Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Επικοινωνιών, Ηλεκτρονικής
Και Συστημάτων Πληροφορικής

Δυναμική Διαχείριση Πόρων σε Kubernetes

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΣΠΥΡΙΔΩΝ Μ. ΚΡΗΤΙΚΟΣ

Επιβλέπων : Συμεών Παπαβασιλείου
Καθηγητής Ε.Μ.Π.

Αθήνα, Σεπτέμβριος 2022

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομεας Επικοινωνιων, Ηλεκτρονικης
Και Συστηματων Πληροφορικης

Δυναμική Διαχείριση Πόρων σε Kubernetes

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΣΠΥΡΙΔΩΝ Μ. ΚΡΗΤΙΚΟΣ

Επιβλέπων : Συμεών Παπαβασιλείου
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 28η Σεπτεμβρίου 2022.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
....	....	....
Συμεών Παπαβασιλείου	Ιωάννα Ρουσσάκη	Θεοδώρα Βαρβαρίγου
Καθηγητής Ε.Μ.Π.	Καθηγήτρια Ε.Μ.Π.	Καθηγήτρια Ε.Μ.Π.

Αθήνα, Σεπτέμβριος 2022

(Υπογραφή)

....

Σπυρίδων Μ. Κρητικός

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Σπυρίδων Κρητικός, 2022
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα.
Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Η ραγδαία ανάπτυξη εφαρμογών υπολογιστικού νέφους αποτελεί μια καινοτομία που ασπάζεται από τον εταιρικό κόσμο όλο και περισσότερο, αφήνοντας πίσω τον παραδοσιακό χαρακτήρα ανάπτυξης εφαρμογών. Σε παράλληλο βαθμό αναπτύσσονται και εφαρμογές που υιοθετούν το μοντέλο του υπολογισμού στα άκρα του δικτύου, το οποίο είναι ένα κατανεμημένο υπολογιστικό πλαίσιο που φέρνει τις επιχειρησιακές εφαρμογές πιο κοντά σε πηγές δεδομένων. Οι επιχειρήσεις που χρησιμοποιούν τις παραπάνω εφαρμογές έχουν ισχυρά πλεονεκτήματα , όπως βελτιωμένους χρόνους απόκρισης. Οι μικροϋπηρεσίες που συνιστούν τις εν λόγω εφαρμογές βασίζονται σε containerized αρχιτεκτονικές (αρχιτεκτονικές κιβωτίων). Τα containers διαμορφώνουν τη νέα εποχή των εφαρμογών υπολογιστικού νέφους λόγω των βασικών οφελών τους, όπως το μικρό βάρος, η ελάχιστη κατανάλωση πόρων για την εκτέλεση μιας εφαρμογής που μειώνει το κόστος και η εύκολη και ταχεία κλιμάκωση ανάλογα με τις απαιτήσεις του φόρτου εργασίας. Ωστόσο, οι εφαρμογές νέφους που βασίζονται σε κιβώτια απαιτούν εξελιγμένες μεθόδους αυτόματης κλιμάκωσης, ικανές για να δεσμεύουν και να αποδεσμεύουν αυτόματα και έγκαιρα πόρους νέφους χωρίς την ανθρώπινη παρέμβαση, ανταποκρινόμενες στις δυναμικές διακυμάνσεις του φόρτου εργασίας. Το Kubernetes, ο ενορχηστρωτής κιβωτίων για εφαρμογές που αναπτύσσονται στο υπολογιστικό νέφος, προσφέρει αυτόματη κλιμάκωση για τον πάροχο της εφαρμογής προκειμένου να ανταποκρίνεται στη συνεχώς μεταβαλλόμενη ένταση της ζήτησης. Αυτή η δυνατότητα αυτόματης κλιμάκωσης μπορεί να προσαρμοστεί με ένα σύνολο παραμέτρων, αλλά αυτές οι παράμετροι διαχείρισης είναι στατικές, ενώ η δυναμική των εισερχόμενων αιτημάτων ιστού συχνά αλλάζει. Στη παρούσα διπλωματική θέσαμε ως απώτερο στόχο να κάνουμε τη διαχείριση των εφαρμογών που βασίζονται στο νέφος καλύτερη και αποτελεσματικότερη. Προτείνουμε μια λογική κατανομής πόρων σε περιβάλλον Kubernetes, χρησιμοποιώντας τεχνικές μηχανικής μάθησης, που καθιστά τις αποφάσεις αυτόματης κλιμάκωσης κατάλληλες για τον χειρισμό της πραγματικής μεταβλητότητας των εισερχόμενων αιτημάτων.

Λέξεις Κλειδιά

Υπολογισμός στα άκρα δικτύου, Kubernetes, αυτόματη κλιμάκωση, μηχανική μάθηση, αποθηκευτικό νέφος

Abstract

The rapid development of cloud computing applications is an innovation that is increasingly embraced by the corporate world, leaving behind the traditional nature of application development. Furthermore, applications are being developed that adopt the model of edge computing, which is a distributed computing framework that brings business applications closer to data sources. Enterprises using these applications have strong advantages, such as improved response times. The microservices that consist these applications are based on containerized architectures. Containers are shaping the new era of cloud computing applications due to their key benefits such as light weight, minimal resource consumption to run an application which reduces costs, and easy and rapid scaling according to workload requirements. However, container-based cloud applications require sophisticated automatic scaling methods capable of automatically and timely provisioning of the cloud resources, without human intervention, in response to dynamic workload fluctuations. Kubernetes, the container orchestrator for applications deployed in cloud computing, offers automatic scaling for the application provider to respond to ever-changing demand intensity. This automatic scaling capability can be customized with a set of parameters, but these management parameters are static, while the dynamics of incoming web requests often change. In this thesis we set the ultimate goal of making the management of cloud-based applications better and more efficient. We propose a resource allocation logic in a Kubernetes environment, using machine learning techniques, that makes automatic scaling decisions suitable for handling the real variability of incoming requests.

Key Words

edge computing, Kubernetes, autoscaling, machine learning, cloud storage

Ευχαριστίες

Η εκπόνηση αυτής της διπλωματικής εργασίας έγινε υπό την επίβλεψη του καθηγητή του Ε.Μ.Π. κ. Συμεών Παπαβασιλείου, τον οποίο θα ήθελα να ευχαριστήσω θερμά για την ευκαιρία που μου έδωσε να μελετήσω ένα τόσο ενδιαφέρον θέμα και να διευρύνω τις γνώσεις μου γύρω από αυτό.

Ακόμα θα ήθελα να ευχαριστήσω τον μεταδιδάκτορα Δημήτρη Δεχουνιώτη και ιδιαιτέρως τους υποψήφιους διδάκτορες Δημήτρη Σπαθαράκη και Ιωάννη Δημολίτσα, για τη στήριξη τους σε κάθε βήμα της εργασίας προσφέροντας ουσιαστική βοήθεια οποτεδήποτε χρειαζόταν.

Τέλος θα ήθελα να ευχαριστήσω την οικογένεια και τους φίλους μου για την αμέριστη συμπαράσταση και τη συνεχή στήριξή τους.

Περιεχόμενα

Περίληψη 6

Abstract 8

Ευχαριστίες 10

Περιεχόμενα 12

Κατάλογος Σχημάτων 13

Κατάλογος Πινάκων 14

Κεφάλαιο 1 : Εισαγωγή 15

1.1 Διαδίκτυο των πραγμάτων και μεταφόρτωση υπολογισμών 15
1.2 Εφαρμογές υπολογιστικού νέφους 15
1.3 Ανάγκη για edge computing 16
1.4 Αντικείμενο της Διπλωματικής Εργασίας 17
1.5 Δομή της εργασίας 18

Κεφάλαιο 2 : Υφιστάμενα μοντέλα και τεχνικές δυναμικής κατανομής πόρων 19

2.1 Κανόνες που βασίζονται σε τιμές-κατώφλια 19
2.2 Θεωρία Ελέγχου 20
2.3 Ανάλυση χρονοσειρών 21
2.4 Reinfocement Learning 21
2.5 Μοντέλο ουράς αναμονής 22
2.6 Performance prediction (Πρόβλεψη Απόδοσης) 23
2.7 Demand forecast (Πρόβλεψη Ζήτησης) 24

Κεφάλαιο 3 : Βασικές έννοιες από εργαλεία που χρησιμοποιήθηκαν 24

3.1 Βασικά στοιχεία του Kubernetes 24
3.2 Scalability 27
3.3 Custom Pod Autoscaler 30
3.4 Εργαλεία παρακολούθησης και οπτικοποίησης των μετρικών 32
3.5 Γλώσσα προγραμματισμού 33
3.6 Αλγόριθμοι επιβλεπόμενης μηχανικής μάθησης 34

Κεφάλαιο 4 : Προτεινόμενη Αρχιτεκτονική 38

4.1 Πιλοτική Υλοποίηση εφαρμογής 38

Κεφάλαιο 5 : Πειραματική Αξιολόγηση 43

5.1 Πειραματικές Συνθήκες 43
5.2 Παρουσίαση αποτελεσμάτων 46

Κεφάλαιο 6 : Συμπεράσματα και Μελλοντικές Προεκτάσεις **51**
6.1 Συμπεράσματα 51
6.2 Μελλοντικές Προεκτάσεις 51

Βιβλιογραφία **52**

Κατάλογος Σχημάτων

Σχήμα 1: Σχέση edge computing αρχιτεκτονικών και IoT **17**
Σχήμα 2: Δομή ενός συστήματος ελέγχου με ανατροφοδότηση **20**
Σχήμα 3: Απλή διαδικασία ενισχυτικής μάθησης **22**
Σχήμα 4: Διάγραμμα δικτύου που χρησιμοποιεί την ουρά αναμονής **22**
Σχήμα 5: Απλή αρχιτεκτονική ενός cluster του Kubernetes **26**
Σχήμα 6: Δομή λειτουργίας του HPA **29**
Σχήμα 7: Ένα απλό configuration file του CPA **31**
Σχήμα 8: Παράδειγμα αρχείου metric.py για τη συλλογή μετρικών **31**
Σχήμα 9: Παράδειγμα αρχείου evaluate.py για τη λογική κλιμάκωσης **32**
Σχήμα 10: Απλό παράδειγμα ενός Dockerfile αρχείου **32**
Σχήμα 11: Λειτουργία ενός Prometheus server **33**
Σχήμα 12: Συνάρτηση κατηγοριοποίησης των δειγμάτων σε περιοχές **40**
Σχήμα 13: Προσομοίωση τελικής αρχιτεκτονικής του συστήματος **42**
Σχήμα 14: Το deployment.yaml αρχείο του ενός flavor της εφαρμογής **44**
Σχήμα 15: Το service.yaml αρχείο του ενός flavor της εφαρμογής **45**
Σχήμα 16: Ρυθμός αιτημάτων στο πείραμα με το Random Forest **46**
Σχήμα 17: Ρυθμός αιτημάτων στο πείραμα με το HPA **47**
Σχήμα 18: Ρυθμός μεταβολής της κατανάλωσης CPU **47**
Σχήμα 19: Απόκριση χρόνου εξυπηρέτησης του συστήματος **48**

Σχήμα 20: Ρυθμός μεταβολής του αριθμού των replicas κάθε flavor **49**

Κατάλογος Πινάκων

Πίνακας 1: Ακρίβεια των αλγορίθμων κατά την μάθηση **41**

Πίνακας 2: Μέσες τιμές μετρικών που χρησιμοποιήθηκαν για το small flavor της εφαρμογής **50**

Πίνακας 3: Μέσες τιμές μετρικών που χρησιμοποιήθηκαν για το big flavor της εφαρμογής **50**

Κεφάλαιο 1 : Εισαγωγή

1.1 Διαδίκτυο των πραγμάτων και μεταφόρτωση υπολογισμών

Το Διαδίκτυο των πραγμάτων ή Ίντερνετ των πραγμάτων (IoT) (Internet of things) συνιστά ένα δίκτυο επικοινωνίας διαφόρων συσκευών καθώς και κάθε αντικειμένου που ενσωματώνει ηλεκτρονικά μέσα, ώστε να γίνεται εφικτή η σύνδεση και η ανταλλαγή δεδομένων. Ουσιαστικά, η λογική του διαδικτύου των πραγμάτων είναι είτε η σύνδεση όλων των ηλεκτρονικών συσκευών μεταξύ τους ή η δυνατότητα σύνδεσής τους στο διαδίκτυο. Η έννοια "πράγματα" (Things) δεν είναι αυστηρά συνδεδεμένη με ορισμένα προϊόντα. Αναφέρεται σε μία ευρεία ποικιλία συσκευών εντελώς διαφορετικά μεταξύ τους, όπως για παράδειγμα αυτοκίνητα με ενσωματωμένους αισθητήρες, κάμερες, κλιματιστικά, φώτα, συστήματα ασφαλείας, έξυπνα ρολόγια ή ακόμα και αυτοκίνητα των οποίων οι περίπλοκοι αισθητήρες εντοπίζουν αντικείμενα στην πορεία τους. Είναι μερικά από τα πολλά προϊόντα τεχνολογίας των οποίων βασικό χαρακτηριστικό είναι η σύνδεση μεταξύ τους, με απώτερο σκοπό την δυνατότητα του χρήστη να τα ελέγχει από έναν υπολογιστή ή κινητό.

Η μεταφόρτωση υπολογισμών (computation offloading) είναι η μεταφορά εργασιών που χρειάζονται απαιτητικούς υπολογιστικούς πόρους σε έναν ξεχωριστό επεξεργαστή, όπως ένας επιταχυντής υλικού (hardware accelerator) , ή σε μία εξωτερική πλατφόρμα, όπως είναι ένα αποθηκευτικό νέφος (cloud storage). Η μεταφόρτωση σε συν-επεξεργαστή μπορεί να χρησιμοποιηθεί για την επιτάχυνση εφαρμογών όπως είναι η απόδοση εικόνας καθώς και οι μαθηματικοί υπολογισμοί. Η computation offloading σε μια εξωτερική πλατφόρμα μέσω δικτύου, μπορεί να παρέχει υπολογιστική ισχύ και κυρίως να ξεπεράσει τους περιορισμούς υλικού μιας συσκευής, όπως η περιορισμένη υπολογιστική ισχύς, ο λιγοστός αποθηκευτικός χώρος και η ενέργεια.

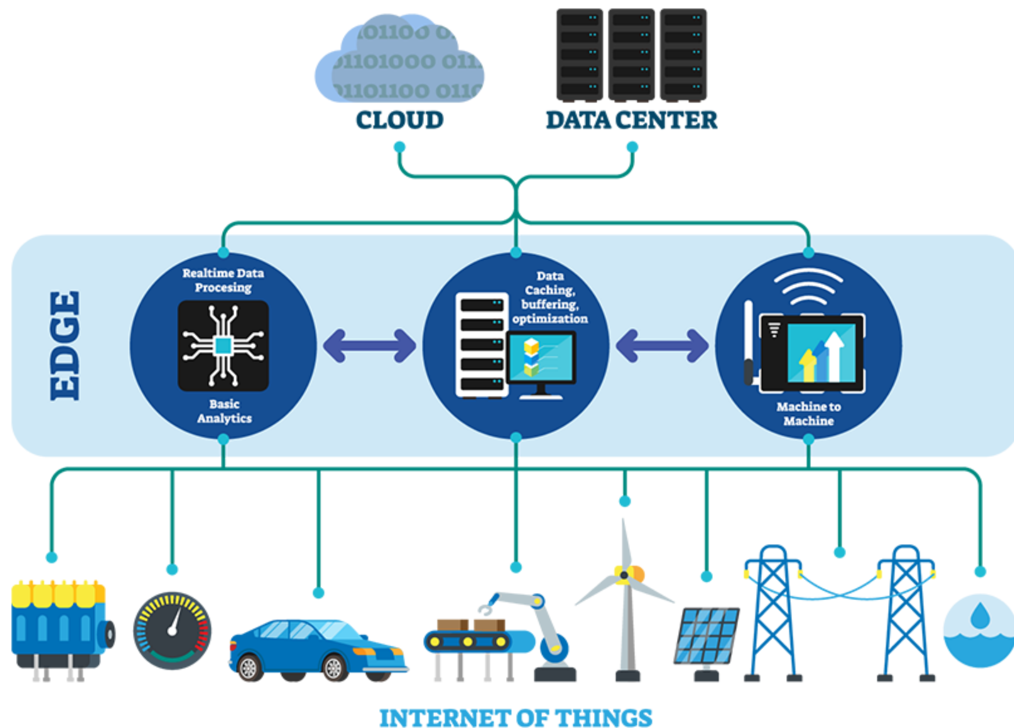
1.2 Εφαρμογές υπολογιστικού νέφους

Το Αποθηκευτικό νέφος (Cloud storage) είναι ένα δικτυακό μοντέλο αποθήκευσης δεδομένων στο οποίο, τα δεδομένα αποθηκεύονται σε απομακρυσμένες δικτυακές τοποθεσίες. Τα δεδομένα αποθηκεύονται σε μεγάλα κέντρα αποθήκευσης δεδομένων (data centers) στα οποία ο χρήστης έχει πρόσβαση μέσω κάποιας δικτυακής διεπαφής (web interface). Ακόμα υπάρχει η δυνατότητα τα δεδομένα αυτά να είναι διασκορπισμένα σε περισσότερους από ένα εξυπηρετητές (servers) [2].
Οι εγγενείς εφαρμογές υπολογιστικού νέφους (cloud-native applications) είναι εφαρμογές που έχουν αναπτυχθεί ειδικά για να τρέχουν σε προγραμματιζόμενες υποδομές υπολογιστικού νέφους (cloud computing). Αυτό σημαίνει ότι έχουν ως στόχο και μπορούν να αξιοποιήσουν τα εγγενή χαρακτηριστικά των cloud native applications [1], όπως την κατανεμημένη διαθεσιμότητα πόρων, την αξιοπιστία που προσφέρει, τη δυναμικότητα

(adaptability - ικανότητα για γρήγορη ανάπτυξη και δοκιμή νέου λογισμικού, την ανθεκτικότητα (resilience - αποδοχή των πιθανών αλλαγών, σφαλμάτων και αποτυχιών και αντιμετώπισή τους λόγω της δυναμικής φύσης) και την επεκτασιμότητα (scalability - η ιδιότητα ενός συστήματος να χειρίζεται έναν αυξανόμενο όγκο εργασίας προσθέτοντας πόρους στο σύστημα), τα οποία είναι σημαντικά για τα συστήματα του Διαδικτύου των πραγμάτων που διαχειρίζονται από το νέφος.

