the robust and highly stable frameworks stable-baselines3 [39] (version 2.6.0) and stable-baselines3 contrib [40] (version 2.6.0) with the following hyperparameters for each model:

Table 4.1. *Hyperparameters of the PPO algorithm in Stable-Baselines3*

Parameter	Description	Experiment Value
policy	Policy model	MlpPolicy
env	Gymnasium environment (can be vectorized)	SocialNetwork
learning_rate	Step size for optimizer	0.0003
n_steps	Steps per environment per update	150
batch_size	Minibatch size	64
n_epochs	Number of passes per update	10
gamma	Discount factor	0.99
gae_lambda	GAE parameter	0.95
clip_range	Policy loss clipping parameter	0.2
clip_range_vf	Value function clipping	None
normalize_advantage	Normalize advantages	True
ent_coef	Entropy bonus coefficient	0.0
vf_coef	Value function loss coefficient	0.5
max_grad_norm	Gradient clipping norm	0.5
use_sde	State-Dependent Exploration	False
sde_sample_freq	SDE noise resampling frequency	-1
rollout_buffer_class	Custom rollout buffer class	None
rollout_buffer_kwargs	Rollout buffer arguments	None
target_kl	KL divergence early stopping	None
stats_window_size	Stats averaging window size	100
tensorboard_log	Path for TensorBoard logs	tensorboard_log_path
policy_kwargs	Policy keyword arguments	None
verbose	Verbosity level	1
seed	Random seed	None
device	Training device	auto
_init_setup_model	Build model on creation	True

Table 4.2. *Hyperparameters of the TRPO algorithm in Stable-Baselines3 contrib*

Parameter	Description	Experiment Value
policy	Policy model	MlpPolicy
env	Gymnasium environment	SocialNetwork
learning_rate	Step size for optimizer	0.0003
n_steps	Steps per environment per update	150
batch_size	Minibatch size	128
gamma	Discount factor	0.99
gae_lambda	GAE parameter	0.95
ent_coef	Entropy bonus coefficient	0.0
vf_coef	Value function loss coefficient	0.5
max_kl	Maximum KL divergence for updates	0.01
cg_damping	Conjugate gradient damping factor	0.1
cg_max_steps	Maximum conjugate gradient iterations	10
line_search_coef	Coefficient for line search	0.8
n_cpu_tf_sess	Number of CPU threads for TensorFlow session	1
normalize_advantage	Normalize advantages	True
tensorboard_log	Path for TensorBoard logs	tensorboard_log_path
policy_kwargs	Policy keyword arguments	None
verbose	Verbosity level	1
seed	Random seed	None
device	Training device	auto
_init_setup_model	Build model on creation	True

Table 4.3. *Hyperparameters of the A2C algorithm in Stable-Baselines3*

Parameter	Description	Experiment Value
`policy`	Policy model	MlpPolicy
`env`	Gymnasium environment	SocialNetwork
`learning_rate`	Step size for optimizer	0.0007
`n_steps`	Steps per environment per update	150
`gamma`	Discount factor	0.99
`gae_lambda`	GAE parameter	1.0
`ent_coef`	Entropy bonus coefficient	0.0
`vf_coef`	Value function loss coefficient	0.5
`max_grad_norm`	Gradient clipping norm	0.5
`rms_prop_epsilon`	RMSProp optimizer epsilon	1e-5
`use_rms_prop`	Whether to use RMSProp optimizer	True
`normalize_advantage`	Normalize advantages	False
`tensorboard_log`	Path for TensorBoard logs	tensorboard_log_path
`policy_kwargs`	Policy keyword arguments	None
`verbose`	Verbosity level	1
`seed`	Random seed	None
`device`	Training device	auto
`_init_setup_model`	Build model on creation	True

The reward function ensuring that the aforementioned Agents will be trained slowly and in the right direction-manner is the following:

$$\mathrm{R} = \sum_{d \in D} [a \cdot SLO_d + (1 - a) \cdot Align_d] \tag{4.2}$$

where

$$SLO_d = 1 - w_d \tag{4.3}$$

$$Align_d = \frac{1}{1 + |p_d - r_d|} \tag{4.4}$$

$$D = \text{set of deployments} \tag{4.5}$$

$$p_d = \text{number of pods for deployment } d \in D \tag{4.6}$$

$$r_d^{(\text{cpu})} = \text{desired number of replicas for deployment } d \in D \tag{4.7}$$

$$= \left\lceil p_d \cdot \frac{\text{cpu\_usage}}{\text{cpu\_target\_usage}} \right\rceil \text{(same for memory)} \tag{4.8}$$

$$\tag{4.9}$$

$$w_d = \text{critical weight of deployment } d \in D \tag{4.10}$$

$$a \in [0, 1] \text{ is the weighting factor} \tag{4.11}$$

$$\tag{4.12}$$

$$\text{cpu\_target\_usage} = p_d \cdot \text{cpu\_target} \tag{4.13}$$

$$\text{mem\_target\_usage} = p_d \cdot \text{mem\_target} \tag{4.14}$$

$$\text{cpu\_target} = \text{threshold} \cdot \text{cpu\_request} \tag{4.15}$$

$$\text{threshold} = 0.75 \text{ (same as KHPA)} \tag{4.16}$$

$$\text{cpu\_request} = \text{initial CPU request of deployment.} \tag{4.17}$$

To ensure that the agents interact with the actual Kubernetes through the custom environment Class, the **sixth layer** which consists of another Class named DeploymentStatus, bridges the gap between these two by ensuring the right communication via the Kubernetes API. The main methods that are defined inside this class are the following:

- **__init__()**: Initializes the mandatory information that must be included for every Service, e.g. cpu_request, cpu_limit, max_pods, min_pods etc. and finally the critical_weight which denotes the probability of being a culprit as previously mentioned. The initial values are defined for each Service separately as can be seen below:

Table 4.4. *Initial resource configuration of SocialNetwork microservices in Kubernetes*