1.3 Ανάγκη για edge computing

Ο υπολογισμός στα άκρα του δικτύου (edge computing) είναι ένα κατανεμημένο υπολογιστικό πλαίσιο που φέρνει τις επιχειρησιακές εφαρμογές πιο κοντά σε πηγές δεδομένων, όπως συσκευές IoT ή τοπικούς διακομιστές άκρων. Αυτή η εγγύτητα μεταξύ των δεδομένων και της πηγής τους μπορεί να προσφέρει ισχυρά επιχειρηματικά οφέλη, όπως ταχύτερες προβλέψεις, βελτιωμένους χρόνους απόκρισης και καλύτερη διαθεσιμότητα του δικτυακού εύρους ζώνης (bandwidth). Η εκρηκτική ανάπτυξη και η αυξανόμενη υπολογιστική ισχύς των συσκευών του IoT έχει οδηγήσει σε πρωτοφανείς όγκους δεδομένων. Και οι όγκοι δεδομένων θα συνεχίσουν να αυξάνονται καθώς τα δίκτυα 5G αυξάνουν τον αριθμό των συνδεδεμένων κινητών συσκευών. Στο παρελθόν, ο στόχος του cloud storage ήταν να αυτοματοποιήσει και να επιταχύνει την καινοτομία χρησιμοποιώντας αξιοποιήσιμες πληροφορίες από τα δεδομένα. Όμως το πρωτοφανές μέγεθος και η πολυπλοκότητα των δεδομένων που δημιουργούνται από τις συνδεδεμένες συσκευές, έχει ξεπεράσει τις δυνατότητες του δικτύου και της υποδομής. Η αποστολή όλων αυτών των δεδομένων που παράγονται από συσκευές σε ένα κεντρικό κέντρο δεδομένων ή στο cloud storage προκαλεί προβλήματα του δικτυακού εύρους ζώνης και χρονικής καθυστέρησης. Οι edge computing αρχιτεκτονικές προσφέρουν μια πιο αποτελεσματική εναλλακτική λύση: τα δεδομένα επεξεργάζονται και αναλύονται πιο κοντά στο σημείο όπου δημιουργούνται. Επειδή τα δεδομένα δεν διασχίζουν ένα δίκτυο, ώστε να μεταφερθούν σε ένα αποθηκευτικό νέφος ή ένα κέντρο δεδομένων, για να υποβληθούν σε επεξεργασία, η χρονική καθυστέρηση μειώνεται σημαντικά. Οι edge computing αρχιτεκτονικές επιτρέπει ταχύτερη και πιο ολοκληρωμένη ανάλυση δεδομένων, δημιουργώντας την ευκαιρία για καλύτερες πληροφορίες, ταχύτερους χρόνους απόκρισης και βελτιωμένες εμπειρίες πελατών. Όλα τα παραπάνω επιτυγχάνονται με την υλοποίηση ενός έξυπνου μηχανισμού για την κατανομή υπολογιστικών πόρων ώστε να ανταποκρίνονται κατάλληλα στις ανάγκες των εισερχόμενων αιτημάτων κάποιας εφαρμογής.



Σχήμα 1: Σχέση edge computing αρχιτεκτονικών και IoT

1.4 Αντικείμενο της Διπλωματικής Εργασίας

Στη συγκεκριμένη εργασία αναπτύξαμε μια εφαρμογή αναγνώρισης εικόνας (image recognition) που κάνει χρήση ενός απλού machine learning αλγορίθμου , και παρακολουθήσαμε διάφορες μετρικές που παράγει κατά την χρήση της. Σκοπός της εργασίας ήταν η υλοποίηση ενός αποδοτικού αλγορίθμου για δυναμική διαχείριση πόρων της εφαρμογής. Για να πετύχουμε τα παραπάνω χρησιμοποιούμε το εργαλείο ενορχήστρωσης πόρων "Kubernetes", που είναι μια φορητή, επεκτάσιμη πλατφόρμα ανοικτού κώδικα για τη διαχείριση containerized εφαρμογών και υπηρεσιών. Κάθε μικροϋπηρεσία, αναπτύσσεται αυτόνομα με τη γλώσσα και το προγραμματιστικό πλαίσιο (framework) που ταιριάζει καλύτερα στη λειτουργικότητά της, πακετάρεται σε ένα container με τις εξαρτήσεις (dependencies), τον κώδικα και τα δεδομένα της και απομονώνεται σε αυτό με όλο το περιβάλλον της (runtime environment). Με αυτό τον τρόπο παρέχεται φορητότητα (portability) και συνέπεια (consistency) στην εκτέλεσή του σε διαφορετικά περιβάλλοντα (υπολογιστές, παραδοσιακές υποδομές ή υποδομές υπολογιστικού νέφους).

Στο περιβάλλον αυτό κάναμε χρήση εργαλείων για να παρακολουθούμε το φόρτο εργασίας που έχει η εφαρμογή στο σύστημά μας. Συγκεκριμένα αξιοποιήσαμε το σύστημα παρακολούθησης μετρικών "Prometheus" η οποία κατέγραφε τις μετρήσεις της εφαρμογής σε πραγματικό χρόνο καθώς και την εφαρμογή "Grafana" με την οποία οπτικοποιήσαμε τις μετρικές που λάβαμε από τον Prometheus σε κατάλληλα γραφήματα που έπειτα αναλύσαμε.

Έπειτα αναπτύξαμε μία λογική κατανομής πόρων στην εφαρμογή χρησιμοποιώντας μαθηματικούς αλγορίθμους και τεχνικές μηχανικής μάθησης (machine learning) ώστε η εφαρμογή μας να μεταβάλλει τον αριθμό των διαθέσιμων πόρων ανάλογα με τις ανάγκες της ίδιας της εφαρμογής, κάθε δοσμένη χρονική στιγμή.

1.5 Δομή της εργασίας

Στο Κεφάλαιο 2 της παρούσας εργασίας παρατίθενται και εξηγούνται διάφοροι μέθοδοι κατανομής πόρων (resource allocation),λαμβάνοντας υπόψη τις πιο σημαντικές δουλειές στην βιβλιογραφία. Έπειτα, στο Κεφάλαιο 3 δίνεται μια ειδικότερη περιγραφή του αντικειμένου της συγκεκριμένης διπλωματικής καθώς και ο μηχανισμός κατανομής πόρων που θα χρησιμοποιηθεί. Παρατίθενται επίσης τα εργαλεία ανοιχτού κώδικα που χρησιμοποιήθηκαν για την ανάπτυξη εφαρμογών καθώς και εργαλεία ενορχήστρωσης και παρακολούθησης που βοήθησαν. Έχοντας αυτό το υπόβαθρο, στο Κεφάλαιο 4 παρουσιάζεται η υλοποίηση του προβλήματος καθώς και η αρχιτεκτονική που ακολουθήσαμε για να το πετύχουμε. Στο Κεφάλαιο 5 περιγράφεται η ανάπτυξη της πιλοτικής εφαρμογής καθώς και διάφορα εργαλεία που χρησιμοποιήθηκαν για την παρακολούθηση της. Για τη συλλογή δεδομένων εκτελείται το πειραματικό μέρος της εργασίας και στη συνέχεια, έπειτα από την κατάλληλη ανάλυση και επεξεργασία, παρουσιάζονται τα αποτελέσματα για κάθε πείραμα. Στο Κεφάλαιο 6 , εξάγονται τα βασικά πορίσματα της εργασίας και παράλληλα προτείνονται ιδέες για την εξέλιξή της.

Κεφάλαιο 2 : Υφιστάμενα μοντέλα και τεχνικές δυναμικής κατανομής πόρων

Στόχος των προσεγγίσεων autoscaling (αυτόματης κλιμάκωσης) για την κατανομή πόρων είναι η απόκτηση και η απελευθέρωση πόρων δυναμικά, διατηρώντας παράλληλα ένα αποδεκτό QoS (Quality of Service) (Ποιότητα της υπηρεσίας). Η διαδικασία autoscaling συνήθως αναπαρίσταται και υλοποιείται από έναν MAPE-K (Monitor, Analyze, Plan and Execute phases over a Knowledge base) βρόχο ελέγχου [3]. Ένας autoscaler σχεδιάζεται με συγκεκριμένο στόχο, βασιζόμενος σε ικανότητες κλιμάκωσης που προσφέρονται από τους παρόχους νέφους ή εστιάζοντας στη δομή της εφαρμογής-στόχου. Μπορούμε να ταξινομήσουμε τις προσεγγίσεις autoscaling σε τρεις ευρύτερες κατηγορίες : Reactive Scaling , Proactive Scaling , Predictive Scaling. Οι Reactive Scaling λογικές, απλώς παρακολουθούν τις εφαρμογές και προσαρμόζουν τον αριθμό των virtualized resources (εικονικών πόρων) ώστε να διατηρεί τη βέλτιστη απόδοση με το ελάχιστο δυνατό κόστος. Μπορεί να είναι ιδιαίτερα χρήσιμη προσέγγιση σε εκείνες τις εφαρμογές στις οποίες η κίνηση αυξομειώνεται συνεχώς χωρίς κάποιο μοτίβο. Οι Proactive Scaling λογικές επιτρέπουν να κλιμακώνεται η εφαρμογή με βάση το εκτιμώμενο φορτίο που θα εμφανιστεί στο μέλλον. Είναι χρήσιμη προσέγγιση για τις εφαρμογές εκείνες όπου είναι γνωστό πότε θα υπάρξει αυξημένο φορτίο και πότε θα υπάρξει μειωμένο φορτίο. Η Predictive Scaling χρησιμοποιεί τεχνικές μηχανικής μάθησης για να προβλέψει τη χρήση της εφαρμογής στο μέλλον και έτσι οι αλλαγές γίνονται ανάλογα. Συλλέγει μεγάλο αριθμό δεδομένων από την χρήση της εφαρμογής και χρησιμοποιεί καλά εκπαιδευμένα μοντέλα μηχανικής μάθησης για να προβλέψει την αναμενόμενη κίνηση. Είναι χρήσιμη προσέγγιση για τις εφαρμογές εκείνες όπου ο φόρτος εργασίας έχει κυκλικό μοτίβο. Όλες οι τεχνικές autoscaling βασίζονται σε κάποιο θεωρητικό μοντέλο και με αυτό το τρόπο θα τις κατηγοριοποιήσουμε, αναφέροντας τις πιο διαδεδομένες προσεγγίσεις autoscaling.

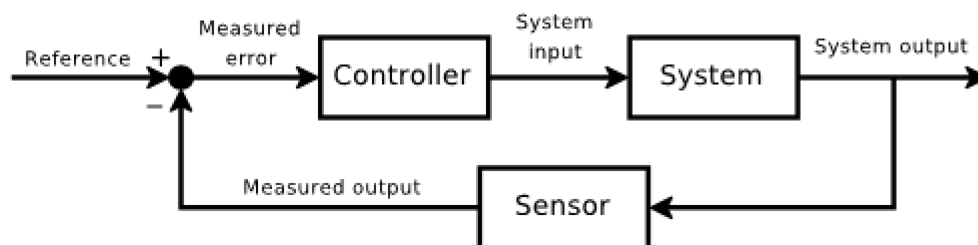
2.1 Κανόνες που βασίζονται σε τιμές-κατώφλια

Οι κανόνες που βασίζονται σε τιμές-κατώφλια είναι η πιο δημοφιλής προσέγγιση που προσφέρεται από πολλές πλατφόρμες, όπως η Amazon ή το OpenStack 3. Οι συνθήκες και οι κανόνες στις προσεγγίσεις που βασίζονται σε τιμές-κατώφλια μπορούν να οριστούν με βάση μία ή περισσότερες μετρικές επιδόσεων, όπως για παράδειγμα το CPU utilization, ο μέσος χρόνος απόκρισης ή ο ρυθμός αιτημάτων. Οι Dutreilh κ.ά. [4] διερευνούν την οριζόντια αυτόματη κλιμάκωση χρησιμοποιώντας τιμές-κατώφλια και τεχνικές reinforcement learning. Οι Han κ.ά. [5], περιγράφουν μια προσέγγιση η οποία λειτουργεί με κλιμάκωση σε επίπεδο πόρων επιπλέον της κλιμάκωσης σε επίπεδο VM (Virtual Machine) (Εικονικό Μηχάνημα), προκειμένου να βελτιώσει τη χρήση των πόρων, ενώ ταυτόχρονα μειώνει το κόστος του παρόχου νέφους. Οι Hasan κ.ά. [6] επεκτείνουν τις δύο τυπικές τιμές-κατωφλίου και προσθέτουν δύο επίπεδα παραμέτρων-κατωφλίου για τη λήψη αποφάσεων κλιμάκωσης. Οι Chieu κ.ά. [7] προτείνουν μια απλή στρατηγική για τη δυναμική κλιμάκωση ενός είδους διαδικτυακών εφαρμογών, με βάση τον αριθμό των ενεργών συνεδριών και την κλιμάκωση του αριθμού των VMs εάν όλες οι περιπτώσεις

έχουν ενεργές συνεδρίες που υπερβαίνουν συγκεκριμένα κατώτατα όρια. Το κύριο πλεονέκτημα των προσεγγίσεων autoscaling βάση τιμών-κατωφλίου είναι η απλότητα που τις καθιστά εύκολες στη χρήση σε παρόχους υπηρεσιών νέφους και επίσης εύκολη στη ρύθμιση από τους πελάτες.

2.2 Θεωρία Ελέγχου

Η θεωρία ελέγχου ασχολείται με τον επηρεασμό της συμπεριφοράς δυναμικών συστημάτων με την παρακολούθηση της εξόδου και τη σύγκρισή της με τιμές αναφοράς. Χρησιμοποιώντας την ανατροφοδότηση της εισόδου του συστήματος, που ορίζεται ως η διαφορά μεταξύ του πραγματικού και του επιθυμητού επιπέδου εξόδου, ο ελεγκτής προσπαθεί να ευθυγραμμίσει την πραγματική έξοδο με την τιμή αναφοράς.



Σχήμα 2: Δομή ενός συστήματος ελέγχου με ανατροφοδότηση

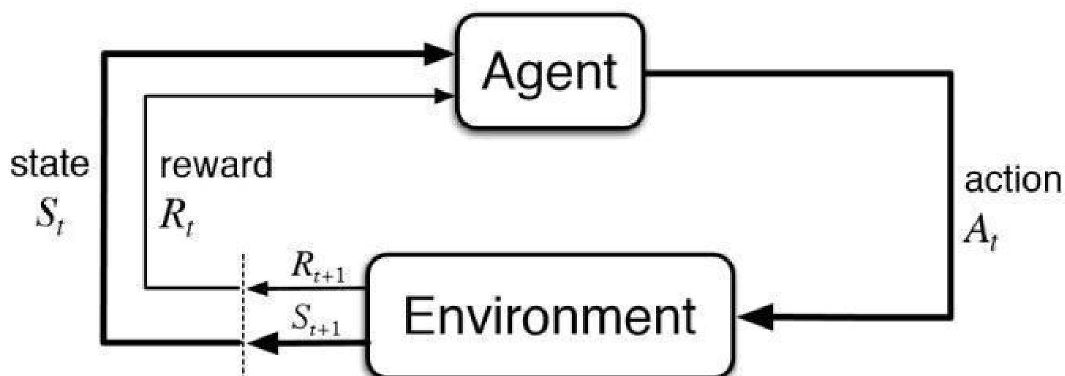
Για το autoscaling, η παράμετρος αναφοράς, δηλαδή ένα αντικείμενο που πρέπει να ελέγχεται, είναι η στοχευμένη τιμή SLA (Service Level Agreement) (Προκαθορισμένο επίπεδο υπηρεσιών) [8]. Το σύστημα είναι η πλατφόρμα-στόχος και η έξοδος του συστήματος είναι παράμετροι για την αξιολόγηση απόδοσης του συστήματος, όπως για παράδειγμα ο μέσος χρόνος απόκρισης ή το CPU utilization. Οι Zhu και Agrawal [9] παρουσιάζουν ένα πλαίσιο που χρησιμοποιεί έναν Proportional-Integral (PI) ελεγκτή, σε συνδυασμό με τεχνικές reinforcement learning, προκειμένου να ελαχιστοποιηθεί το κόστος της εφαρμογής. Οι Ali-Eldin κ.ά. [10],[11] προτείνουν δύο προσαρμοστικούς υβριδικούς ελεγκτές reactive/proactive προκειμένου να υποστηριχθεί η ελαστικότητα των υπηρεσιών με χρήση της θεωρίας της ουράς αναμονής για την εκτίμηση του μελλοντικού φορτίου. Οι Padala κ.ά. [12] προτείνουν έναν ανατροφοδοτούμενο σύστημα ελέγχου πόρων που προσαρμόζεται αυτόματα σε δυναμικές μεταβολές του φόρτου εργασίας για την ικανοποίηση των στόχων του επιπέδου εξυπηρέτησης. Χρησιμοποιούν έναν online εκτιμητή μοντέλου για τη δυναμική διατήρηση της σχέσης μεταξύ των εφαρμογών και των πόρων καθώς και ένα ελεγκτή δύο επιπέδων με πολλαπλές εισόδους και πολλαπλές εξόδους που κατανέμει δυναμικά τους πόρους στις εφαρμογές. Οι Kalyvianaki κ.ά. [13] ενσωματώνουν ένα φίλτρο Kalman σε ελεγκτές ανατροφοδότησης που ανιχνεύει συνεχώς το CPU utilization και προσαρμόζει δυναμικά την κατανομή των πόρων προκειμένου να επιτευχθούν οι στόχοι του QoS.

2.3 Ανάλυση χρονοσειρών

Στόχος της ανάλυσης χρονοσειρών είναι η προσεκτική συλλογή και μελέτη των παρελθοντικών παρατηρήσεων των ιστορικών δεδομένων που συλλέγονται, για να να δημιουργήσει μελλοντική αξία για τη χρονοσειρά. Ορισμένα μοντέλα πρόβλεψης, όπως τα Autoregressive (AR), επικεντρώνονται στην άμεση πρόβλεψη των μελλοντικών τιμών, ενώ άλλες προσεγγίσεις όπως η αντιστοίχιση μοτίβων και οι τεχνικές επεξεργασίας σήματος, προσπαθούν πρώτα να εντοπίσουν μοτίβα και στη συνέχεια να προβλέψουν μελλοντικές τιμές. Οι Huang κ.ά [14] πρότειναν ένα μοντέλο πρόβλεψης (για CPU utilization και Memory usage) που βασίζεται σε διπλή εκθετική εξομάλυνση για τη βελτίωση της ακρίβεια πρόβλεψης στην παροχή πόρων. Οι Mi κ.ά. [15] χρησιμοποίησαν την τετραγωνική εκθετική εξομάλυνση του Brown για την πρόβλεψη μελλοντικού φόρτου εργασίας εφαρμογών παράλληλα με έναν αλγόριθμο για την εύρεση μιας σχεδόν βέλτιστης αναδιαμόρφωσης των εικονικών μηχανών. Οι Roy κ.ά. [16] παρουσίασαν ένα look-ahead αλγόριθμο κατανομής πόρων για την ελαχιστοποίηση της παροχής πόρων κόστους, ενώ παράλληλα εγγυάται το QoS της εφαρμογής. Οι Islam κ.ά. συνδυάζοντας artificial neural networks (τεχνητά νευρωνικά δίκτυα) και μια προσέγγιση sliding window σε προηγούμενα δεδομένα πρότειναν μία προσαρμοστική προσέγγιση για τη μείωση του κινδύνου παραβιάσεων του SLA με την αρχικοποίηση των VMs και την εκτέλεση της διαδικασίας εκκίνησής τους, πριν από τις αρχικές απαιτήσεις πόρων. Οι Gong κ.ά.[17] χρησιμοποίησαν τον Fast Fourier Transform (γρήγορο μετασχηματισμό Fourier) για τον εντοπισμό επαναλαμβανόμενων μοτίβων.

2.4 Reinfocement Learning

Η διαδικασία Reinforcement Learning (RL) (ενισχυτική μάθηση) [18] είναι η διαδικασία εκμάθησης ενός agent, για να ενεργεί με σκοπό τη μεγιστοποίηση των ανταμοιβών του. Η τυπική αρχιτεκτονική RL δίνεται ως εξής : ο agent ορίζεται ως ένας auto-scaler, η ενέργεια είναι η κλιμάκωση προς τα πάνω ή προς τα κάτω, το αντικείμενο είναι η εφαρμογή-στόχος και η ανταμοιβή είναι η βελτίωση της απόδοσης μετά την εφαρμογή της ενέργειας. Ο στόχος της RL είναι πώς να γίνει η επιλογή μιας ενέργειας σε μια δοσμένη κατάσταση, για τη μεγιστοποίηση των ανταμοιβών. Υπάρχουν διάφοροι τρόποι για την υλοποίηση της διαδικασίας μάθησης. Γενικά, οι RL προσεγγίσεις μαθαίνουν εκτιμήσεις των αρχικοποιημένων τιμών Q, όπου Q(s, a), και αντιστοιχίζουν όλες τις καταστάσεις του συστήματος s στην καλύτερη δράση τους, a. Αρχικοποιούνται όλα τα Q(s, a) και κατά τη διάρκεια της μάθησης, επιλέγεται μια δράση a για την κατάσταση s με βάση την ε-Greedy πολιτική και εφαρμόζεται στο στόχο-πλατφόρμα. Στη συνέχεια, παρατηρείται η νέα κατάσταση s’ και η ανταμοιβή r και ενημερώνεται η τιμή Q του τελευταίου ζεύγους κατάστασης-δράσης Q(s, a) σχετικά με την παρατηρούμενη κατάσταση του αποτελέσματος (s’) και της ανταμοιβής (r).

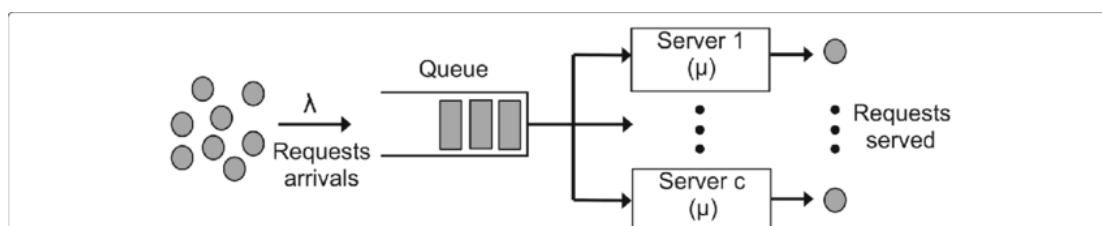


Σχήμα 3: Απλή διαδικασία ενισχυτικής μάθησης

Οι πιο γνωστές προσεγγίσεις RL είναι η SARSA και η Q-learning [18]. Οι Dutreilh κ.ά. [19] χρησιμοποιούν μια κατάλληλη αρχικοποίηση των τιμών Q για να αποκτήσουν μια καλή πολιτική από την αρχή καθώς και επιτάχυνση της σύγκλισης για να πετύχουν συνολικότερη επιτάχυνση της διαδικασίας εκμάθησης σε σύντομους χρόνους σύγκλισης. Οι Tesauro κ.ά. [20] προτείνουν ένα υβριδικό σύστημα μάθησης, συνδυάζοντας το μοντέλο δικτύου ουράς αναμονής και την προσέγγιση ενισχυτικής μάθησης SARSA, για να πετύχουν οι αποφάσεις κατανομής των πόρων να λαμβάνονται βάση του φόρτου εργασίας της εφαρμογής καθώς και του χρόνου απόκρισης. Το σημαντικότερο χαρακτηριστικό των προσεγγίσεων RL είναι η μάθηση χωρίς προηγούμενη γνώση του σεναρίου-στόχου και η δυνατότητα online μάθησης και ενημέρωσης της γνώσης του περιβάλλοντος, με πραγματικές παρατηρήσεις.

2.5 Μοντέλο ουράς αναμονής

Η θεωρία ουρών αναμονής αναφέρεται στη μαθηματική μελέτη του σχηματισμού, της λειτουργίας και της συμφόρησης των ουρών αναμονής. Ουσιαστικά μια κατάσταση ουράς περιλαμβάνει δύο κύρια μέρη. Κάποιος ή κάτι που ζητά μια υπηρεσία (συνήθως αναφέρεται ως πελάτης, εργασία ή αίτημα) και κάποιος ή κάτι που ολοκληρώνει ή παραδίδει τις υπηρεσίες (συνήθως αναφέρεται ως διακομιστής). Η θεωρία της ουράς εξετάζει ολόκληρο το σύστημα αναμονής στη σειρά, συμπεριλαμβανομένων στοιχείων όπως ο ρυθμός άφιξης των πελατών, ο αριθμός των διακομιστών, ο αριθμός των πελατών, η χωρητικότητα του χώρου αναμονής και ο μέσος χρόνος ολοκλήρωσης της εξυπηρέτησης και τους κανόνες της ουράς, για παράδειγμα αν συμπεριφέρεται με βάση την αρχή του first-in-first-out, last-in-first-out, προτεραιοποίηση ή εξυπηρέτηση με τυχαία σειρά [21].



Σχήμα 4: Διάγραμμα δικτύου που χρησιμοποιεί την ουρά αναμονής

Αρκετά μοντέλα ουράς περιγράφουν συστήματα autoscaling υποθέτοντας μια διαδικασία άφιξης χωρίς μνήμη και έναν προσαρμοστικό αριθμό διακομιστών. Οι Kaboudan κ.ά. [22] παρουσίασαν ένα μοντέλο ουράς αναμονής διακριτού χρόνου με δυναμικό αριθμό διακομιστών χρησιμοποιώντας μια πολιτική κλιμάκωσης με βάση τιμές-κατώφλια. Εκτέλεσαν προσομοιώσεις, και συνέκριναν το μοντέλο τους με τη συμπεριφορά μιας ουράς M/M/c. Οι Mazalov και Gurtov [23] πρότειναν ένα μοντέλο ουράς αναμονής με έναν προκαθορισμένο αριθμό διακομιστών, ο οποίος εξαρτιόταν από το μήκος της της ουράς, και υπέθεσαν μια διαδικασία αφίξεων Poisson με καθορισμένο ρυθμό. Οι Jia κ.ά. [24] εισήγαγαν ένα μοντέλο ουράς αναμονής το οποίο χρησιμοποίησε μια πολιτική με βάση τιμές-κατώφλια, για να περιγράψει καλύτερα μια βιομηχανική διαδικασία παραγωγής. Το μοντέλο υπέθεσε μια διαμορφωμένη κατά Markov διαδικασία Poisson (MMPP) ως άφιξη και ο αριθμός των διακομιστών επιλέχθηκε από δύο προκαθορισμένες τιμές με βάση το μήκος της ουράς. Σε όλες αυτές τις αναφερόμενες εργασίες βλέπουμε ότι η μαρκοβιανή υπόθεση των εκθετικών χρόνων μεταξύ των αφίξεων αποτελεί περιοριστικό παράγοντα.

2.6 Performance prediction (Πρόβλεψη Απόδοσης)

Οι Wajahat κ.ά.[25] πρότειναν μια εφαρμογή βασισμένη σε νευρωνικά δίκτυα και μία λύση αυτόματης κλιμάκωσης βασισμένη στη τεχνική regression, η οποία ονομάζεται MLscale. Χρησιμοποίησαν ένα νευρωνικό δίκτυο πολλαπλών επιπέδων για τη δημιουργία του μοντέλου απόδοσης της εφαρμογής, με βάση τον ρυθμό αιτημάτων και τα στατιστικά στοιχεία των πόρων του συστήματος. Το μοντέλο ήταν σε θέση να προβλέψει το χρόνο απόκρισης της εφαρμογής. Για την πρόβλεψη του νέου χρόνου απόκρισης μετά από μια προτεινόμενη ενέργεια κλιμάκωσης, εκπαίδευσαν πολλαπλά μοντέλα linear regression, τα οποία ήταν σε θέση να προσδιορίσουν τις μετρικές από την τρέχουσα κατάσταση λαμβάνοντας υπόψη την απόφαση κλιμάκωσης. Οι Rahman και Lama [26] πρότειναν μια λύση για την πρόβλεψη της καθυστέρησης της ουράς, από άκρο σε άκρο, των εφαρμογών που βασίζονται σε μικρουπηρεσίες και την κλιμάκωση με βάση αυτή. Εκπαίδευσαν διάφορες ML μεθόδους, όπως η random forest, η linear regression, η support vector regression και η deep NN, για να προβλέψουν την καθυστέρηση της εφαρμογής. Για μια δεδομένη κατάσταση φόρτου εργασίας, χρησιμοποίησαν το καλύτερο μοντέλο για να βρουν τις υψηλότερες τιμές αξιοποίησης των πόρων των σχετικών μικρουπηρεσιών, στις οποίες οι δεδομένοι στόχοι SLA δεν θα παραβιάζονταν, και το κατώφλι κλιμάκωσης ορίστηκε στην ανακτώμενη τιμή χρησιμοποίησης των του συστήματος. Οι Yu κ.ά. [27] παρουσίασαν τον Microscaler, ένα σύστημα autoscaling που εντόπιζε αυτόματα παραβιάσεις του SLA, προσδιόριζε τις υπηρεσίες που απαιτούν κλιμάκωση και αξιολογούσε πόσοι πόροι χρειαζόντουσαν. Εισήγαγαν τη Service Power, μια μετρική που μπορούσε να υπολογιστεί από τους χρόνους απόκρισης της υπηρεσίας, και χρησιμοποίησαν αυτή τη μετρική για να βρουν τις υπηρεσίες που χρειάζονταν κλιμάκωση. Οι Rzadca κ.ά.[28] παρείχαν μια λύση με την ονομασία Autopilot, έναν horizontal και vertical autoscaler βασισμένο σε τεχνικές ML, που χρησιμοποιείται από την Google για τη διαμόρφωση των πόρων, για τις απαιτούμενες εργασίες σε μια δουλειά. Ο πρωταρχικός στόχος ήταν να μειωθεί η διαφορά μεταξύ του ορίου και της πραγματικής χρήσης των πόρων (slack), ενώ ταυτόχρονα να ελαχιστοποιείται ο κίνδυνος να σκοτωθεί μια εργασία. Χρησιμοποίησαν διάφορες τεχνικές ML για την πρόβλεψη των vertical ορίων πόρων με βάση ιστορικά δεδομένα, καθώς και διάφορες προσεγγίσεις που βασίζονται σε κανόνες για την horizontal κλιμάκωση. Στην πράξη, οι

εργασίες με αυτόματη πλοήγηση εμφάνισαν ένα slack της τάξης του 23%, σε σύγκριση με 46% για τις εργασίες που διαχειρίζονται χειροκίνητα.

2.7 Demand forecast (Πρόβλεψη Ζήτησης)

Η βραχυπρόθεσμη πρόβλεψη της ζήτησης ήταν στο επίκεντρο των ερευνητών σε διάφορους τομείς εφαρμογών για πολλά χρόνια. Στον τομέα της ενέργειας για παράδειγμα, είναι πολύ σημαντικό να γνωρίζουμε εκ των προτέρων την ισχύ που παράγουν οι ανεμογεννήτριες. Για την επίλυση αυτού του προβλήματος οι Li κ.ά. [29] ανέπτυξαν ένα νευρωνικό δίκτυο τεσσάρων εισόδων το οποίο αποδείχθηκε καλύτερο από την παραδοσιακή προσέγγιση με μία παράμετρο. Οι Catalão κ.ά. [30] πρότειναν ένα feed-forward νευρωνικό δίκτυο τριών επιπέδων για την πρόβλεψη των τιμών της αγοράς ηλεκτρικής ενέργειας της επόμενης εβδομάδας. Η προσέγγισή τους αποδείχθηκε καλύτερη από τις μεθόδους που είχαν παρουσιαστεί προηγουμένως με το μοντέλο ARIMA (auto-regressive integrated moving average model) καθώς είναι λιγότερο χρονοβόρα και ευκολότερη στην εφαρμογή. Για το υπολογιστικό νέφος, ο Chen [31] υλοποίησε μια δυναμική τεχνικής παροχής διακομιστών για την ελαχιστοποίηση της κατανάλωσης ενέργειας των κέντρων δεδομένων. Στη μελέτη του παρουσιάστηκε μια προσαρμοσμένη μέθοδος AR για την πρόβλεψη του αριθμού των συνδέσεων στους διακομιστές του Windows Live Messenger. Οι Gong κ.ά. [32] εφάρμοσαν επεξεργασία σήματος και αλγόριθμους στατιστικής μάθησης για την επίτευξη προβλέψεων σε πραγματικό χρόνο των δυναμικών απαιτήσεων των πόρων της εφαρμογής. Η προβλεπόμενη λύσή τους βασιζόταν είτε σε περιπτώσεις ιστορικής χρήσης πόρων, είτε σε μια αλυσίδα Markov διακριτού χρόνου με πεπερασμένο αριθμό καταστάσεων, ανάλογα με το αν τα ίχνη εισόδου παρουσίαζαν επαναλαμβανόμενα μοτίβα ή όχι. Ένα αποτελεσματικό μοντέλο πρόβλεψης δόθηκε επίσης από τους Islam κ.ά. [33] για προσαρμοστική παροχή πόρων στο αποθηκευτικό νέφος. Αξιολόγησαν διάφορα μοντέλα ML, όπως τα νευρωνικά δίκτυα και η linear regression, σε ένα σύνολο δεδομένων. Ισχυρίστηκαν ότι η λύση τους ήταν κατάλληλη για την πρόβλεψη της ζήτησης πόρων πριν από την εγκατάσταση των VMs. Οι Messias κ.ά. [34] πρότειναν μια προσαρμοστική μέθοδο πρόβλεψης με τη χρήση genetic αλγορίθμων για να συνδυάσουν μοντέλα πρόβλεψης χρονοσειρών, όπως το ARIMA, και την απλή και εκτεταμένη εκθετική εξομάλυνση.

Κεφάλαιο 3 : Βασικές έννοιες από εργαλεία που χρησιμοποιήθηκαν

3.1 Βασικά στοιχεία του Kubernetes

Για τη δημιουργία των μικροϋπηρεσιών σε containers ένα από τα πιο διάσημα εργαλεία που χρησιμοποιούνται είναι το Docker. Στην πραγματικότητα ο κώδικας, οι εξαρτήσεις και το περιβάλλον της κάθε μικροϋπηρεσίας πακετάρονται σε ένα δυαδικό αρχείο που ονομάζεται εικόνα του container (container image), η οποία είναι διαθέσιμη και αποθηκεύεται σε ένα αποθετήριο εφαρμογών (registry). Το Docker διαθέτει το δικό του αποθετήριο, το «Docker Hub». Τα containers με τη σειρά τους δημιουργούνται τρέχοντας την εικόνα του container για την υπηρεσία που θα υλοποιήσουν [35]. Η χειροκίνητη κλιμάκωση των containerized εφαρμογών είναι πολύ απαιτητική και δύσκολη. Αυτό οφείλεται στο μεγάλο πλήθος υποτμημάτων της εφαρμογής (μικροϋπηρεσίες) αλλά και σε παράγοντες που πρέπει να ληφθούν υπόψιν, όπως η εξισορρόπηση φορτίου (load balancing), η ορθή κατανομή πόρων και η ασφαλής λειτουργία των containers.

Τα εργαλεία ενορχήστρωσης λύνουν το πρόβλημα αυτοματοποιώντας αυτές τις διαδικασίες και φροντίζοντας για τη σωστή εκτέλεση και τη διαθεσιμότητα των εφαρμογών. Το πιο διαδεδομένο και ευρέως χρησιμοποιούμενο εργαλείο ενορχήστρωσης είναι το Kubernetes (K8S) [36], το οποίο είναι ένα σύστημα ανοιχτού κώδικα με στόχο να ομαδοποιεί τα κατανεμημένα τμήματα της εφαρμογής, να τα εκτελεί και να τα διαχειρίζεται αυτόματα, με δυνατότητα κλιμάκωσής τους ανάλογα με την ανάγκη τους σε πόρους. Βασίζεται σε master-slave αρχιτεκτονική, με τα κύρια τμήματα (master-components) να διαχειρίζονται την κατάσταση του cluster (συστάδας υπολογιστών - εικονικών ή πραγματικών συνδεδεμένων μέσω δικτύου), το οποίο αποτελείται από κόμβους-εργάτες (worker-nodes) [37]. O master-node διαθέτει τα ακόλουθα components που βοηθούν στη διαχείριση των worker-nodes:

Kube-APIServer: ο οποίος λειτουργεί ως frontend (μπροστινό μέρος) για το cluster. Όλες οι εξωτερικές επικοινωνίες με το cluster γίνονται μέσω του API-Server.