Service	CPU (req/limit)(MB)	Mem (req/limit)(MB)
compose-post-service	100 / 300	100 / 300
home-timeline-redis	200 / 300	200 / 300
home-timeline-service	100 / 300	100 / 300
jaeger	100 / 300	600 / 800
media-frontend	100 / 300	100 / 300
media-memcached	200 / 300	200 / 300
media-mongodb	200 / 300	200 / 300
media-service	100 / 300	100 / 300
nginx-thrift	200 / 300	200 / 300
post-storage-memcached	200 / 300	200 / 300
post-storage-mongodb	200 / 300	200 / 300
post-storage-service	100 / 300	100 / 300
social-graph-mongodb	200 / 300	200 / 300
social-graph-redis	200 / 300	200 / 300
social-graph-service	100 / 300	100 / 300
text-service	100 / 300	100 / 300
unique-id-service	100 / 300	100 / 300
url-shorten-memcached	200 / 300	200 / 300
url-shorten-mongodb	200 / 300	200 / 300
url-shorten-service	100 / 300	100 / 300
user-memcached	200 / 300	200 / 300
user-mongodb	200 / 300	200 / 300
user-service	100 / 300	100 / 300
user-mention-service	100 / 300	100 / 300
user-timeline-mongodb	200 / 300	200 / 300
user-timeline-redis	200 / 300	200 / 300
user-timeline-service	100 / 300	100 / 300

- **update_obs_k8s()**: Mainly called before reward calculation at step() method to ensure right observation values. Queries Prometheus and fetches the desired metrics which will be used later to calculate the desired Pod replicas for each Service.
- **update_replicas()**: Called inside the update_obs_k8s() method. The main purpose of this method is the calculation of the desired replicas for each Service based on the metrics previously fetched.

- **update_deployment()**: Updates the Deployment via the Kubernetes API and then calls patch_deployment() method to patch it.
- **patch_deployment()**: Patches the Deployment via the Kubernetes API.
- **deploy_pod_replicas()**: Called inside the custom environment by the Agent when the action is to add replica Pods to the Service. Calls update_deployment() to ensure this action.
- **terminate_pod_replicas()**: Called inside the custom environment by the Agent when the action is to subtract replica Pods from the Service. Calls update_deployment() to ensure this action.

Chapter 5

Evaluation

In this chapter, the evaluation of the performance and effectiveness of the proposed system is presented, starting with Section 5.1 that focuses on Critical Component Localization, evaluating the probabilistic SVM effectiveness in identifying critical components. Section 5.2 presents the Training Evaluation, analyzing the behavior of the reinforcement learning agent during training. Section 5.3 describes the workload pattern used for experimentation, detailing the characteristics and generation process of the input load. Finally, Section 5.4 reports the End-to-End Performance results, comparing the trained agent's performance against baseline method of KHPA across key operational metrics such as CPU and memory usage.

5.1 Critical Component Evaluation

This section evaluates the performance of a very important module of this thesis, which is the SVM classifier. The specific classifier was trained on well-annotated data with the library that was mentioned in previous chapters and at the same time to this end, the experiments were conducted with a few data in order to avoid overfit as it is one of the well-known problems of these classifiers. The evaluation results are primarily presented using the ROC curve and the confusion matrix, both of which provide complementary insights into the classifier's effectiveness. The ROC curve illustrates the balance between sensitivity and specificity across different thresholds. At the same time, the confusion matrix provides a clear view of classification outcomes, with particular emphasis on the absence of anomalous data being incorrectly categorized as normal. This property is critical for the reliability of the overall system, as false negatives could potentially harm the system and also mislead the RL agent.

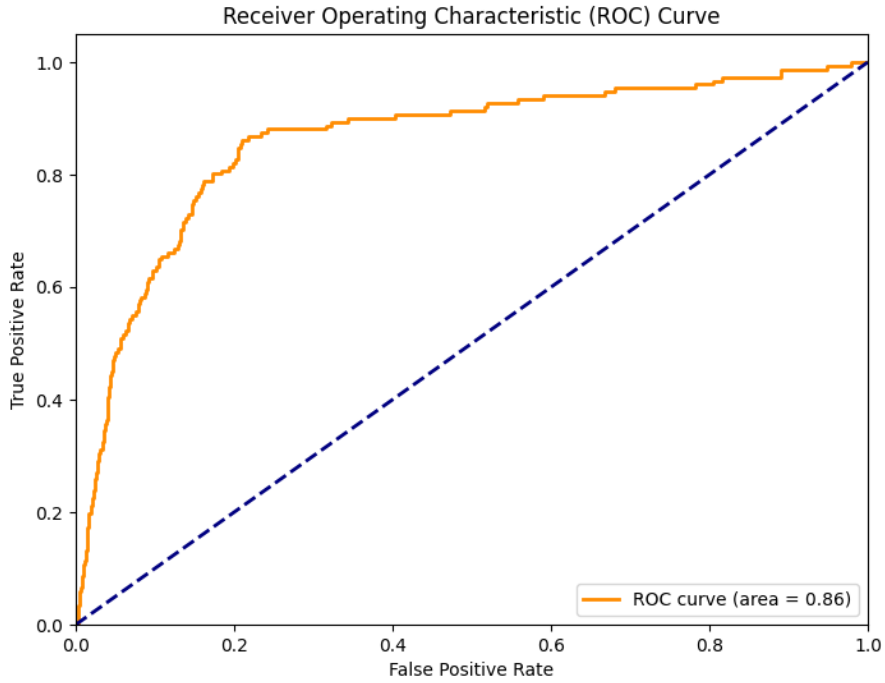


Figure 5.1. *ROC curve of the classifier*

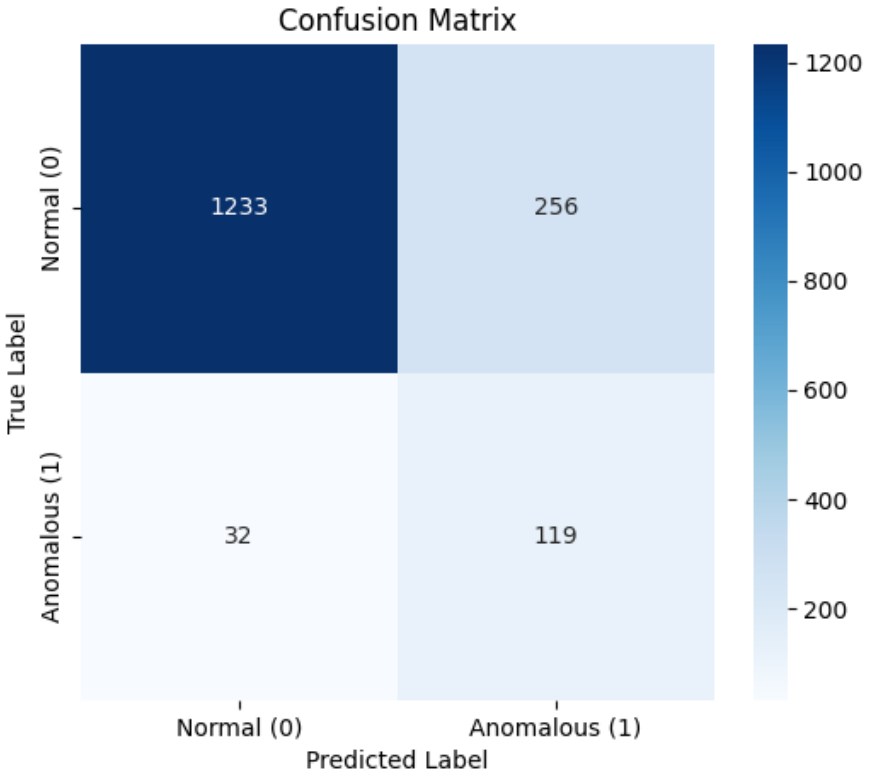


Figure 5.2. *Confusion matrix of the classifier.*

At this point, the classifier is trained and has achieved satisfactory performance in order to perform the critical component annotation and assist the RL agent with the right decision of which component to allocate resources. After this point, the classifier was calibrated to a probabilistic one as described in previous sections.

5.2 Training Evaluation

The training process was carried out using the Kubernetes cluster previously mentioned on the SocialNetwork benchmark of DeathStarBench. Each agent was trained for 1300 episodes and maximum of 20 steps per episode due to the high cost (time-wise) of training in the real cluster. More specifically, the cost of training the PPO, A2C, TRPO agents was approximately 3.8, 4.3, 4.2 days respectively. Similarly to training, the initial conditions of the environment were kept random, emphasizing and testing the ability of both the agents and the KHPA to adapt and generalize to the given problem. To emphasize the importance of increasing the training of RL agents in such complex systems, the PPO agent was further trained for an additional 1300 episodes, reaching the total training episodes to 2600 episodes and reaching the training time to a total of 8.11 days.

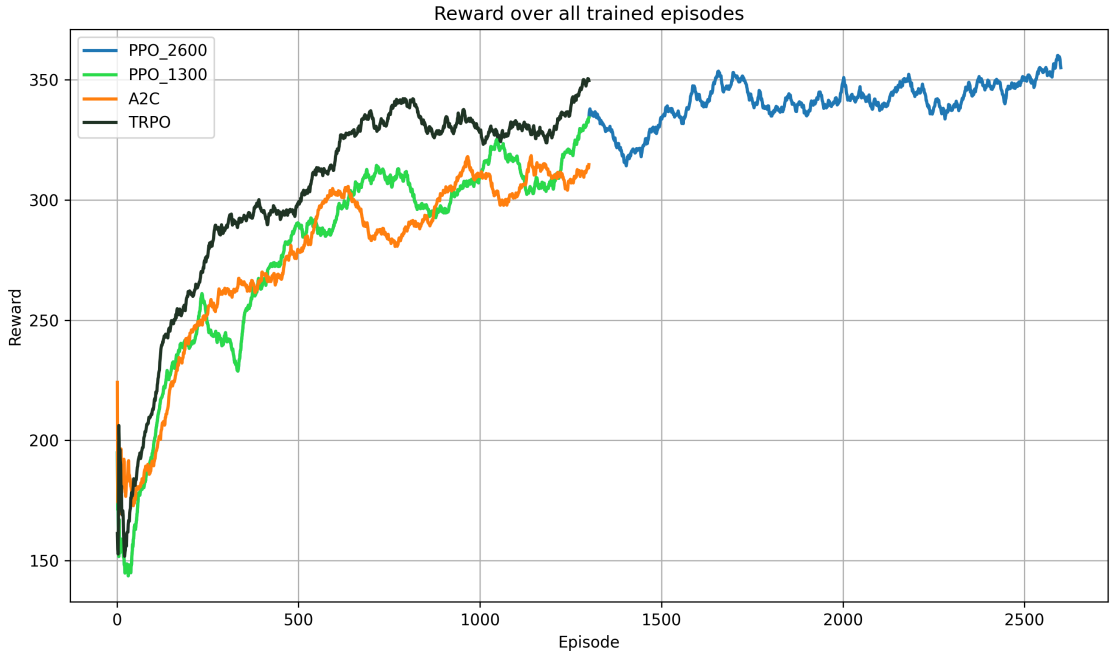


Figure 5.3. *Training results of the RL Agents.*

The results of the training phase are shown in figure 5.3 with smoothed training curves using 100-episode moving average window to give emphasis on the learning trend. Initially, all agents begin to learn with low reward values and gradually improve until the whole training is finished. TRPO agent clearly shows the most rapid and stable improvement throughout the training phase, consistently outperforming the other agents. At the same time, PPO agent follows a similar learning curve to TRPO but with the many fluctuations, especially around the 200-400 episode mark. Finally, the A2C agent follows

a steady learning curve with a slight drop around the 600-800 and 1000-1100 marks.