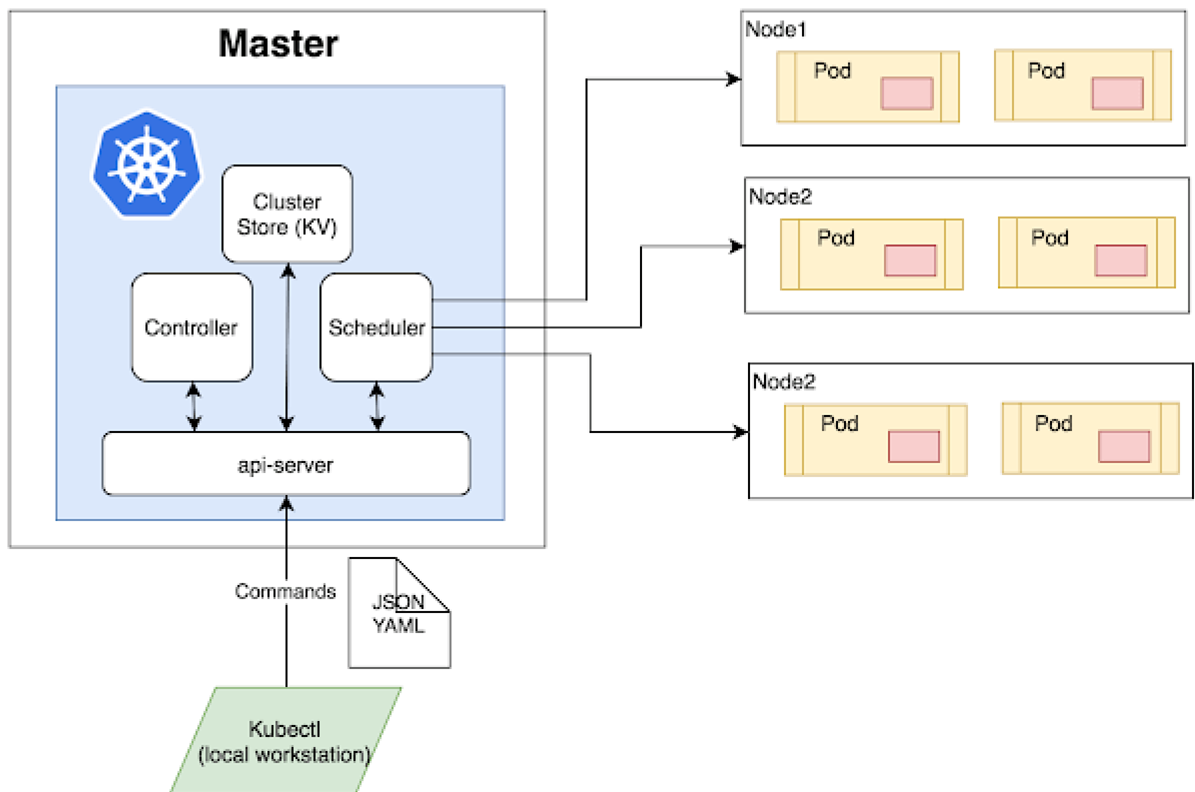
Kube-Controller-Manager: ο οποίος εκτελεί ένα σύνολο controllers (ελεγκτών) για το τρέχον cluster. Ο controller-manager υλοποιεί τη διακυβέρνηση σε όλο το cluster.

Etcd: η βάση δεδομένων κατάστασης του cluster.

Kube Scheduler: ο οποίος προγραμματίζει τις δραστηριότητες στους worker-nodes με βάση τα γεγονότα που συμβαίνουν στην etcd. Διατηρεί επίσης το σχέδιο πόρων των κόμβων για να καθορίσει την κατάλληλη ενέργεια για το συμβάν που προκλήθηκε.

O worker-node είναι μια μονάδα που διαθέτει υπολογιστικούς πόρους (CPU, RAM), η οποία συνιστά δομικό στοιχείο της συστάδας (cluster) στην οποία στήνεται μια containerized εφαρμογή. Οι κόμβοι είναι είτε φυσικά μηχανήματα (εξυπηρετητές) σε κέντρα φιλοξενίας υποδομής (data centers) είτε εικονικές μηχανές σε παρόχους υπολογιστικής

υποδομής (cloud) και στον καθένα μπορεί να τρέχουν ένα ή περισσότερα pods. Το pod είναι η μικρότερη υπολογιστική μονάδα που μπορεί να δημιουργηθεί και να διαχειριστεί από το Kubernetes, η οποία εκτελεί ένα ή περισσότερα containers που θα μοιράζονται τον ίδιο αποθηκευτικό χώρο και δίκτυο. Σε κάθε pod ιδανικά τρέχει ένα container, ώστε να απομονώνονται τα τμήματα της εφαρμογής [38]. Όταν καθορίζεται ένα pod, γίνεται προαιρετικά να καθοριστεί πόση ποσότητα από κάθε πόρο χρειάζεται το container του. Οι πιο συνηθισμένοι πόροι που καθορίζονται είναι η CPU (Central Processing Unit - Κεντρική Μονάδα Επεξεργασίας) και η μνήμη (RAM). Ουσιαστικά ορίζονται κάτω όρια (requests) και άνω όρια (limits) τα οποία αφορούν την ομαλή λειτουργία του pod. Το replicaset καθορίζει πόσα ενεργά αντίγραφα pods θέλουμε κάθε δεδομένη χρονική στιγμή. Για την διαχείριση του replicaset υπάρχει ένα υψηλότερο επίπεδο, το deployment (εκτελεστής). Το deployment καθορίζει τα replicas (αντίγραφα των pods), την container image που χρησιμοποιούμε αλλά και διάφορα άλλα πράγματα που αφορούν το pod, όπως για παράδειγμα το metadata tag το οποίο ονομάζει το pod και ταυτόχρονα του δίνει ένα label (ετικέτα) το οποίο είναι μοναδικό για το pod και βοηθάει στο να αναγνωρίζουν άλλα στοιχεία του Kubernetes το εν λόγω pod. Φροντίζει μάλιστα για την επαναδημιουργία τους σε περίπτωση που κάποιο από αυτά τερματίσει για οποιοδήποτε λόγο. Έτσι επιτυγχάνεται μια αυτοματοποίηση στη διαχείριση των Pods και διασφάλιση της επιθυμητής κατάστασης του συστήματος από τον ίδιο τον ενορχηστρωτή.



Σχήμα 5: Απλή αρχιτεκτονική ενός cluster του Kubernetes

Ακόμα στο Kubernetes, υπάρχει το στοιχείο Service που ορίζει ένα λογικό σύνολο pods και μια πολιτική με την οποία θα έχει πρόσβαση σε αυτά. Το σύνολο των pods που στοχεύει ένα service καθορίζεται συνήθως από τον selector (επιλογέα) , ο οποίος επιτρέπει στον χρήστη

να φιλτράρει τους διαθέσιμους πόρους κοιτώντας το κατάλληλο label του καθένα [39]. Το Kubernetes δίνει στα pods τις δικές τους διευθύνσεις IP (Internet Protocol) και ένα ενιαίο όνομα DNS (Domain Name System) για ένα σύνολο pods και μπορεί παράλληλα να εξισορροπήσει το φόρτο εργασίας μεταξύ τους. Για ορισμένα τμήματα εφαρμογών που χρειάζεται να εκθέσουν ένα service σε μια εξωτερική διεύθυνση IP (εκτός του cluster) το Kubernetes παρέχει τα ServiceTypes, τα οποία καθορίζουν το είδος του service που χρειάζεται ο χρήστης. Υπάρχουν 4 διαφορετικά ServiceTypes:

ClusterIP: Ο προεπιλεγόμενος τύπος ServiceType. Εκθέτει το service σε μια εσωτερική διεύθυνση IP του cluster. Η επιλογή αυτής της τιμής καθιστά το service προσβάσιμο μόνο από το εσωτερικό του cluster.

NodePort: Εκθέτει το service στην διεύθυνση IP κάθε κόμβου σε μια στατική θύρα (τη θύρα κόμβου). Δημιουργείται αυτόματα ένα ClusterIP service, στην οποία δρομολογείται το NodePort service. Μπορούμε να επικοινωνήσουμε με το NodePort service, από το εξωτερικό του cluster, ζητώντας <NodeIP>:<NodePort>.

LoadBalancer: Εκθέτει το service εξωτερικά χρησιμοποιώντας τον εξισορροπιστή φορτίου (load balancer) ενός παρόχου cloud. Οι NodePort και ClusterIP services, στις οποίες δρομολογεί ο εξωτερικός εξισορροπητής φορτίου, δημιουργούνται αυτόματα.

ExternalName: Χαρτογραφεί την Υπηρεσία στα περιεχόμενα του πεδίου externalName, επιστρέφοντας μια εγγραφή CNAME με την τιμή της.

3.2 Scalability

Ένα ακόμα σημαντικό στοιχείο του Kubernetes είναι το Autoscaling (αυτόματη κλιμάκωση). Αντί να κατανέμουμε τους πόρους χειροκίνητα, μπορούμε να δημιουργήσουμε αυτοματοποιημένες διαδικασίες που εξοικονομούν χρόνο και επιτρέπουν να ανταποκρινόμαστε γρήγορα στις αιχμές της ζήτησης καθώς και να εξοικονομούμε κόστος, μειώνοντας την κλίμακα όταν οι πόροι δεν είναι πλέον απαραίτητοι. Υπάρχουν δύο είδη scaling στο Kubernetes:

Pod-based-scaling (Κλιμάκωση με βάση τα Pod): που υποστηρίζεται από τον Horizontal Pod Autoscaler (HPA), τον Vertical Pod Autoscaler (VPA).

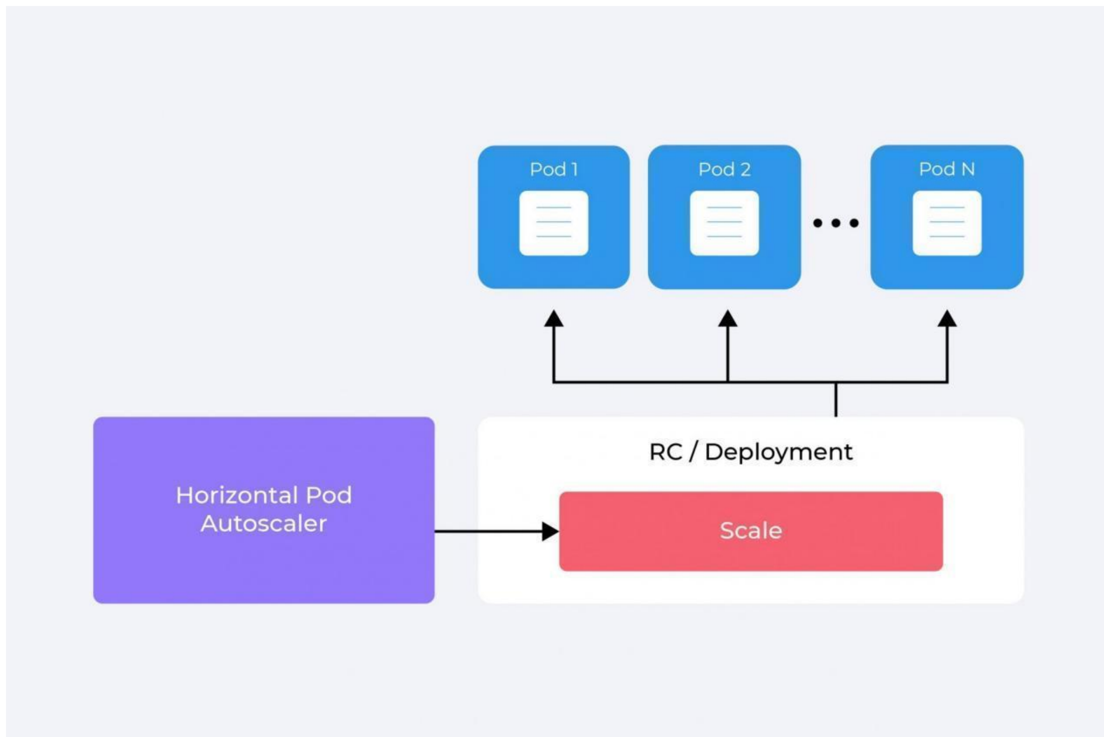
Node-based scaling (Κλιμάκωση με βάση τους κόμβους): υποστηριζόμενη από τον Cluster Autoscaler.

Ο Cluster Autoscaler αλλάζει τον αριθμό των κόμβων του cluster. Ο cluster autoscaler αναζητά pods που δεν μπορούν να προγραμματιστούν σε κάποιο κόμβο και προσπαθεί να ενοποιήσει pods που είναι επί του παρόντος αναπτυγμένα σε λίγους μόνο κόμβους. Τα μη προγραμματιζόμενα pods είναι αποτέλεσμα ανεπαρκών πόρων μνήμης ή CPU ή αδυναμίας αντιστοίχισης με έναν υπάρχοντα κόμβο λόγω των taint tolerations του pod (κανόνες που εμποδίζουν ένα pod να προγραμματίσει σε έναν συγκεκριμένο κόμβο), των κανόνων affinity (κανόνες που ωθούν ένα pod να αναπτυχθεί σε έναν μόνο συγκεκριμένο κόμβο). Εάν ένας cluster περιέχει pods χωρίς δυνατότητα προγραμματισμού, ο cluster autoscaler ελέγχει την ομάδα των διαχειριζόμενων κόμβων για να δει εάν η προσθήκη ενός νέου κόμβου μπορεί

να ξεμπλοκάρει το pod. Εάν ναι προσθέτει έναν κόμβο στην ομάδα κόμβων. Ο cluster autoscaler σαρώνει επίσης τους κόμβους μιας διαχειριζόμενης ομάδας κόμβων για πιθανό επαναπρογραμματισμό pods σε άλλους διαθέσιμους κόμβους του cluster. Εάν βρει κάτι τέτοιο, διώχνει τα pods και αφαιρεί τον κόμβο.

Ο Vertical Pod Autoscaler χρησιμοποιεί ζωντανά δεδομένα για να θέσει περιορισμούς στους πόρους των container. Τα περισσότερα container τηρούν περισσότερο τα αρχικά τους requests παρά τα limits. Ως αποτέλεσμα, ο προεπιλεγμένος προγραμματιστής του Kubernetes, συχνά υπερδεσμεύει τις κρατήσεις μνήμης και CPU ενός κόμβου. Για να αντιμετωπιστεί αυτό, ο VPA αυξάνει και μειώνει τα αιτήματα που γίνονται από pod containers για να διασφαλίσει ότι η πραγματική χρήση είναι σύμφωνη με τους διαθέσιμους πόρους μνήμης και CPU. Ο VPA υπολογίζει τις τιμές-στόχους παρακολουθώντας τη χρήση των πόρων, χρησιμοποιώντας το recommender στοιχείο του. Το updater στοιχείο του, διώχνει τα pods που πρέπει να ενημερωθούν με νέα όρια πόρων. Τέλος, το admission controller στοιχείο του αντικαθιστά τα requests πόρων των pod όταν δημιουργούνται, χρησιμοποιώντας ένα μεταβαλλόμενο admission webhook [40].

Ο Horizontal Pod Autoscaler είναι ο autoscaler με την πιο ευρεία και συχνή χρήση στο Kubernetes. Ένας HPA ενημερώνει αυτόματα έναν πόρο φόρτου εργασίας του cluster (όπως ένα Deployment), με στόχο την αυτόματη κλιμάκωση του πόρου φόρτου εργασίας, ώστε να ανταποκρίνεται στη ζήτηση. Η οριζόντια κλιμάκωση (horizontal scaling) σημαίνει ότι η απάντηση σε αυξημένο φορτίο είναι η ανάπτυξη περισσότερων pods. Αυτό διαφέρει από την κάθετη κλιμάκωση (vertical scaling) που αναλύσαμε προηγουμένως, η οποία για το Kubernetes θα σήμαινε την ανάθεση περισσότερων πόρων στα pods που ήδη εκτελούνται για το φόρτο εργασίας. Εάν το φορτίο μειωθεί και ο αριθμός των pods είναι πάνω από το ρυθμισμένο ελάχιστο όριο, το HorizontalPodAutoscaler δίνει εντολή στον πόρο φόρτου εργασίας να μειώσει ξανά την κλίμακα. Ο HPA υλοποιείται ως πόρος του Kubernetes API και ταυτόχρονα ως ελεγκτής. Αυτό σημαίνει πως ο τρόπος που ορίζεται ως πόρος στο Kubernetes επηρεάζει και τον τρόπο που συμπεριφέρεται ως ελεγκτής. Ο HPA controller προσαρμόζει περιοδικά την επιθυμητή κλίμακα του στόχου του (π.χ. του Deployment) ώστε να διατηρείται πάντα η τιμή που έχουμε επιλέξει για την μετρική που παρακολουθούμε, που μπορεί να είναι η μέση χρήση CPU, η μέση χρήση μνήμης ή οποιαδήποτε άλλη προσαρμοσμένη μετρική θέλουμε. Ο HorizontalPodAutoscaler ελέγχει την κλίμακα Deployment και του ReplicaSet του. Το Kubernetes υλοποιεί τον HPA ως έναν βρόχο ελέγχου που εκτελείται κατά διαστήματα. Το διάστημα ορίζεται από την παράμετρο --horizontal-pod-autoscaler-sync-period στον kube-controller-manager (και το προεπιλεγμένο διάστημα είναι 15 δευτερόλεπτα). Μία φορά κατά τη διάρκεια κάθε περιόδου, ο controller manage πραγματοποιεί ερωτήματα (queries) για τη χρήση των πόρων σε σχέση με τις μετρικές που καθορίζονται στον ορισμό του πόρου του HPA. Ο controller manager βρίσκει τον πόρο-στόχο, στη συνέχεια επιλέγει τα pods με βάση τις ετικέτες του πόρου-στόχου και λαμβάνει τις μετρήσεις είτε από το resource metrics API (για per-pod μετρικές), είτε από το custom metrics API (για όλες τις άλλες μετρικές).



Σχήμα 6: Δομή λειτουργίας του HPA

Για per-pod metrics (όπως η CPU utilization), ο controller αντλεί τις μετρικές από το resource metrics API για κάθε pod που στοχεύει ο HPA. Στη συνέχεια, εάν έχει οριστεί μια τιμή στόχου χρήσης, ο ελεγκτής υπολογίζει την τιμή χρήσης (utilization value) ως ποσοστό της αντίστοιχης αίτησης πόρων για τα containers σε κάθε Pod. Στη συνέχεια, ο controller λαμβάνει το μέσο όρο της τιμής χρησιμοποίησης σε όλα τα στοχευμένα Pods και παράγει μια αναλογία που χρησιμοποιείται για την κλιμάκωση του αριθμού των επιθυμητών replicas. Για τα per-pod custom metrics, ο ελεγκτής λειτουργεί παρόμοια με τις per-pod metrics, με τη διαφορά ότι λειτουργεί με ακατέργαστες τιμές (raw values) και όχι με τιμές χρήσης. Για τα object metrics και external metrics, ανακτάται μία μόνο μετρική, η οποία περιγράφει το εν λόγω αντικείμενο. Αυτή η μετρική συγκρίνεται με την τιμή-στόχο, για να παραχθεί μια αναλογία όπως παραπάνω. Η συνήθης χρήση του HPA είναι να ρυθμίζεται ώστε να λαμβάνει μετρικές από aggregated APIs (metrics.k8s.io, custom.metrics.k8s.io ή external.metrics.k8s.io).

Ο αλγόριθμος πίσω από τον HPA είναι ο εξής και λειτουργεί με βάση την αναλογία της επιθυμητής τιμής της μετρικής μαζί με της τρέχουσας :

```
desiredReplicas = ceil[currentReplicas * ( currentMetricValue / desiredMetricValue )]
```

Ουσιαστικά ο αλγόριθμος αυτός παίρνει την τωρινή τιμή της μετρικής-στόχου (currentMetricValue) τη διαιρεί με την επιθυμητή τιμή, που έχει ορίσει ο χρήστης, της μετρικής-στόχου (desiredMetricValue) και έπειτα την πολλαπλασιάζει με τον αριθμό των τωρινών replicas (currentReplicas), του Deployment που επιθυμούμε να κάνουμε scale. Τέλος, στρογγυλοποιεί το αποτέλεσμα της παραπάνω μαθηματικής πράξης στον μεγαλύτερο πλησιέστερο ακέραιο και εξάγει τον επιθυμητό αριθμό replicas (desiredReplicas) για να μπορέσει να αρχίσει το scaling. Για παράδειγμα αν η desiredMetricValue ήταν 60, η currentMetricValue ήταν 200 και οι currentReplicas ήταν 1, ο HPA θα δήλωνε πως

χρειαζόμαστε 4 desiredReplicas (αφόυ 200/60 = 3.333 και μετά στρογγυλοποιείται στο 4) [41].

3.3 Custom Pod Autoscaler

Πέρα από τους διαθέσιμους autoscaler που παρέχουν ήδη το Kubernetes, υπάρχει η δυνατότητα , μέσω του στοιχείου Custom Resource Definition (CRD) του Kubernetes, κάθε χρήστης να φτιάξει τον δικό του εξατομικευμένο autoscaler. Για την εκπόνηση της παρούσας διπλωματικής χρησιμοποιήσαμε το Custom Pod Autoscaler Framework (https://github.com/jthomperoo/custom-pod-autoscaler) το οποίο παρέχει ένα πρόγραμμα που αφαιρεί τις πολύπλοκες αλληλεπιδράσεις του Kubernetes και χειρίζεται την εξατομικευμένη λογική του χρήστη, που καθορίζει τον τρόπο λειτουργίας του autoscaler. Το συγκεκριμένο Framework υποστηρίζει όλες τις δυνατότητες του HPA και επιπλέον παρέχει τα εξής χαρακτηριστικά :

1) Υποστήριξη οποιασδήποτε προγραμματιστικής γλώσσας και περιβάλλοντος.
2) Επιτρέπει το scaling από και προς τις μηδέν replicas.
3) Μπορεί να αναπτυχθεί χωρίς πρόσβαση στον master node.
4) Επιτρέπει οποιαδήποτε λογική χρήστη για το πως θα λαμβάνεται η απόφαση για τον αριθμό των desiredReplicas.

Ουσιαστικά ο λόγος που επιλέξαμε τον Custom Pod Autoscaler είναι ο περιορισμός που έχει ο HPA στον αλγόριθμο απόφασης για το scaling. Αν θέλουμε η απόφαση για το scaling να αφορά παραπάνω από μία μετρική-στόχο ή να ξεφεύγει από την απλή μαθηματική πράξη του HPA και να ακολουθεί έναν διαφορετικό αλγόριθμο , τότε ο HPA δεν μας καλύπτει. Οπότε επιλέξαμε τον CPA για να έχουμε μεγαλύτερη ευελιξία γράφοντας την δική μας scaling λογική [42]. Η ανάπτυξη του σε περιβάλλον Kubernetes είναι αρκετά εύκολη και ο ορισμός του βασίζεται σε 4 διαφορετικά αρχεία [43] :

1) Αρχικά ρυθμίζουμε τις παραμέτρους του CPA, ποια scripts θα τρέχουμε , πως θα τα καλούμε και θέτουμε ένα χρονικό όριο εκπλήρωσής τους. Ορίζουμε ακόμα και το πόσες φορές θα τρέχει ο CPA μας (π.χ. per-resource που σημαίνει πως θα τρέχει μία φορά για κάθε πόρο ή per-pod που σημαίνει πως θα τρέχει μία φορά για κάθε διαφορετικό pod)

```yaml
evaluate:
  type: "shell"
  timeout: 2500
  shell:
    entrypoint: "python"
    command:
      - "/evaluate.py"
metric:
  type: "shell"
  timeout: 2500
  shell:
    entrypoint: "python"
    command:
      - "/metric.py"
runMode: "per-resource"
```

Σχήμα 7: Ένα απλό configuration file του CPA

2) Έπειτα δημιουργούμε το τμήμα συλλογής μετρικών του scaling, όπου αυτό το τμήμα θα διαβάζει απλώς την περιγραφή του πόρου-στόχου που παρέχεται και θα εξάγει την τιμή της μετρικής που επιθυμούμε (εδώ η τιμή numPods, πριν την εκδώσει πίσω για να λάβει ο αξιολογητής μια απόφαση.

```python
import os
import json
import sys

def main():
    # Parse spec into a dict
    spec = json.loads(sys.stdin.read())
    metric(spec)

def metric(spec):
    # Get metadata from resource information provided
    metadata = spec["resource"]["metadata"]
    # Get labels from provided metdata
    labels = metadata["labels"]

    if "numPods" in labels:
        # If numPods label exists, output the value of the numPods
        # label back to the autoscaler
        sys.stdout.write(labels["numPods"])
    else:
        # If no label numPods, output an error and fail the metric gathering
        sys.stderr.write("No 'numPods' label on resource being managed")
        exit(1)

if __name__ == "__main__":
    main()
```

Σχήμα 8: Παράδειγμα αρχείου metric.py για τη συλλογή μετρικών

3) Στη συνέχεια δημιουργούμε τον αξιολογητή , ο οποίος θα διαβάζει τις μετρικές που παρέχονται στο προηγούμενο βήμα και θα υπολογίζει με κάποιον τρόπο τον αριθμό των desiredReplicas με βάση αυτές τις μετρικές (σε αυτή την περίπτωση θα είναι απλώς η τιμή numPods). Εάν η τιμή δεν είναι ακέραιος αριθμός, το σενάριο θα επιστρέψει ένα σφάλμα.

```python
import json
import sys
import math

def main():
    # Parse provided spec into a dict
    spec = json.loads(sys.stdin.read())
    evaluate(spec)

def evaluate(spec):
    try:
        value = int(spec["metrics"][0]["value"])

        # Build JSON dict with targetReplicas
        evaluation = {}
        evaluation["targetReplicas"] = value

        # Output JSON to stdout
        sys.stdout.write(json.dumps(evaluation))
    except ValueError as err:
        # If not an integer, output error
        sys.stderr.write(f"Invalid metric value: {err}")
        exit(1)

if __name__ == "__main__":
    main()
```

Σχήμα 9: Παράδειγμα αρχείου evaluate.py για τη λογική κλιμάκωσης

4) Τέλος δημιουργούμε ένα Dockerfile στο οποίο βάζουμε όλες τις απαραίτητες εξαρτήσεις που χρειαστήκαμε στα προηγούμενα βήματα (π.χ. βιβλιοθήκες python που χρησιμοποιήσαμε) καθώς και τα 3 παραπάνω αρχεία . Κάνουμε ένα docker build στο Kubernetes και έχουμε έτοιμη την container image του autoscaler μας.

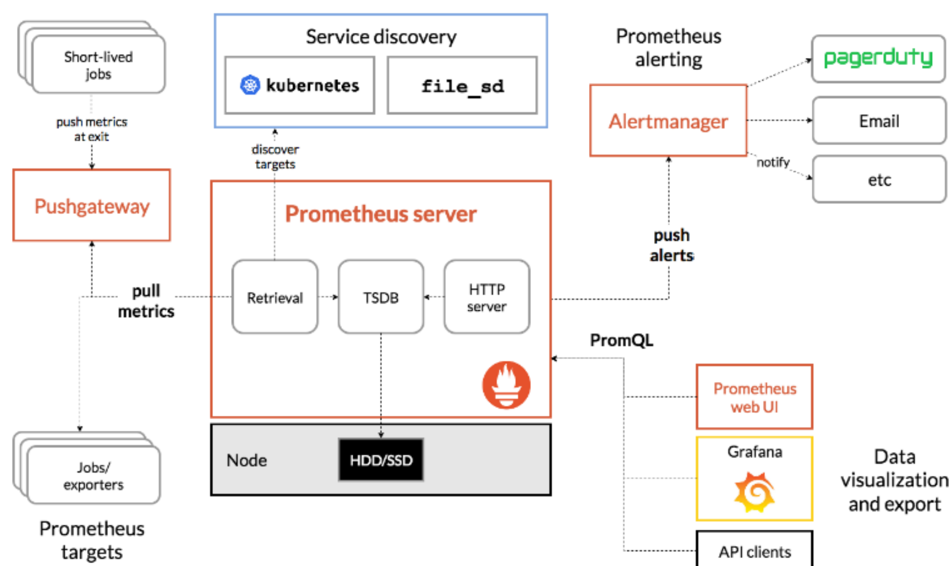
```dockerfile
# Pull in Python build of CPA
FROM custompodautoscaler/python:latest

# Add config, evaluator and metric gathering Py scripts
ADD config.yaml evaluate.py metric.py /
```

Σχήμα 10: Απλό παράδειγμα ενός Dockerfile αρχείου

3.4 Εργαλεία παρακολούθησης και οπτικοποίησης των μετρικών

Για την παρακολούθηση των επιθυμητών μετρικών που παράγει η εφαρμογή μας (π.χ. CPU utilization , αριθμός replicas , response time) χρησιμοποιήσαμε τον “Prometheus”. Ο Prometheus είναι ένα εργαλείο παρακολούθησης και ειδοποιήσεων συστημάτων ανοιχτού κώδικα. Συλλέγει και αποθηκεύει μετρικές ως δεδομένα χρονοσειράς, δηλαδή οι

πληροφορίες των μετρικών αποθηκεύονται μαζί με τη χρονική στιγμή κατά την οποία καταγράφηκαν, μαζί με προαιρετικά ζεύγη κλειδιών-τιμών που ονομάζονται ετικέτες (labels) [44].



Σχήμα 11: Λειτουργία ενός Prometheus server

Ο προμηθέας συλλέγει τις μετρικές που παράγει η εφαρμογή μας και αποθηκεύει τοπικά όλα τα δεδομένα που συλλέγονται και έπειτα εκτελεί κανόνες πάνω σε αυτά τα δεδομένα είτε για να συγκεντρώσει και να καταγράψει νέες χρονοσειρές από τα υπάρχοντα δεδομένα είτε για να δημιουργήσει ειδοποιήσεις. Το συγκεκριμένο επιτυγχάνεται μέσω του Prometheus-Operator, ο οποίος παρέχει την ανάπτυξη και τη διαχείριση του Prometheus και των σχετικών στοιχείων παρακολούθησης του Kubernetes [45].

Για την οπτικοποίηση των μετρικών που εξάγουμε από τον Prometheus χρησιμοποιούμε την “Grafana” ένα λογισμικό ανοικτού κώδικα που επιτρέπει την αναζήτηση και οπτικοποίηση των μετρικών και των αρχείων καταγραφής, οπουδήποτε και αν είναι αυτά αποθηκευμένα. Παρέχει εργαλεία για να μετατρέψουμε τα δεδομένα της βάσης δεδομένων χρονοσειρών (TSDB) σε διορατικά γραφήματα και οπτικοποιήσεις [46].

3.5 Γλώσσα προγραμματισμού

Για την δημιουργία του CPA χρησιμοποιήθηκε η γλώσσα προγραμματισμού Python [47]. Η Python είναι μια εύκολη στην εκμάθηση, ισχυρή γλώσσα προγραμματισμού. Διαθέτει αποδοτικές δομές δεδομένων υψηλού επιπέδου και μια απλή αλλά αποτελεσματική προσέγγιση στον αντικειμενοστραφή προγραμματισμό. Ο διερμηνέας της Python και η εκτεταμένη πρότυπη βιβλιοθήκη διατίθενται ελεύθερα σε πηγαία ή δυαδική μορφή για όλες τις κύριες πλατφόρμες. Ακόμα χρησιμοποιήσαμε αρκετά python modules σε διάφορα στάδια της εργασίας αυτής. Η Python διαθέτει έναν τρόπο να τοποθετείς ορισμούς σε ένα αρχείο και να τους χρησιμοποιείς σε ένα script. Ένα τέτοιο αρχείο ονομάζεται module

(δομικό στοιχείο) - οι ορισμοί από μια ενότητα μπορούν να εισαχθούν σε άλλες modules ή στην main module. Ένα module είναι ουσιαστικά ένα αρχείο που περιέχει ορισμούς και εντολές της Python [48]. Τα δύο σημαντικότερα modules που χρησιμοποιήσαμε στην εργασία είναι το Scikit-learn και το pandas. Το pandas είναι ένα module της Python που παρέχει γρήγορες και ευέλικτες δομές δεδομένων, σχεδιασμένες για να κάνουν την εργασία με δεδομένα εύκολη. Στόχος του είναι να αποτελέσει το θεμελιώδες δομικό στοιχείο για την πραγματοποίηση πρακτικών αναλύσεων δεδομένων στην Python.
Το pandas είναι κατάλληλο για πολλά διαφορετικά είδη δεδομένων: πίνακες δεδομένων με ετερογενείς τυποποιημένες στήλες, όπως σε ένα φύλλο εργασίας του Excel, ταξινομημένα και μη ταξινομημένα δεδομένα χρονοσειρών καθώς και οποιαδήποτε άλλη μορφή συνόλων στατιστικών δεδομένων. Οι δύο κύριες δομές δεδομένων του pandas, Series (μονοδιάστατη) και DataFrame (δισδιάστατη), χειρίζονται τη συντριπτική πλειοψηφία των τυπικών περιπτώσεων χρήσης στα χρηματοοικονομικά, τη στατιστική, τις κοινωνικές επιστήμες και πολλούς τομείς της μηχανικής [49]. Το scikit-learn είναι ίσως το πιο χρήσιμο module για μηχανική μάθηση στην Python. Η βιβλιοθήκη sklearn περιέχει πολλά αποδοτικά εργαλεία για μηχανική μάθηση και στατιστική μοντελοποίηση. Μερικά χαρακτηριστικά της είναι : αλγόριθμοι επιβλεπόμενης μάθησης, cross-validation (διασταυρούμενη επαλήθευση), αλγόριθμοι μάθησης χωρίς επίβλεψη [50].

3.6 Αλγόριθμοι επιβλεπόμενης μηχανικής μάθησης

Τα Random Forests ή random decision forests είναι μια μέθοδος μάθησης συνόλου για classification (ταξινόμηση) , regression (παλινδρόμηση) και άλλες εργασίες, που λειτουργεί με την κατασκευή ενός πλήθους decision trees (δέντρων απόφασης) κατά τη διάρκεια της εκπαίδευσης. Για εργασίες classification, η έξοδος του random forest είναι η κλάση που επιλέγεται από τα περισσότερα δέντρα. Για εργασίες regression, επιστρέφεται ο μέσος όρος ή η μέση πρόβλεψη των επιμέρους δέντρων. Τα random decision forests διορθώνουν τη συνήθεια των δέντρων απόφασης να προσαρμόζονται υπερβολικά στο σύνολο εκπαίδευσής τους (overfitting). Ο αλγόριθμος εκπαίδευσης για τα random forests εφαρμόζει τη γενική τεχνική του bootstrap aggregating, ή bagging, σε μαθητές δέντρα. Δεδομένου ενός συνόλου εκπαίδευσης $X = x_1, ..., x_n$ με αποκρίσεις $Y = y_1, ..., y_n$, το bagging επιλέγει επανειλημμένα (B φορές) ένα τυχαίο δείγμα με αντικατάσταση από το σύνολο εκπαίδευσης και προσαρμόζει δέντρα σε αυτά τα δείγματα:

Για b = 1, ..., B:

Δείγμα, με αντικατάσταση, n παραδείγματα εκπαίδευσης από τα X, Y- ονομάστε τα X_b, Y_b.

Εκπαιδεύστε ένα δέντρο classification ή regression f_b στα X_b, Y_b.

Μετά την εκπαίδευση, μπορούν να γίνουν προβλέψεις για αθέατα δείγματα x' με τη μέση τιμή των προβλέψεων από όλα τα μεμονωμένα δέντρα regression στο x':

$$\hat{f} = \frac{1}{B}\sum_{b=1}^{B} f_b(x')$$

ή με τη λήψη της πλειοψηφίας στην περίπτωση των δέντρων classification.

Αυτή η διαδικασία bootstrapping οδηγεί σε καλύτερη απόδοση του μοντέλου επειδή μειώνει τη διακύμανση (variance) του μοντέλου, χωρίς να αυξάνει την προκατάληψη (bias). Αυτό σημαίνει ότι ενώ οι προβλέψεις ενός μεμονωμένου δέντρου είναι ιδιαίτερα ευαίσθητες στο θόρυβο του συνόλου εκπαίδευσής του, ο μέσος όρος πολλών δέντρων δεν είναι, εφόσον τα δέντρα δεν συσχετίζονται. Η απλή εκπαίδευση πολλών δέντρων σε ένα μόνο σύνολο εκπαίδευσης θα έδινε έντονα συσχετιζόμενα δέντρα (ή ακόμη και το ίδιο δέντρο πολλές φορές, αν ο αλγόριθμος εκπαίδευσης είναι ντετερμινιστικός) - η δειγματοληψία bootstrap είναι ένας τρόπος απο-συσχέτισης των δέντρων, παρουσιάζοντάς τους διαφορετικά σύνολα εκπαίδευσης. Η παραπάνω διαδικασία περιγράφει τον αρχικό αλγόριθμο bagging για δέντρα. Τα random forests περιλαμβάνουν επίσης έναν άλλο τύπο σχήματος bagging: χρησιμοποιούν έναν τροποποιημένο αλγόριθμο εκμάθησης δέντρων που επιλέγει, σε κάθε υποψήφιο διαχωρισμό στη διαδικασία εκμάθησης, ένα τυχαίο υποσύνολο των χαρακτηριστικών (features). Αυτή η διαδικασία ονομάζεται "feature bagging". Ο λόγος που γίνεται αυτό είναι η συσχέτιση των δέντρων σε ένα συνηθισμένο δείγμα bootstrap: εάν ένα ή λίγα χαρακτηριστικά είναι πολύ ισχυροί προγνωστικοί παράγοντες για τη μεταβλητή απόκρισης (έξοδος-στόχος), αυτά τα χαρακτηριστικά θα επιλεγούν σε πολλά από τα δέντρα B, προκαλώντας τη συσχέτισή τους [51].

Στη στατιστική μοντελοποίηση, η regression analysis είναι ένα σύνολο στατιστικών διαδικασιών για την εκτίμηση των σχέσεων μεταξύ μιας εξαρτημένης μεταβλητής (συχνά αποκαλούμενης ως μεταβλητή "αποτελέσματος" ή ως "label" στην ορολογία της μηχανικής μάθησης) και μιας ή περισσότερων ανεξάρτητων μεταβλητών (συχνά αποκαλούμενων ως "προβλεπτικές μεταβλητές", ή “features”). Οι δύο βασικοί τύποι regression είναι η simple linear regression και η multiple linear regression, αν και υπάρχουν μέθοδοι non linear regression για πιο περίπλοκα δεδομένα και αναλύσεις. Η simple linear regression χρησιμοποιεί μία ανεξάρτητη μεταβλητή για να εξηγήσει ή να προβλέψει το αποτέλεσμα της εξαρτημένης μεταβλητής Y, ενώ η multiple linear regression χρησιμοποιεί δύο ή περισσότερες ανεξάρτητες μεταβλητές για να προβλέψει το αποτέλεσμα. Η γενική μορφή κάθε τύπου regression είναι η εξής:

simple linear regression : $Y = a + bX + u$
multiple linear regression: $Y = a + b_1X_1 + b_2X_2 + b_3X_3 + ... + b_tX_t + u$

Όπου:

Y = η μεταβλητή που προσπαθούμε να προβλέψουμε (εξαρτημένη μεταβλητή).
X = η μεταβλητή που χρησιμοποιούμε για να προβλέψουμε την Y (ανεξάρτητη μεταβλητή).
α = η τομή.
b = η κλίση.
u = το υπόλοιπο.