5.3 Testing workload pattern

The workload pattern that was used with the aim of testing the aforementioned pipeline in the Kubernetes cluster, is composed of the wrk2 open-loop http generator, the same used to train the models, with the following workload pattern:

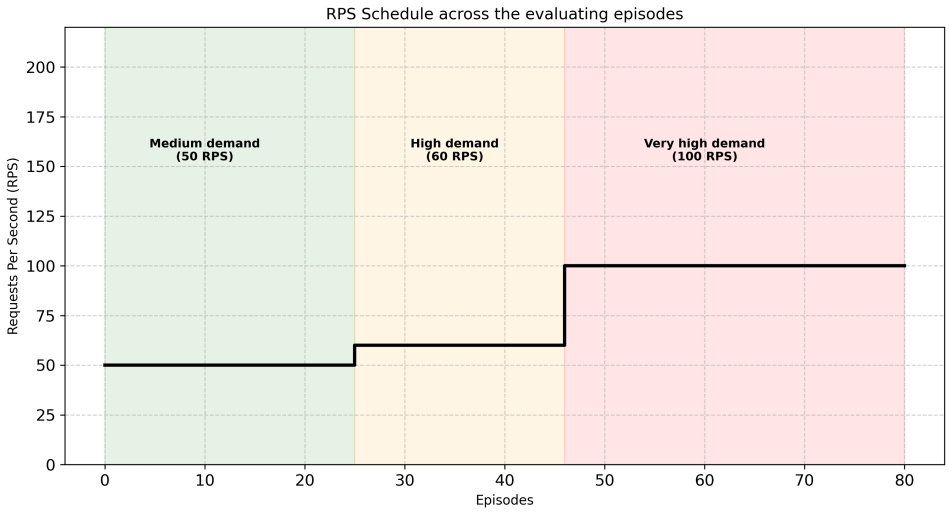


Figure 5.4. *Testing workload pattern.*

The workload pattern used during the evaluation phase was designed to reflect realistic and challenging system conditions. Specifically, the initial stages of evaluation subjected the system to medium request loads, simulating initial periods typically observed in real-world deployments but also the last part of the evaluation consists of higher demand in users, also assuming that after some time there is an increase in workload to the system. In the next section, there is an evaluation of the agents using numerous metrics such as many percentiles of latency and also the number of Pods deployed. The former reflects the actual impact on the user's side and the latter reflects the cost of maintaining the aforementioned performance.

5.4 End-to-End Evaluation

In this section, the end-to-end evaluation of the trained RL agent will be presented at the aforementioned benchmark including the workload pattern and all the other conditions that were previously extensively analyzed. At the beginning, the first 3 plots analyze the agent's performance on the SocialNetwork benchmark with metric evaluation being the end-to-end latency of the system, pointing out the effects on the client's side. Finally, two plots illustrating server-side behavior are presented, highlighting how the agent manages the allocation of resources, specifically the pods within the current namespace, under such complex operating conditions.

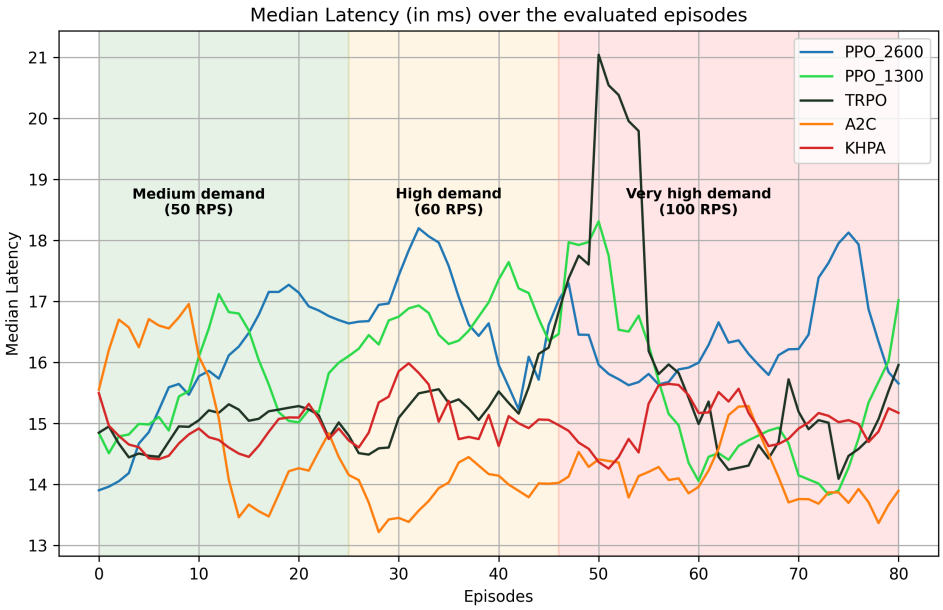


Figure 5.5. *Median Latency over the evaluated episodes.*

The test results of the agents shown in the plots, particularly when compared with the baseline performance of KHPA, can be reasonably justified due to the low number of training episodes and the high dimension space of the action space, as mentioned also in the FIRM ([31], page-12) paper, where similar problems occurred under such training conditions and particularly at such low, episodic-wise, training time. Initially, it can be seen that all agents behave in a way that outperform the performance of KHPA in the metrics of latency. More specifically, the A2C agent starts with an increased latency but around the 10th episode mark, it lowers the latency below the KHPA baseline. That is a sign of an agent that needs time to adapt to the real system options and provide great actions if done so. The TRPO agent seems to match the performance of the KHPA baseline throughout the evaluation stage. The significant spike observed around the change in workload is somewhat justified due to the sudden change from 60 RPS to 100 RPS. Lastly, the first PPO agent that was trained for 1300 episodes, can be seen struggling to adapt to the latency-wise metrics of the system, but around the 55th episode mark it starts to behave better than the KHPA baseline. Also, the second and most trained PPO agent struggle to give good results in the median latency metrics, although it can be easily justified in the pod plots below. It is noticeable that as agents adapt to the workload change, the performance becomes not only comparable with the baseline, but can be seen to outperform the baseline of KHPA, especially by the A2C agent. All latency-related results shown in plot above will be justified at a later stage, where the primary plot illustrating Pod deployment within the Kubernetes cluster will be presented. At the same time, there is a clear indication that even with a low number of training episodes, the agents still have managed to learn valuable patterns, helping the median end-to-end latency performance of the Kubernetes system. The focus now shifts to comparing the

baseline KHPA with the aforementioned agents under the critical latency metrics of the 95th and 99th percentiles, which are essential to measure in such complex and latency-sensitive systems.

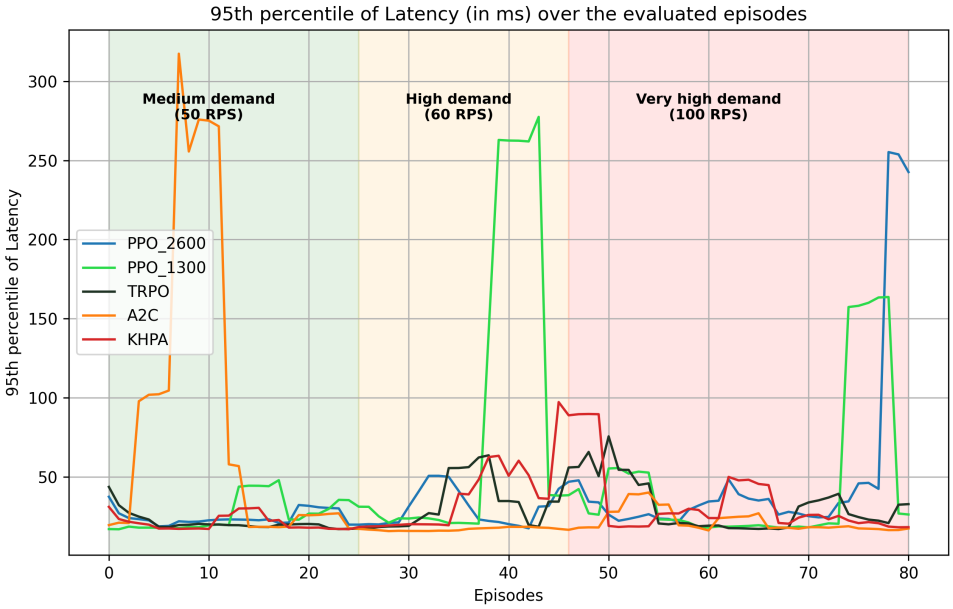


Figure 5.6. *95th percentile of Latency over the evaluated episodes.*

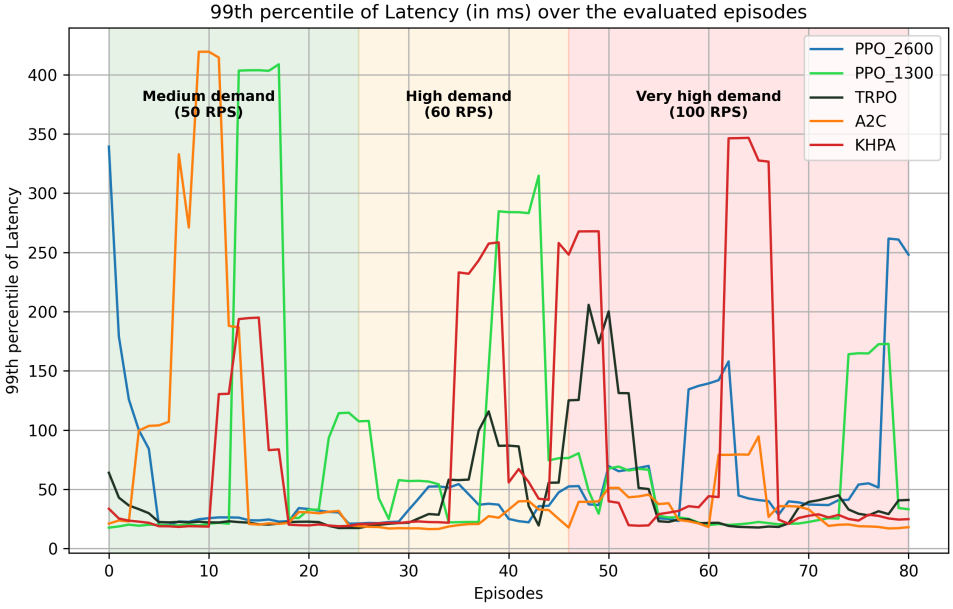


Figure 5.7. *99th percentile of Latency over the evaluated episodes.*

In the 95th percentile plot, it is observed that both the A2C agent and the 1300 episode trained PPO, have some sudden spikes during the first stages of evaluation indicating the inability of handling the most of the queue of requests effectively at all times. Excluding

the aforementioned points, all agents are outperforming the KHPA in many points of the evaluation stage. Lastly, in the 99th percentile of latency plot KHPA and all of the agents have an increased number of latency spikes, highlighting the challenge of maintaining optimal performance throughout the queue under extreme conditions. Significant improvement is demonstrated by the most trained PPO agent which has a small number of high latency spikes. This suggests that, with sufficient training, the PPO agent is able to handle request queues more efficiently and maintain lower response times under varying load conditions. To further assess resource efficiency, the main reward objective of the agents, the average number of Pods deployed and the baseline are presented in the following plot:

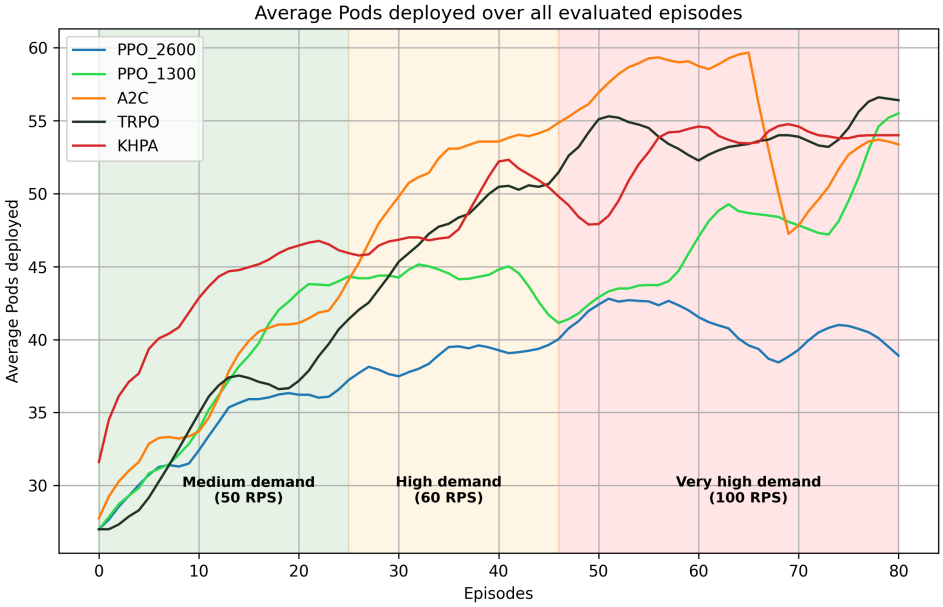


Figure 5.8. *Average Pods deployed over all evaluated episodes.*

The first plot analyzing the number of Pods deployed illustrates the way each agent adapts its scaling behavior in response to the workload change. It can be easily observed that all agents in the entire first period of 50 RPS have deployed less Pods than the KHPA. This is a significant advantage, as fewer Pods directly translate to lower server-side costs. At a later stage, A2C agent overprovisions resources in response with the workload change, demonstrating adaptive scaling behavior but a slight waste of resources as it surpasses the KHPA baseline. It is important to note that in the last 12 episodes of the evaluation stage, the agent adapts to the environment and reduces the number of deployed Pods below the baseline of KHPA, indicating slower adaptation performance. At the same time, the TRPO agent shows better allocation ability in the first stage but in later stages aligns his decisions with the KHPA baseline. Finally, both two PPO agents demonstrate excellent resource allocation capabilities, effectively aligning their provisioning decisions with the workload demand. It is important to note here that the most trained PPO agent achieves a performance of 22.55% less Pods allocated in return for 9.16% higher median latency.