Η regression παίρνει μια ομάδα τυχαίων μεταβλητών, που θεωρείται ότι προβλέπουν την Y, και προσπαθεί να βρει μια μαθηματική σχέση μεταξύ τους. Η σχέση αυτή έχει συνήθως τη μορφή μιας ευθείας γραμμής (linear regression) που προσεγγίζει καλύτερα όλα τα επιμέρους σημεία δεδομένων. Στην multiple regression, οι επιμέρους μεταβλητές διαφοροποιούνται με τη χρήση δεικτών [52].

Στη μηχανική μάθηση, τα support-vector machines (SVM) είναι μοντέλα επιβλεπόμενης μάθησης που αναλύουν δεδομένα για classification και regression analysis. Δεδομένου ενός συνόλου παραδειγμάτων εκπαίδευσης, καθένα από τα οποία χαρακτηρίζεται ότι ανήκει σε μία από τις δύο κατηγορίες, ένας αλγόριθμος εκπαίδευσης SVM δημιουργεί ένα μοντέλο που αναθέτει τα νέα παραδείγματα στη μία ή την άλλη κατηγορία, καθιστώντας τον έναν μη-στοχαστικό δυαδικό γραμμικό ταξινομητή. Ο SVM αντιστοιχίζει τα παραδείγματα εκπαίδευσης σε σημεία στο χώρο έτσι ώστε να μεγιστοποιεί το πλάτος του χάσματος μεταξύ των δύο κατηγοριών. Στη συνέχεια, τα νέα παραδείγματα αντιστοιχίζονται στον ίδιο χώρο και προβλέπεται ότι θα ανήκουν σε μια κατηγορία με βάση το σε ποια πλευρά του χάσματος εμπίπτουν. Όταν τα δεδομένα δεν έχουν labels, η επιβλεπόμενη μάθηση δεν είναι δυνατή και απαιτείται μια προσέγγιση μάθησης χωρίς επίβλεψη, η οποία προσπαθεί να βρει φυσική ομαδοποίηση των δεδομένων σε ομάδες και στη συνέχεια να αντιστοιχίσει τα νέα δεδομένα σε αυτές τις σχηματιζόμενες ομάδες. Μας δίνεται ένα training dataset με n σημεία της μορφής :

$(x_1,y_1), \ldots, (x_n,y_n)$ όπου το y_i παίρνει τιμές 1 ή -1 , καθεμία από τις οποίες δείχνουν την κλάση που ανήκει το x_i. Κάθε x_i είναι ένα διάνυσμα πραγματικών αριθμών p-διαστάσεων. Ο στόχος είναι να βρούμε το “maximum-margin hyperplane” το οποίο χωρίζει τις ομάδες των στοιχείων x_i, για τα οποία $y_i = 1$ από τις ομάδες των στοιχείων x_i, για τα οποία $y_i = -1$ και ταυτόχρονα ορίζεται με τέτοιο τρόπο ώστε η απόσταση μεταξύ του hyperplane και του κοντινότερου σημείου x_i από κάθε ομάδα να είναι μέγιστη. Ένα οποιοδήποτε hyperplane μπορεί να εκφραστεί ως η ομάδα των στοιχείων x που ικανοποιεί την εξής σχέση :

$w^T x - b = 0$, όπου το w το κάθετο διάνυσμα στο hyperplane [53].

Ο ταξινομητής Naive Bayes είναι ένα στοχαστικό μοντέλο μηχανικής μάθησης που χρησιμοποιείται για εργασίες classification. Η ουσία του ταξινομητή βασίζεται στο παρακάτω θεώρημα του Bayes :

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Χρησιμοποιώντας το θεώρημα του Bayes, μπορούμε να βρούμε την πιθανότητα να συμβεί το γεγονός A, δεδομένου ότι έχει συμβεί το γεγονός B. Εδώ, το B είναι η απόδειξη και το A είναι η υπόθεση. Η υπόθεση που γίνεται εδώ είναι ότι τα features (χαρακτηριστικά) είναι ανεξάρτητα. Δηλαδή η παρουσία ενός συγκεκριμένου χαρακτηριστικού δεν επηρεάζει το άλλο. Αναφορικά με το dataset που θα χρησιμοποιηθεί ορίζουμε ως C_k τις πιθανές κλάσεις και και ως x το διάνυσμα $(x_1, \ldots, x_n)$ με τα n ανεξάρτητα features. Άρα έχω

$$p(C_k \mid \mathbf{x}) = \frac{p(C_k)\, p(\mathbf{x} \mid C_k)}{p(\mathbf{x})}$$

Στη παραπάνω σχέση μας ενδιαφέρει ο αριθμητής καθώς ο παρονομαστής δεν εξαρτάται από το C και αφού τα features είναι γνωστά θεωρείται σταθερός όρος. Με βάση το μοντέλο της κοινής κατανομής ο αριθμητής γίνεται $p(C_k, x_1, \ldots, x_n)$ και έπειτα από τον κανόνα αλυσίδας έχουμε :

$p(C_k, x_1, \ldots, x_n) = p(x_1 \mid x_2, \ldots, x_n, C_k)\, p(x_2 \mid x_3, \ldots, x_n, C_k) \ldots p(x_{n-1} \mid x_n, C_k)\, p(x_n \mid C_k)\, p(C_k)$. Καθώς τα features είναι ανεξάρτητα έχουμε $p(x_1 \mid x_{\iota+1}, \ldots, x_n, C_k) = p(x_\iota \mid C_k)$ και τελικά

$p(C_k \mid x_1, \ldots, x_n) \propto p(C_k) \prod_{i=1}^{n} p(x_\iota \mid C_k)$, όπου το ∞ δηλώνει αναλογικότητα. Τώρα, πρέπει να δημιουργήσουμε ένα μοντέλο ταξινομητή. Ο ταξινομητής naive Bayes συνδυάζει το μοντέλο Bayes με έναν κανόνα απόφασης. Ένας συνήθης κανόνας είναι να επιλέγεται η πιο πιθανή υπόθεση, ώστε να ελαχιστοποιείται η πιθανότητα εσφαλμένης classification. Ο αντίστοιχος ταξινομητής, ένας ταξινομητής Bayes [54], είναι η συνάρτηση που αποδίδει μια ετικέτα κλάσης $y = C_k$ για κάποιο k ώς εξής :

$$\hat{y} = \underset{k \in \{1, \ldots, K\}}{\operatorname{argmax}}\ p(C_k) \prod_{i=1}^{n} p(x_i \mid C_k).$$

Στη στατιστική, ο αλγόριθμος k-nearest neighbors (κοντινότεροι γείτονες) (k-NN) είναι μια μη παραμοποιημένη επιβλεπόμενη μέθοδος μηχανικής μάθησης που χρησιμοποιείται τόσο για classification όσο και για regression. Και στις δύο περιπτώσεις, η είσοδος αποτελείται από τα k πλησιέστερα παραδείγματα εκπαίδευσης σε ένα σύνολο δεδομένων. Η έξοδος εξαρτάται από το αν ο k-NN χρησιμοποιείται για classification ή για regression. Στην classification k-NN, η έξοδος είναι η ένταξη σε ένα class. Ένα αντικείμενο ταξινομείται με βάση την πλειοψηφία της “ψήφων” των γειτόνων του, με το αντικείμενο να κατατάσσεται στην κλάση που είναι πιο συχνή μεταξύ των k πλησιέστερων γειτόνων του (το k είναι ένας θετικός ακέραιος αριθμός, συνήθως μικρός). Εάν το k ισούται με 1, τότε το αντικείμενο απλώς κατατάσσεται στην κλάση αυτού του μοναδικού πλησιέστερου γείτονα. Δεδομένου ότι αυτός ο αλγόριθμος βασίζεται στην απόσταση για το classification, εάν τα features αντιπροσωπεύουν διαφορετικές φυσικές μονάδες ή έρχονται σε πολύ διαφορετικές κλίμακες, τότε η κανονικοποίηση των δεδομένων εκπαίδευσης μπορεί να βελτιώσει δραματικά την ακρίβεια του αλγορίθμου. Τόσο για το classification όσο και για το regression, μια χρήσιμη τεχνική μπορεί να είναι η ανάθεση βαρών στις συνεισφορές των γειτόνων, έτσι ώστε οι πιο κοντινοί γείτονες να συνεισφέρουν περισσότερο στον μέσο όρο της τελικής ψήφου από ό,τι οι πιο απομακρυσμένοι. Οι γείτονες λαμβάνονται από ένα σύνολο αντικειμένων για τα οποία είναι γνωστή η κλάση τους και αυτό μπορεί να θεωρηθεί ως το σύνολο εκπαίδευσης για τον αλγόριθμο. Τα παραδείγματα εκπαίδευσης είναι διανύσματα σε έναν πολυδιάστατο χώρο features, το καθένα με μια ετικέτα κλάσης. Η φάση εκπαίδευσης του αλγορίθμου αποτελείται μόνο από την αποθήκευση των διανυσμάτων των features και των ετικετών κλάσης των δειγμάτων εκπαίδευσης. Στη φάση του classification, το k είναι μια σταθερά που ορίζεται από τον χρήστη και ένα διάνυσμα χωρίς ετικέτα ταξινομείται με την ανάθεση της ετικέτας που είναι η πιο συχνή μεταξύ των k δειγμάτων εκπαίδευσης που βρίσκονται πλησιέστερα στο εν λόγω σημείο ερώτησης. Μια ευρέως χρησιμοποιούμενη μετρική απόστασης για συνεχείς μεταβλητές είναι η ευκλείδεια απόσταση. Για διακριτές μεταβλητές, όπως για την ταξινόμηση κειμένου, μπορεί να χρησιμοποιηθεί άλλη μετρική, όπως η απόσταση Hamming [55].

Κεφάλαιο 4 : Προτεινόμενη Αρχιτεκτονική

4.1 Πιλοτική Υλοποίηση εφαρμογής

Υπάρχουν διάφορες εφαρμογές αναγνώρισης εικόνων, οι οποίες όμως λόγω περιορισμένων πόρων, όπως για παράδειγμα η κατάσταση της μπαταρίας, στέλνουν τα αιτήματά τους σε έναν edge server (task offloading). Με αυτό το τρόπο γλιτώνουν χρόνο επεξεργασίας και πόρους. Δημιουργήσαμε αρχικά δύο flavors (εκδόσεις) της ίδιας εφαρμογής , μία “μικρή” και μία “μεγάλη” που αποτελούνται όπως αναφέραμε και παραπάνω από ένα deployment, ένα service και ένα service monitor, οι οποίες έχουν το ίδιο docker image αλλά διαφορετικά labels για να ξεχωρίζουν, καθώς και διαφορετικά requests και limits. Η μία εφαρμογή έχει requests και limits 1Gb Memory και 2 πυρήνες CPU (μικρή), ενώ η άλλη έχει requests και limits 2Gb Memory και 4 πυρήνες CPU (μεγάλη). Η εφαρμογή μας κατά τη λειτουργία της επιστρέφει στον χρήστη μία απάντηση για κάθε εικόνα που τροφοδοτείται , η οποία είναι αρνητική αν η διαδικασία αναγνώρισης απέτυχε ή θετική αν η διαδικασία αναγνώρισης πέτυχε. Τα 2 αυτά flavors στέλνουν με τη σειρά τους στον server τις εικόνες , για να αρχίσει η διαδικασία αναγνώρισης.

Έτσι ο επόμενος στόχος είναι η βέλτιστη λειτουργία του server. Αναλόγως το φορτίο εισόδου με το οποίο τροφοδοτούμε την εφαρμογή μας (πόσα αιτήματα ανά δευτερόλεπτο) θα έχω και αντίστοιχες επιβαρύνσεις στους πόρους τους server (για παράδειγμα υψηλή CPU utilization , μεγάλο χρόνο απόκρισης και υψηλή Memory usage). Στόχος είναι να παρακολουθούμε τις παραπάνω επιβαρύνσεις και με μία διαδικασία reactive autoscaling να ρυθμίζουμε τον αριθμό των replicas της εφαρμογής μας συνεχώς , ώστε να επιτύχουμε τη βέλτιστη λειτουργία του server. Βέλτιστη λειτουργία σημαίνει όχι μόνο να αυξήσουμε τον αριθμό των replicas της εφαρμογής για να ανταπεξέλθει στον αυξανόμενο φόρτο εργασίας , αλλά να είμαστε σε θέση να μειώσουμε και τον αριθμό των replicas αν βλέπουμε πως η εφαρμογή χρησιμοποιεί περισσότερους πόρους από αυτούς που χρειάζεται.

Για να πετύχει αυτό πραγματοποιήθηκαν αρκετά offline πειράματα για το κάθε flavor , τροφοδοτώντας την κάθε εφαρμογή με διαφορετικό φορτίο εισόδου σε κάθε πείραμα , με στόχο να προσδιοριστούν οι “καλές” περιοχές λειτουργίας του κάθε flavor. Πιο συγκεκριμένα, το κάθε πείραμα είχε διάρκεια 30 λεπτών και για κάθε εφαρμογή εξετάζαμε ξεχωριστά τρεις καταστάσεις : 1 replica , 2 replicas , 3 replicas. Ως είσοδο δίναμε στην εφαρμογή μας κάθε φορά, μία κατανομή Poisson της οποίας, το lambda αποτελούσε τον μέσο αριθμό των αιτημάτων ανά 30 δευτερόλεπτα. Έπειτα προσπαθούσαμε για κάθε περίπτωση να βρούμε τρεις διαφορετικές τιμές του lambda; μια υψηλή στην οποία η εφαρμογή μας θα στρεσάρεται , μία μεσαία τιμή στην οποία η εφαρμογή μας θα λειτουργεί ομαλά και μία χαμηλή τιμή στην οποία η εφαρμογή μας θα λειτουργεί με παραπάνω από τους απαραίτητους πόρους. Για να χαρακτηρίσουμε τις παραπάνω περιπτώσεις αξιοποιήσαμε το CPU utilization του συστήματος κατά τη λειτουργία της εφαρμογής μας. Οι τιμές κοντά στο 60% ανήκουν στην πρώτη κατηγορία , οι τιμές κοντά στο 40% ανήκουν στη δεύτερη κατηγορία και οι τιμές κοντά στο 20% ανήκουν στην τρίτη κατηγορία. Από όλα τα πειράματα που πραγματοποιήθηκαν παράχθηκαν πολλά διαφορετικά datasets , τα οποία

ενοποιήθηκαν τελικά σε ένα ενιαίο dataset. Το τελικό dataset καθώς και η διαδικασία εκπαίδευσης με τους διάφορους αλγορίθμους μπορεί να βρεθεί στο παρακάτω google collaboratory:
https://colab.research.google.com/drive/1o5-jJ7D2MxW1FvGuz7pKn0fzq4sUk2uq?usp=sharing.

Για να αξιοποιήσουμε τις πληροφορίες που παράγει ο server χρησιμοποιήσαμε το Prometheus. Εγκαταστήσαμε στο περιβάλλον μας το kube-prometheus (https://github.com/prometheus-operator/kube-prometheus/tree/release-0.9) το οποίο μεταξύ άλλων περιλαμβάνει το Prometheus , το Prometheus Operator, το Prometheus Adapter και τη Grafana. Το Prometheus Operator είναι μια επέκταση του Kubernetes που διαχειρίζεται τις περιπτώσεις παρακολούθησης του Prometheus με έναν πιο αυτοματοποιημένο και αποτελεσματικό τρόπο. Ο Prometheus Operator επιτρέπει να ορίζουμε και να διαχειριζόμαστε τις περιπτώσεις παρακολούθησης ως πόρους του Kubernetes [57]. Ο Prometheus Adapter συλλέγει τα ονόματα των διαθέσιμων μετρικών από τον Prometheus, ανά τακτά χρονικά διαστήματα, και στη συνέχεια εκθέτει μόνο τις μετρικές που ακολουθούν συγκεκριμένες φόρμες [58]. Το kube-prometheus παρέχει υποστήριξη για την χρήση custom metrics και ταυτόχρονα δέχεται και παραμετροποίηση στη διαμόρφωση του. Έπειτα από την εγκατάσταση όλων των παραπάνω δημιουργήσαμε ίδιους κανόνες και για τα δύο flavors στην γλώσσα του Prometheus (PromQL) για να μπορεί να ανιχνεύει τις επιθυμητές μετρικές. Παρακάτω παραθέτω αναφορικά τα ονόματα των κανόνων-μετρικών που χρησιμοποιήθηκαν στην παρακολούθηση των πειραμάτων.

edge_server_art_for_10 : Το art είναι συντομογραφία του average response time, δηλαδή το μέσο χρόνο εξυπηρέτησης του κάθε flavor της εφαρμογής. Το for 10 σημαίνει πως κάθε χρονική στιγμή που η μετρική αυτή επιστρέφει αποτέλεσμα , το αποτέλεσμα αυτό αφορά τη μέση τιμή των χρόνων εξυπηρέτησης των τελευταίων 10 δευτερολέπτων.

edge_server_cpu_cores : To cpu cores σημαίνει πόσους πυρήνες CPU, σε απόλυτο αριθμό, χρησιμοποιεί το κάθε flavor της εφαρμογής.

edge_server_cpu_per_limit_for_60 : Το cpu per limit σημαίνει πόσους πυρήνες CPU χρησιμοποιεί κάθε flavor της εφαρμογής, σε ποσοστιαίες μονάδες, συγκριτικά με το ανώτατο όριο διαθέσιμων πυρήνων CPU του εκάστοτε flavor.

edge_server_cpu_throttling_for_60 : Το cpu throttling σημαίνει ότι τα resources που έχουν δοθεί στο container του εκάστοτε flavor δεν αντιστοιχούν στην τιμή που έχει γίνει request , διότι το CPU είναι κοντά στο limit.

edge_server_memory : Το memory σημαίνει πόση μνήμη, σε απόλυτο αριθμό, καταναλώνει το κάθε flavor της εφαρμογής.

edge_server_memory_per_limit : Το memory per limit σημαίνει πόση μνήμη, σε ποσοστιαίες μονάδες, καταναλώνει το κάθε flavor της εφαρμογής συγκριτικά με το ανώτατο όριο διαθέσιμης μνήμης του εκάστοτε flavor.

edge_server_replicas_count : Το replicas count σημαίνει πόσες ενεργές replicas του κάθε flavor, έχουμε οποιαδήποτε χρονική στιγμή.

edge_server_rr_for_2 : Το rr σημαίνει request rate , δηλαδή το ρυθμό με τον οποίο έρχονται τα αιτήματα στην είσοδο του εκάστοτε flavor.

edge_server_rr_for_30 : Το ίδιο με προηγουμένως απλά αλλάζει το χρονικό διάστημα που εξετάζουμε.

edge_server_total_requests_interval_for_30 : To total requests interval for 30 σημαίνει πόσα ήταν τα συνολικά αιτήματα στο εκάστοτε flavor της εφαρμογής, σε ένα διάστημα 30 δευτερολέπτων.