All metrics and results from the evaluated stage are summarized in the table below:

Table 5.1. *Median Differences in evaluated Metrics between Agents and KHPA (Episodes 0-80)*

Metric	PPO_2600	PPO_1300	A2C	TRPO
Pod Usage				
Relative (%)	22.55%	10.30%	-0.74%	2.57%
Absolute	10.77	5.25	-0.05	2.03
p50 Latency				
Relative (%)	-9.16%	-6.09%	4.89%	-0.72%
Absolute (ms)	-1.369	-0.931	0.807	-0.120
p95 Latency				
Relative (%)	-14.64%	-6.05%	4.46%	-1.13%
Absolute (ms)	-2.770	-1.131	0.983	-0.251
p99 Latency				
Relative (%)	-12.29%	-8.17%	6.78%	-4.69%
Absolute (ms)	-2.516	-2.350	1.757	-1.182

The last plot of the Cumulative Distribution Function (CDF) of both the desired replicas and the actual deployed pods shown below indicates one more metric of optimal resource allocation. This performance gives a strong indication that with more training the agent not only can allocate less Pods, but also in the right direction regarding the CPU and memory usage. The rest of the agents, PPO with 1300 episode training, A2C and TRPO achieve unsatisfactory performace, except PPO_1300 which shows the right direction to converging performance. The difference in the distributions shows how close the agents are to achieving optimal performance over the entire evaluation phase.

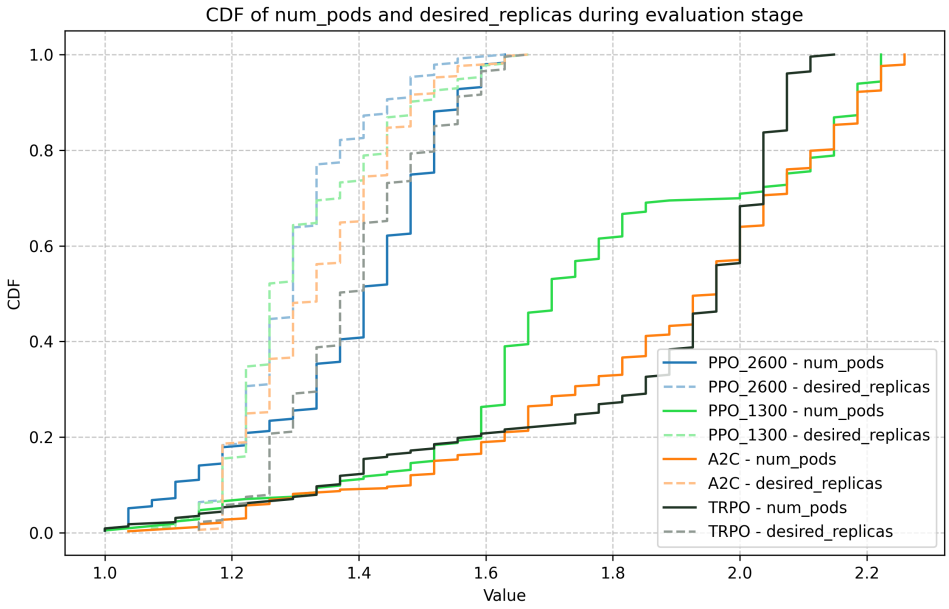


Figure 5.9. *CDF over the number of pods deployed and the desired replicas.*

As demonstrated in the above plot, the agents trained for 1300 episodes do not have the ability to adapt their allocation ability optimal to match the distribution of the desired replicas. In contrast, the 2300 episode trained PPO agent has excellent ability to allocate resources in alignment with the desired replicas distribution, illustrating the importance of training in such complex systems.

Part III

Epilogue

Chapter 6

Conclusion

This chapter concludes the thesis and summarizes the observations and key findings, but also difficulties drawn from the pipeline designed with the objective of resource allocation in Kubernetes clusters and finally recommends some ideas that will improve the work done in this thesis.

6.1 Final remarks

The most important conclusion that emerged from this thesis is that building an efficient and accurate reinforcement learning agent that is able to correctly allocate resources dynamically in a Kubernetes cluster is an inherently complex and challenging task. The highly dynamic nature of microservices based environments, with complex and highly dynamic inter-dependencies introduces high variability that must be taken into account by the agent into the training and testing. Also, the online on-policy reinforcement learning algorithms are proven expensive in systems such as Kubernetes in a way that it takes days to train for only a few episodes, indicating that this type of training might not be the best one to count on. This thesis leveraged the critical path extracted from the traces fetched via Jaeger and trained an SVM classifier using that information in order to classify the detected services as culprit services. In extension, the binary output was transformed into a probabilistic output using the calibration of the classifier, allowing the developers to make safer decisions where services were ranked based on their criticality rather than binary classified as culprit or not. In the next stage, the enriched information that was previously extracted, was then added into the input of the RL agent adjusting the resources based not only in the iterative learning, but also to the importance of each service. The end-to-end results indicate that, building an artificial intelligence agent that is able of making impactful decisions requires careful integration of multiple components ranging from the observability tools and its ability to capture the right metrics in the appropriate timing, but also the machine learning and reinforcement learning models to be accurately trained with enough episode runs, with minimal bias and the correct action decisions that will later impact the aforementioned system.

Despite these challenges, the proposed pipeline has shown promising results outperforming the classic baseline of KHPA. The agent with the most amount of trained episodes (PPO_2600) has achieved relative difference of 22.55% less Pod usage in return for 9.16%

more median latency or 12.29% more 99^{th} percentile of latency in the Kubernetes cluster. This translates into 10.77 less Pods in return for 1.369 (ms) more median latency or 2.516 (ms) more 99^{th} percentile of latency throughout the evaluation stage consisting of 81 episodes with dynamic workload going from 50 to 100 RPS. Even the rest of the agents that have been trained for less episodes achieved great results in the entire evaluation stage as shown in table 5.1.

6.2 Future work

Initially, it is very important that the developer ensures that the system is built resiliently and in a correct manner to later try and build an agent to interact with it. For example, building a separate database, like Cassandra or ElasticSearch, to store and hold the data for Jaeger and not in memory as this thesis adopted, can avoid the significant overhead that appeared when trying to fetch the aforementioned data (via http requests) in order to feed the CRISP algorithm to later extract the critical path from these traces. In addition, a Kafka pipeline can be set along with the database ensuring the optimal preprocess of the data in order to train the machine learning models optimally. With that being said, the training time of the agents is expected to be lowered in a critical manner, enabling the developers to not only train the agents for more episodes, but also try different approaches and novelties.

At the same time, a significant improvement can be made in the outlier detection algorithm or as described before the localization of the critical component. As seen in literature, methods such as Density-based methods can be used in order to detect outliers, for example Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is a great algorithm that finds core samples of high density and expands clusters from them, essentially detecting points outside of these clusters and annotating them as outliers.

Another great improvement, which can enhance the overall performance of the entire pipeline, is based solely on the RL agent and its training. As described and shown in this thesis, the training of the RL agent in such complex systems is arguably the most critical aspect in order to match and eventually outperform the baseline strategies of Kubernetes or similar systems but it also represents a major challenge as shown in the evaluation chapter, meaning it take days to train for a low number of episodes. An idea to increase the speed of training is to use Generative Artificial Intelligence models in order to simulate and recreate the distributions of state-action space, artificially creating new training data that are very close to the reality and in this way the training of the RL agent can improve significantly.

Also, a great improvement that can enhance the overall performance is to use a multi-agent reinforcement learning structure by deploying many RL agents that share the observation state of the entire Kubernetes cluster. Instead of relying to a centralized agent to manage all the different resources, a multi-agent setup distributes the workload or the actions responsibility to many agents with the ability to make different actions on different services enhancing the localized decision making, with the observation of the cluster being shared across the agents from actions previously made.

Lastly, a strong improvement that can enhance the overall performance of the entire pipeline is to use Graph Neural Networks (GNN) and more graph theory-based techniques in combination with the reinforcement learning agent. In extension to this work, the critical path or the critical DAG that is extracted from the Jaeger traces or even the entire graph that is composed from the Jaeger traces can be fed into that GNN model, extracting valuable insights. In this way, the understanding of the environment from the agents will be enhanced significantly, improving the resource allocation decisions made from those agents.

References

[1] The Kubernetes Authors. *Kubernetes: Production-Grade Container Orchestration*. https://kubernetes.io/, 2025.

[2] CloudZero. *Horizontal vs. Vertical Scaling: What's the Difference?*, 2024.

[3] Jaeger Authors. *Jaeger: Open Source, End-to-End Distributed Tracing*, 2025.

[4] Prometheus Authors. *Prometheus: Monitoring System & Time Series Database*, 2025.

[5] Richard S. Sutton και Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2ndη έκδοση, 2018.

[6] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare και Joelle Pineau. *An Introduction to Deep Reinforcement Learning*. *Foundations and Trends in Machine Learning*, 11:219–354, 2018.

[7] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood και Milind Chabbi. *CRISP: Critical Path Analysis of Large-Scale Microservice Architectures*. *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, Carlsbad, CA, 2022. USENIX Association.

[8] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla και Christina Delimitrou. *An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems*. *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019.

[9] Devopedia. *Docker*. https://devopedia.org/docker, 2020.

[10] Mohammad Mustafa Taye. *Understanding of Machine Learning with Deep Learning: Architectures, Workflow, Applications and Future Directions*, 2023.

[11] Sumari Putra, Syed Saqib Jamal και Laith Abualigah. *A Novel Deep Learning Pipeline Architecture based on CNN to Detect Covid-19 in Chest X-ray Images*. *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, Vol. 12:2001–2011, 2021.

[12] Juan Terven, Diana Margarita Cordova-Esparza, Alfonzo Ramirez-Pedraza και Edgar Chávez Urbiola. *Loss Functions and Metrics in Deep Learning: A Review*. *arXiv preprint arXiv:2307.02694*, 2023.

[13] Yingjie Tian και Yuqi Zhang. *A Comprehensive Survey on Regularization Strategies in Machine Learning*. *Information Fusion*, 80:146–166, 2022.

[14] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin και Larisa Safina. *Microservices: yesterday, today, and tomorrow*. *Present and ulterior software engineering*, σελίδες 195–216, 2017.

[15] Inc. Amazon Web Services. *Amazon Web Services (AWS) - Cloud Computing Services*, 2024.

[16] Microsoft Corporation. *Microsoft Azure - Cloud Computing Services*, 2024.

[17] Google LLC. *Google Cloud Platform (GCP) - Cloud Services*, 2024.

[18] SoundCloud. *SoundCloud: Discover, Stream, and Share Music*, 2025.

[19] Cloud Native Computing Foundation. *Cloud Native Computing Foundation (CNCF)*, 2025.

[20] Sara Brown. *Machine Learning, Explained*. *MIT Sloan School of Management*, 2021.

[21] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[22] Daniel Jurafsky και James H. Martin. *Chapter 7: Language Modeling*. *Speech and Language Processing (3rd ed.)*. Draft, Stanford University, 2023.

[23] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan και Pieter Abbeel. *Trust Region Policy Optimization*, 2017.

[24] Jonathon Shlens. *Notes on Kullback-Leibler Divergence and Likelihood*, 2014.

[25] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford και Oleg Klimov. *Proximal Policy Optimization Algorithms*, 2017.

[26] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver και Koray Kavukcuoglu. *Asynchronous Methods for Deep Reinforcement Learning*, 2016.