Για να οπτικοποιήσουμε τις μετρικές που δημιουργήσαν οι κανόνες που γράψαμε χρησιμοποιήσαμε τη Grafana. Συνθέσαμε μία δική μας Grafana Dashboard η οποία μας επιτρέπει την οπτικοποίηση όλων των μετρικών μας ταυτόχρονα. Η Grafana διευκολύνει πολύ την παρακολούθηση της λειτουργίας της εφαρμογής μας και χρησιμεύει στην παρατήρηση μοτίβων αλλά και στην ανάλυση των δεδομένων , καθώς επιτρέπει την αποθήκευση των χρονοσειρών σε αρχεία csv. Για την ανάλυση των δεδομένων των χρονοσειρών χρησιμοποιήσαμε python και κυρίως τη βιβλιοθήκη pandas.

Ένα ακόμα στοιχείο του Kubernetes που χρησιμοποιήσαμε ήταν ο autoscaler. Στο πλαίσιο της εργασίας αναπτύξαμε διάφορους custom pod autoscalers και τους συγκρίναμε μεταξύ τους, αλλά και με τον διαθέσιμο autoscaler των Kubernetes , HPA. Εγκαταστήσαμε τον custom pod autoscaler operator (https://github.com/jthomperoo/custom-pod-autoscaler-operator/blob/master/INSTALL.md) δίνοντας του πλήρη πρόσβαση στο cluster μας. Αυτός είναι ο χειριστής για τη διαχείριση των Custom Pod Autoscalers (CPA) και επιτρέπει να προσθέσουμε τους δικούς μας CPA στο cluster για να διαχειριστούμε όλες τις εφαρμογές που θέλουμε να κάνουμε autoscaling. Στη συνέχεια αρχίσαμε να δημιουργούμε custom docker images για τον cpa χρησιμοποιώντας γλώσσα python και τεχνικές ML.

Οι τρείς προαναφερθείσες κατηγορίες οδήγησαν άμεσα και στην λογική της απόφασης κλιμάκωσης. Για να εφαρμόσω στην πράξη την παραπάνω λογική χρησιμοποίησα μια απλή μαθηματική συνάρτηση για να κατηγοριοποιήσω τα δείγματα των πειραμάτων σε περιοχές -1,0,1 , όπου κάθε περιοχή αντιστοιχούσε και σε μία απόφαση. Παρατίθεται η απλή συνάρτηση που χρησιμοποιήθηκε για την κατηγοριοποίηση των δειγμάτων που λάβαμε από τα πειράματα της “μεγάλης” εφαρμογής (για την “μικρή” αλλάξαμε απλά τις τιμές των περιπτώσεων για να ταιριάζουν τα αποτελέσματα των δεδομένων από τα πειράματά της).

```
def categorise(row):
    if row['Response'] > 400 or row['Throttling'] > 1 or row['Cpu%'] > 60 :
        return int('1')
    elif row['Cpu%'] < 25  :
        return int('-1')
    return int('0')
```

Σχήμα 12: Συνάρτηση κατηγοριοποίησης των δειγμάτων σε περιοχές

Ουσιαστικά αν βρίσκομαι στην κατηγορία στρεσαρίσματος η απόφαση κλιμάκωσης είναι scale out (δηλαδή να αυξήσω τον αριθμό των replicas της εφαρμογής κατά ένα) , αν βρίσκομαι στην ομαλή κατηγορία η απόφαση κλιμάκωσης είναι να μην κάνω τίποτα και αν βρίσκομαι στην κατηγορία πλεονάσματος η απόφαση κλιμάκωσης είναι scale in (δηλαδή να μειώσω τον αριθμό των replicas της εφαρμογής κατά ένα). Προφανώς οι αποφάσεις scale in και scale out περιορίζονται ταυτόχρονα από τα άνω και κάτω όρια των replicas της ίδιας της εφαρμογής.

Για να πετύχουμε περαιτέρω αξιοπιστία στην λήψη της απόφασης κλιμάκωσης χρησιμοποιήσαμε διάφορες τεχνικές μηχανικής μάθησης και επικεντρωθήκαμε κυρίως, στην χρήση του αλγορίθμου Random Forest. Χρησιμοποιήσαμε PCA (Principal Component Analysis) για να προσδιορίσουμε τα πιο σημαντικά features των δειγμάτων και έπειτα πραγματοποιήσαμε τη διαδικασία του Hyperparameter Tuning για να αποφανθούμε για τις βέλτιστες τιμές των παραμέτρων του Random Forest αλγορίθμου. Έπειτα από όλα αυτά εκπαιδεύσαμε το μοντέλο μας στο 70% του συνόλου δεδομένων μας και ελέγξαμε την απόδοση του στο εναπομείναντα 30%. Παραθέτω την ακρίβεια όλων τον μοντέλων που χρησιμοποιήθηκαν κατά την offline εκμάθηση.

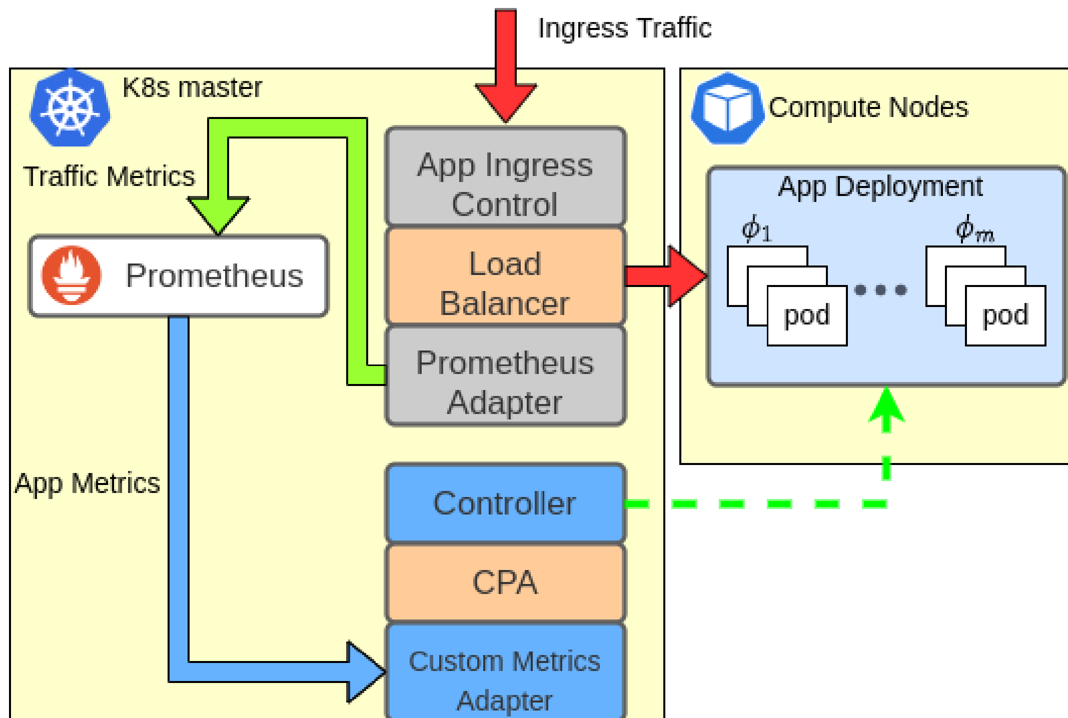
Model	**Accuracy**
Logistic Regression	0.917926
SVM	0.854963
KNN	0.940296
NB	0.889185
Random Forest	0.891704

Πίνακας 1: Ακρίβεια των αλγορίθμων κατά την μάθηση

Έπειτα εισάγαμε το παραπάνω μοντέλο του Random Forest αλγορίθμου καθώς και των υπολοίπων, στον κώδικα του CPA και φτιάξαμε 5 διαφορετικούς autoscaler, καθένας εκπαιδευμένος με ένα διαφορετικό μοντέλο.

Τέλος, δημιουργήσαμε έναν loadbalancer, ο οποίος κατένεμε το φορτίο εισόδου στις δύο εφαρμογές ανάλογα με τα βάρη που του είχαμε δώσει. Συγκεκριμένα το 75% του φορτίου εισόδου κατευθυνόταν στο μεγάλο flavor , ενώ το 25% κατευθυνόταν μέσω του loadbalancer στο μικρό flavor. Έτσι πραγματοποιήσαμε τα τελικά πειράματα στους 5 δικούς μας autoscaler καθώς και στον HPA για ένα συγκεκριμένο φορτίο εισόδου και συγκρίναμε τα αποτελέσματα.

Συνοψίζοντας, το φορτίο εισέρχεται σε μία εξωτερική διεύθυνση ip όπου με τη βοήθεια του loadbalancer γίνεται ανακατεύθυνση στις εσωτερικές θύρες που αντιστοιχίζονται στα pods των δύο flavors της εφαρμογής. Έπειτα το Prometheus συλλέγει τις προαναφερθείσες μετρικές και τροφοδοτεί τα αποτελέσματα στον custom pod autoscaler μέσω του Prometheus adapter. Έπειτα ο controller του custom pod autoscaler, ανάλογα με ποιο μοντέλο μηχανικής μάθησης τον εκπαιδεύσαμε, λαμβάνει μία απόφαση κλιμάκωσης. Έτσι τα pods των δύο flavors κάνουν είτε scale in ή scale out, αναλόγως με την απόφαση του autoscaler.



Σχήμα 13: Προσομοίωση τελικής αρχιτεκτονικής του συστήματος

Κεφάλαιο 5 : Πειραματική Αξιολόγηση

5.1 Πειραματικές Συνθήκες

Για την εκπόνηση αυτής της εργασίας πραγματοποιήσαμε μία πιλοτική υλοποίηση μιας εφαρμογής αναγνώρισης εικόνας , η οποία όπως αναφέραμε προωθεί τα αιτήματά της σε έναν server , σε περιβάλλον containerized εφαρμογών. Η πιλοτική εφαρμογή δοκιμάστηκε σε εγκατάσταση της πλατφόρμας Kubernetes τοπικά σε Linux, έκδοσης Ubuntu 18.04 LTS, χρησιμοποιώντας το Minikube. Το Minikube είναι μια έκδοση του Kubernetes για τοπική εκτέλεση σε υπολογιστή, που λειτουργεί ως μια εικονική μηχανή (VM). Με το minikube δημιουργείται ένα cluster με ένα κόμβο (single-node cluster) σε ρόλο master και worker, με προ-εγκατεστημένο περιβάλλον για docker ώστε να μπορούν να τρέχουν containers μέσα σε αυτό. [56] Για την δημιουργία του κάθε flavor της εφαρμογής χρησιμοποιήσαμε δύο .yaml αρχεία στα οποία ορίσαμε το deployment και το αντίστοιχο service. Ο κώδικας που χρησιμοποιήθηκε στην πορεία αυτής της εργασίας, βρίσκεται διαθέσιμος στο github: https://github.com/spyr0us/DynamicResourceAllocation. Στο .yaml αρχείο που χρησιμοποιούμε για τη δημιουργία της εφαρμογής ορίζουμε τα requests (απαιτήσεις) και limits (όρια) πόρων που είναι απαραίτητα για την ομαλή λειτουργία της εφαρμογής μας. Παράλληλα δίνουμε και ένα μοναδικό label στην εφαρμογή μας για να μπορούμε μετέπειτα να τη συσχετίσουμε με το απαραίτητο service.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: edge-server-cv
  labels:
    app: edge-server-cv
spec:
  replicas: 1
  selector:
    matchLabels:
      app: edge-server-cv
  template:
    metadata:
      labels:
        app: edge-server-cv
    spec:
      containers:
      - image: dspatharakis/controller_druidnet:edge-server
        name: edge-server-cv
        resources:
          requests:
            cpu: "2000m"
            memory: "4000Mi"
          limits:
            cpu: "2000m"
            memory: "4000Mi"
        env:
        - name: EDGE_SERVER_PORT
          value: '8000'
        - name: CPU_LIMIT
          value: '2'
        - name: PROMETHEUS_MULTIPROC_DIR
          value: '/tmp'
        ports:
        - name: es-port
          containerPort: 8000
        imagePullPolicy: Never
```

Σχήμα 14: Το deployment.yaml αρχείο του ενός flavor της εφαρμογής

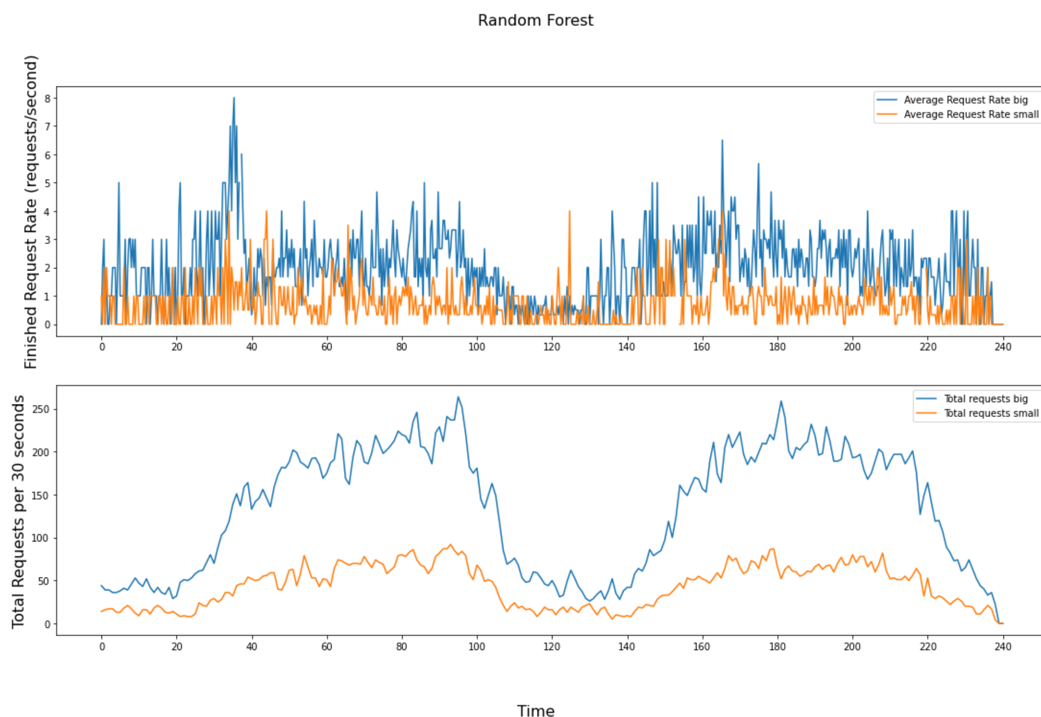
Στο δεύτερο .yaml αρχείο ορίζουμε το service της εφαρμογής καθώς και τις θύρες στις οποίες “ακούει” η εφαρμογή μας. Παράλληλα στο ίδιο αρχείο ορίζουμε και το service monitor της εφαρμογής το οποίο είναι απαραίτητο για να μπορεί το Prometheus να βρίσκει τις μετρικές που παράγει η εφαρμογή μας.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: edge-server-cv
  labels:
    app: edge-server-cv
spec:
  ports:
  - name: edge-server-cv-port
    port: 6004
    nodePort: 30796
    targetPort: 8000
  selector:
    app: edge-server-cv
  type: NodePort
---
kind: ServiceMonitor
apiVersion: monitoring.coreos.com/v1
metadata:
  name: edge-server-cv
  labels:
    app: edge-server-cv
spec:
  selector:
    matchLabels:
      app: edge-server-cv
  endpoints:
  - port: edge-server-cv-port
    interval: 1s
```

Σχήμα 15: Το service.yaml αρχείο του ενός flavor της εφαρμογής

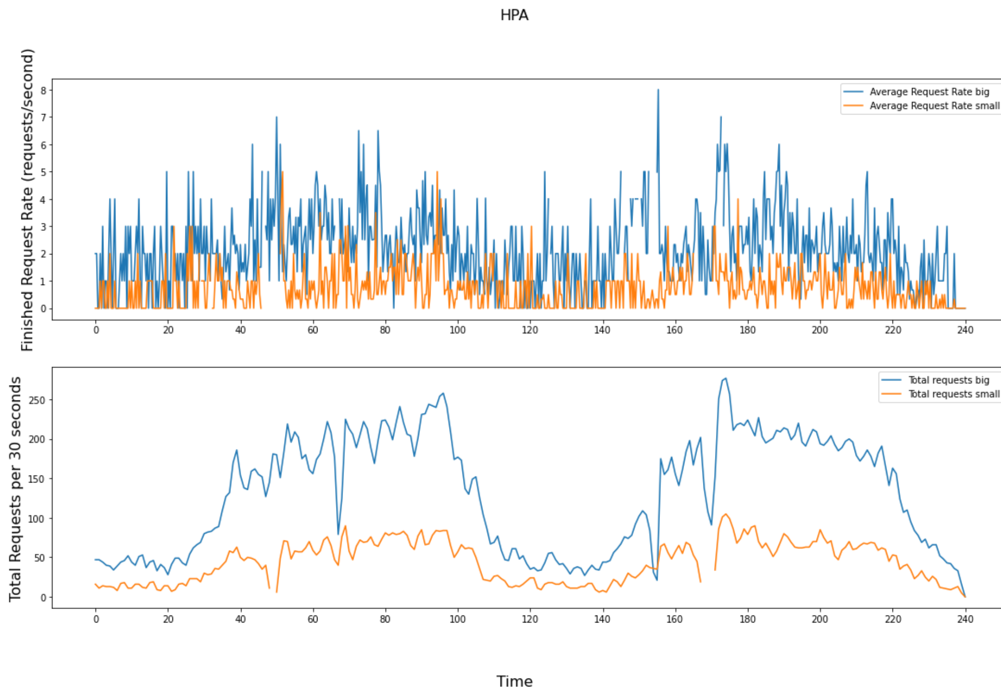
Διαθέτουμε έναν φάκελο με 10 διαφορετικές εικόνες τις οποίες χρησιμοποιεί η εφαρμογή μας για να προσπαθήσει να αναγνωρίσει διάφορα πράγματα. Φτιάχνουμε ένα script σε python το οποίο τρέχουμε για να λειτουργεί η εφαρμογή μας. Το συγκεκριμένο script τροφοδοτεί την εφαρμογή μας με μία τυχαία εικόνα (από τις 10 διαθέσιμες) ανά κάποιο χρονικό διάστημα. Το πετυχαίνει αυτό χρησιμοποιώντας το πρωτόκολλο POST στην κατάλληλη θύρα που έχουμε ορίσει στο service της εφαρμογής. Το φορτίο εισόδου της εφαρμογής είναι άρρηκτα συνδεδεμένο με την χρονική απόσταση των αιτημάτων. Για την υλοποίηση των τελικών πειραμάτων χρησιμοποιήθηκε κάθε φορά το ίδιο φορτίο εισόδου το οποίο όμως λόγω κατανομών Poisson , έχει κάποιες μικρές διαφοροποιήσεις.