[27] Gil Tene. *wrk2: A constant throughput, correct latency recording variant of wrk*. https://github.com/giltene/wrk2, 2014.

[28] Apache Software Foundation. *Apache Thrift*. https://github.com/apache/thrift, 2025. GitHub repository.

[29] Google. *gRPC: A high-performance, open-source universal RPC framework*. https://grpc.io/, 2024.

[30] Mark Muldoon. *Lecture 14: Critical Path Analysis*. University of Manchester lecture notes, online PDF.

[31] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk και Ravishankar K. Iyer. *FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices*. *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 2020.

[32] Jacob Benesty, Jingdong Chen, Yiteng Huang και Israel Cohen. *Pearson Correlation Coefficient*, σελίδες 1–4. Springer, 2009.

[33] Rob Eisinga, Manfredte Grotenhuis και Ben Pelzer. *The reliability of a two-item scale: Pearson, Cronbach, or Spearman-Brown?* *International Journal of Public Health*, 58:637–642, 2013.

[34] Chaos Mesh. *Chaos Mesh: A Chaos Engineering Platform for Kubernetes*. https://chaos-mesh.org/, n.d.

[35] John Platt. *Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods*. *Adv. Large Margin Classif.*, 10, 2000.

[36] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot και E. Duchesnay. *Scikit-learn: Machine Learning in Python*. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[37] Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U. Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Hannah Tan και Omar G. Younis. *Gymnasium: A Standard Interface for Reinforcement Learning Environments*, 2024.

[38] José Santos, Tim Wauters, Bruno Volckaert και Filip De Turck. *gym-hpa: Efficient Auto-Scaling via Reinforcement Learning for Complex Microservice-based Applications in Kubernetes*. *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*, 2023.

[39] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus και Noah Dormann. *Stable-Baselines3: Reliable Reinforcement Learning Implementations*. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[40] Antonin Raffin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto και Noah Dormann. *Stable Baselines3*. https://github.com/DLR-RM/stable-baselines3, 2019.

[41] Kubernetes Authors. *Horizontal Pod Autoscaler*. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[42] Kubernetes Authors. *Vertical Pod Autoscaler*. https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler.

[43] Chenhao Qu, Rodrigo N. Calheiros και Rajkumar Buyya. *Auto-Scaling Web Applications in Clouds*. *ACM Computing Surveys*, 51:1–33, 2019.

[44] Yanqi Zhang, Zhuangzhuang Zhou, Sameh Elnikety και Christina Delimitrou. *Ursa: Lightweight Resource Management for Cloud-Native Microservices*. *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, σελίδες 954–969. IEEE, 2024.

[45] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G. Edward Suh και Christina Delimitrou. *Sinan: ML-Based and QoS-Aware Resource Management for Cloud Microservices*. *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, σελίδες 167–181, New York, NY, USA, 2021. Association for Computing Machinery.

[46] Lei Liu, Xinglei Dou και Yuetao Chen. *Intelligent Resource Scheduling for Co-located Latency-critical Services: A Multi-Model Collaborative Learning Approach*. *21st USENIX Conference on File and Storage Technologies (FAST 23)*, σελίδες 153–166, Santa Clara, CA, 2023. USENIX Association.

[47] Yu Gan, Meghna Pancholi, Siyuan Hu, Dailun Cheng, Yuan He και Christina Delimitrou. *Seer: Leveraging Big Data to Navigate the Increasing Complexity of Cloud Debugging*. *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, Boston, MA, 2018. USENIX Association.

[48] Google Cloud. *What is Microservices Architecture?* https://cloud.google.com/learn/what-is-microservices-architecture, 2025.

[49] Mozilla Developer Network. *HTTP: Hypertext Transfer Protocol*. https://developer.mozilla.org/en-US/docs/Web/HTTP, 2025.

[50] Docker, Inc. *Docker: Accelerate how you build, share, and run applications*. https://www.docker.com/, 2025.

[51] IBM. *CAP Theorem Explained*, 2024.

[52] OpenTracing Authors. *OpenTracing: Specification and Libraries for Distributed Tracing*, 2025.

[53] OpenTelemetry Authors. *OpenTelemetry: Observability Framework for Cloud-Native Software*, 2025.

[54] IBM. *User Datagram Protocol*, 2025.

[55] Apache Software Foundation. *Apache Cassandra: The Highly Scalable NoSQL Database*, 2025.

[56] Elastic NV. *Elasticsearch: Distributed, RESTful Search and Analytics Engine*, 2025.

[57] Apache Software Foundation. *Apache Kafka: A Distributed Streaming Platform*, 2025.

[58] Xue Ying. *An Overview of Overfitting and its Solutions*. *Journal of Physics: Conference Series*, 1168:022022, 2019.

[59] Qiong Liu και Ying Wu. *Supervised Learning*. *Encyclopedia of Machine Learning*, 2012.

[60] Samreen Naeem, Aqib Ali, Sania Anam και Munawar Ahmed. *An Unsupervised Machine Learning Algorithms: Comprehensive Review*. *IJCDS Journal*, 13:911–921, 2023.

[61] Richard Bellman. *A Markovian Decision Process*. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957.

[62] Beakcheol Jang, Myeonghwi Kim, Gaspard Harerimana και Jong Wook Kim. *Q-Learning Algorithms: A Comprehensive Classification and Applications*. *IEEE Access*, 7:133653–133667, 2019.

[63] Sreehari Rammohan, Shangqun Yu, Bowen He, Eric Hsiung, Eric Rosen, Stefanie Tellex και George Konidaris. *Value-Based Reinforcement Learning for Continuous Control Robotic Manipulation in Multi-Task Sparse Reward Settings*, 2021.

[64] Richard E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[65] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra και Martin A. Riedmiller. *Playing Atari with Deep Reinforcement Learning*. *CoRR*, abs/1312.5602, 2013.

[66] Roberto Ierusalimschy, Luiz Henriquede Figueiredo και Waldemar Celes. *Lua: A Lightweight Multi-paradigm Programming Language*. https://www.lua.org/, 1993.

[67] Amro Al Said Ahmad, Lamis F. Al-Qora'n και Ahmad Zayed. *Exploring the impact of chaos engineering with various user loads on cloud native applications: an exploratory empirical study*. *Computing*, 106:2389–2425, 2024.