5.2 Παρουσίαση αποτελεσμάτων



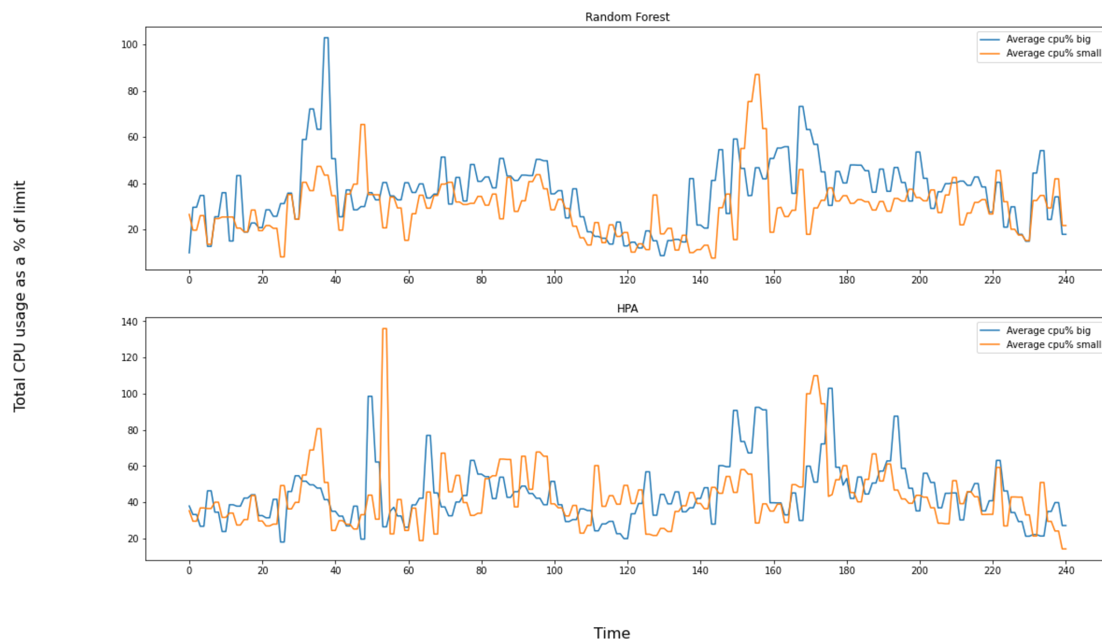
Σχήμα 16: Ρυθμός αιτημάτων στο πείραμα με το Random Forest

Στο παραπάνω σχήμα, που αφορά το πείραμα με τη χρήση του CPA και του Random Forest, παρατηρούμε το ρυθμό με τον οποίο μεταβάλλεται ο αριθμός των αιτημάτων των flavor της εφαρμογής μας. Εύκολα βλέπουμε πως το συγκεκριμένο φορτίο εισόδου αυξάνεται σταδιακά μέχρι το t=90. Στο συγκεκριμένο χρονικό διάστημα το μεγάλο flavor δέχεται περίπου 250 αιτήματα ανά 30 δευτερόλεπτα , ενώ το μικρό γύρω στα 100 αιτήματα ανά 30 δευτερόλεπτα. Έπειτα υπάρχει μία προσωρινή μείωση του ρυθμού αιτημάτων μέχρι το t=130 , όπου εκεί το μεγάλο flavor δέχεται περίπου 40 αιτήματα ανά 30 δευτερόλεπτα , ενώ το μικρό περίπου 30 αιτήματα ανά 30 δευτερόλεπτα. Στη συνέχεια ξαναέχουμε αύξηση του φορτίου εισόδου μέχρι το t=190 όπου από εκεί και έπειτα παρατηρείται σταδιακή μείωση του ρυθμού των αιτημάτων , έως ότου τερματίσει το εκάστοτε πείραμα.



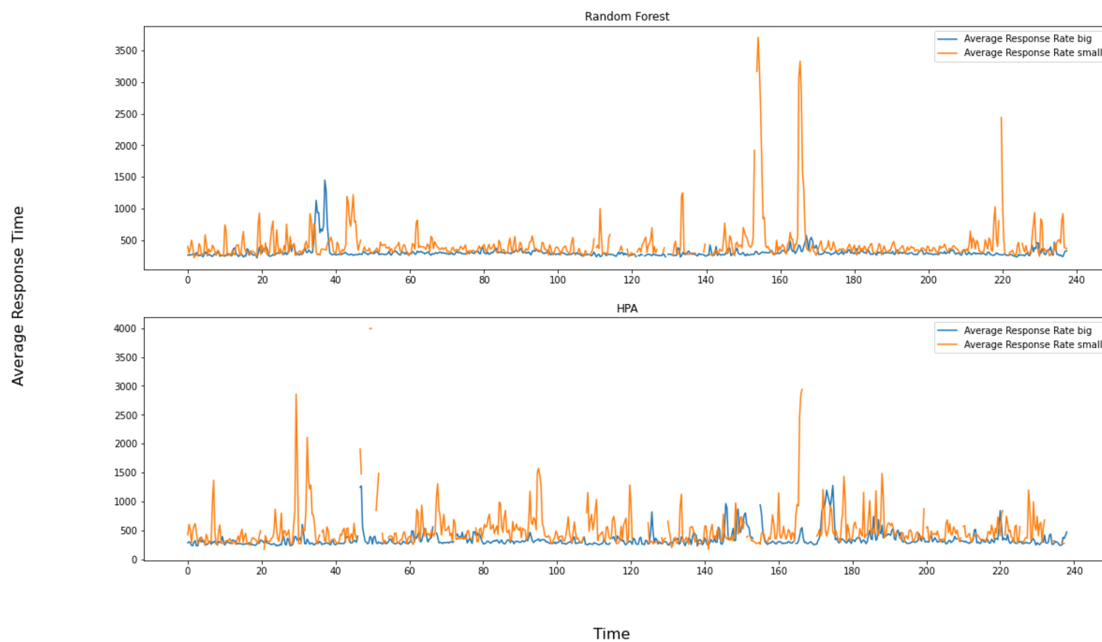
Σχήμα 17: Ρυθμός αιτημάτων στο πείραμα με το HPA

Αντιστοίχως στην περίπτωση του πειράματος που γίνεται χρήση του HPA , βλέπουμε πως το φορτίο εισόδου αυξάνεται και πάλι μέχρι το t=90 και έπειτα μειώνεται μέχρι το t=130. Παρόλα αυτά οι μέγιστες τιμές των αιτημάτων ανά δευτερόλεπτο έχουν κάποιες μικρές διαφορές , τόσο στην τιμή όσο και στο χρονικό διάστημα που λαμβάνουν χώρα.



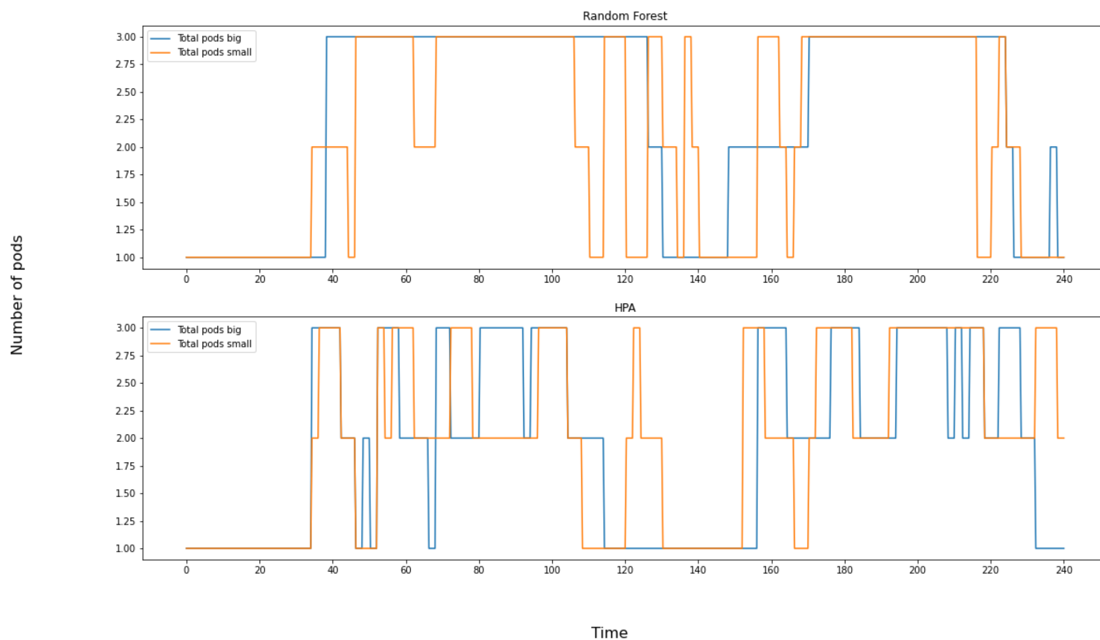
Σχήμα 18: Ρυθμός μεταβολής της κατανάλωσης CPU

Στο παραπάνω διάγραμμα βλέπουμε την δυναμική μεταβολή της κατανάλωσης CPU του συστήματος , που συνδέεται άρρηκτα με τον μεταβαλλόμενο ρυθμό αιτημάτων. Για τον Random Forest βλέπουμε πως στο διάστημα t=35, για το μεγάλο flavor, έχουμε τη μέγιστη τιμή του average requests / second , που ισούται με 8 , και παρατηρούμε πως σε εκείνο το διάστημα έχουμε και μέγιστη τιμή της κατανάλωσης CPU . ίση με λίγο παραπάνω από 100%. Αντίστοιχα παρατηρούμε και στον HPA στο διάστημα t=50 έχουμε 7 αιτήματα ανά δευτερόλεπτο για το μεγάλο flavor που συνοδεύεται από 100% κατανάλωση CPU. Παρόλα αυτά βλέπουμε πως για το μικρό flavor , στον πείραμα του Random Forest και του CPA , η κατανάλωση CPU μεγιστοποιείται γύρω στο 90% στο t=155. Ενώ για το πείραμα με τον HPA , η κατανάλωση CPU του μικρού flavor μεγιστοποιείται γύρω στο 140% στο t=50.



Σχήμα 19: Απόκριση χρόνου εξυπηρέτησης του συστήματος

Στο παραπάνω διάγραμμα βλέπουμε την απόκριση χρόνου εξυπηρέτησης του συστήματος , μετρημένη σε milliseconds. Εύκολα καταλαβαίνουμε πως συνδέεται άμεσα με τις αυξομειώσεις της κατανάλωσης CPU και στα δύο πειράματα. Για παράδειγμα στον πείραμα του Random Forest το μεγάλο flavor κάνει αρνητικό ρεκόρ χρόνου εξυπηρέτησης περίπου 1500 ms στο t=30 , ενώ το μικρό flavor κάνει αντίστοιχο αρνητικό ρεκόρ χρόνου εξυπηρέτησης περίπου 3500 ms στο t=155.



Σχήμα 20: Ρυθμός μεταβολής του αριθμού των replicas κάθε flavor

Στο παραπάνω σχήμα βλέπουμε την δυναμική αλλαγή των replicas του κάθε flavor. Προφανώς οι αλλαγές αυτές συμβαίνουν στις συγκεκριμένες χρονικές στιγμές που ο ρυθμός αιτημάτων της εφαρμογής αυξάνεται ή μειώνεται απότομα, με στόχο το σύστημα να λειτουργεί βέλτιστα κάθε φορά με τους ελάχιστα δυνατούς πόρους.

Τέλος παραθέτουμε πίνακες που περιέχουν τις μέσες τιμές από τους πόρους του συστήματος κατά τη διάρκεια κάθε τελικού πειράματος, τόσο για το μικρό όσο και για το μεγάλο flavor της εφαρμογής. Σημειώνουμε πως για το μικρό flavor της εφαρμογής φαίνεται πως ο CPA με τον Random Forest απέδωσε καλύτερα από τον HPA , καθώς χρησιμοποίησε 0.17 παραπάνω replicas αλλά πέτυχε καλύτερη κατανάλωση CPU και χρήση μνήμης , όπως και καλύτερο χρόνο απόκρισης. Για το μεγάλο flavor της εφαρμογής παρατηρούμε πως ο CPA με τον SVM απέδωσε καλύτερα από τον HPA , καθώς χρησιμοποίησε 0.10 replicas παραπάνω αλλά πέτυχε καλύτερο χρόνο απόκρισης καθώς και λιγότερη κατανάλωση CPU και χρήση μνήμης.

Small Flavor	No. Of Pods	Memory %	Cpu %	Cpu Throttling	Response Time (in ms)	Memory (in MB)	Cpu (in cores)
Random Forest	2.17	39.4	29.7	0.4	434	788	0.3
SVM	1.78	40.6	33.9	0.66	545	812	0.34
KNN	2	37.9	31.4	0.37	432	758	0.31
NB	2.48	37.2	34.6	0.34	434	744	0.35
Regression	2.03	38.4	33.7	0.59	512	768	0.34
HPA	2	39.9	43.2	0.76	520	798	0.43

Πίνακας 2: Μέσες τιμές μετρικών που χρησιμοποιήθηκαν για το small flavor της εφαρμογής

Big Flavor	No. Of Pods	Memory %	Cpu %	Cpu Throttling	Response Time (in ms)	Memory (in MB)	Cpu (in cores)
Random Forest	2.31	25.6	36.3	0.34	308	1024	0.73
SVM	2.1	25.5	35.7	0.32	299	1020	0.71
KNN	2.24	25.5	35.5	0.29	295	1020	0.71
NB	2.73	25.1	37	0.27	301	1004	0.74
Regression	2.32	26.3	35.1	0.28	294	1052	0.70
HPA	1.97	27.1	44.4	0.59	350	1084	0.89

Πίνακας 3: Μέσες τιμές μετρικών που χρησιμοποιήθηκαν για το big flavor της εφαρμογής

Κεφάλαιο 6 : Συμπεράσματα και Μελλοντικές Προεκτάσεις

6.1 Συμπεράσματα

Εύκολα παρατηρούμε πως όσο πιο σύνθετες είναι οι εφαρμογές, τόσο πιο σύνθετα γίνονται και τα προβλήματα κλιμάκωσης για να αντιμετωπιστούν από την μονομερή λογική του HPA. Όλο και περισσότερες cloud infrastracture εταιρείες έχουν αρχίσει να αναπτύσσουν τις δικές τους λογικές κλιμάκωσης, δημιουργώντας ξεχωριστούς autoscalers, για να αντιμετωπίσουν τον εκάστοτε φόρτο εργασίας. Η δική μας αρχιτεκτονική κλήθηκε να αντιμετωπίσει ένα χρονικά μεταβαλλόμενο αριθμό από requests και να επιτύχει καλύτερη διαχείριση πόρων από τον HPA. Όπως είδαμε και στους παραπάνω πίνακες καταφέραμε για λίγο περισσότερα pods, κατά μέσο όρο, να έχουμε σημαντικά χαμηλότερες καταναλώσεις CPU και μνήμης. ’Ετσι συμπεραίνουμε πως οι autoscaling λύσεις με reactive λογικές, βασισμένες σε αλγορίθμους μηχανικής μάθησης, έχουν καλή απόδοση στην αντιμετώπιση δυναμικών φορτίων σε περιβάλλον Kubernetes.

6.2 Μελλοντικές Προεκτάσεις

Μέσω της διερεύνησης ποικίλων τρόπων αυτόματης κλιμάκωσης εφαρμογών στο πλαίσιο της παρούσας διπλωματικής, προέκυψαν ανάγκες και ιδέες για πιθανές επεκτάσεις στο μέλλον. Καθίσταται επιτακτική η διενέργεια δοκιμών της προτεινόμενης αρχιτεκτονικής σε πραγματικό Kubernetes cluster, μεγαλύτερης κλίμακας από το minikube, πραγματοποιώντας δοκιμές με πολυπλοκότερο φορτίο εισόδου και λαμβάνοντας μεγαλύτερο dataset. Με αυτό τον τρόπο θα επιτευχθεί μια συνολικότερη σύγκριση με τον προκαθορισμένο αλγόριθμο του HPA σε πιο απαιτητικές καταστάσεις. Τέλος, αν επιθυμούσαμε να αναπτύξουμε περισσότερο την παραπάνω προτεινόμενη αρχιτεκτονική για να πετύχουμε ακόμα χαμηλότερες καταναλώσεις, θα μπορούσαμε να εφαρμόσουμε μια Reinforcement Learning λογική στον αλγόριθμο μηχανικής μάθησης. Με αυτόν τον τρόπο θα λαμβάναμε ανά τακτά χρονικά διαστήματα feedback για την απόφαση κλιμάκωσης, σε πραγματικό χρόνο, το οποίο θα χρησιμοποιούνταν μελλοντικά για να βελτιώσει τις επόμενες αποφάσεις κλιμάκωσης.

Βιβλιογραφία

[1] B. Lutkevich and K. Goulart, “cloud-native application,” TechTarget, Jul. 26, 2021.
[2] Δημήτρης Γ. Τριχινάς, (20 Νοεμβρίου 2012). «Μελέτη και ανάπτυξη διαδικτυακής υπηρεσίας για την επίβλεψη νέφους αποθήκευσης δεδομένων»

[3] M.C. Huebscher and J.A. McCann. A survey of autonomic computing: Degrees, models, and applications. ACM Comp Surveys, 40(3):7, 2008
[4] X. Dutreilh, A. Moreau, J. Malenfant, N. Rivierre, and I. Truck. From data center resource allocation to control theory and back. In IEEE 3rd International Conference on Cloud Computing, pages 410–417, 2010.
[5] R. Han, L. Guo, M.M. Ghanem, and Y. Guo. Lightweight resource scaling for cloud applications. In 12th International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pages 644–651, 2012.
[6] M.Z. Hasan, E. Magana, A. Clemm, L. Tucker, and S.L.D. Gudreddi. Integrated and autonomic cloud resource scaling. In Network Operations and Management Symposium (NOMS), pages 1327–1334, 2012.
[7] T.C. Chieu, A. Mohindra, A.A. Karve, and A. Segal. Dynamic scaling of web applications in a virtualized cloud computing environment. In IEEE Intl Conf on e-Business Engineering, pages 281–286, 2009
[8] P. Jamshidi, C. Pahl, and N. C. Mendona. Managing uncertainty in autonomic cloud elasticity controllers. IEEE Cloud Computing, 3(3):50–60, 2016.
[9] Q. Zhu and G. Agrawal. Resource provisioning with budget constraints for adaptive applications in cloud environments. In ACM Intl Symp on High Performance Distributed Computing, pages 304–307, 2010.
[10] A. Ali-Eldin, M. Kihl, J. Tordsson, and E. Elmroth. Efficient provisioning of bursty scientifific workloads on the cloud using adaptive elasticity control. In Workshop on Scientifific Cloud Computing Date, 2012.
[11] A. Ali-Eldin, J. Tordsson, and E. Elmroth. An adaptive hybrid elas ticity controller for cloud infrastructures. In Network Operations and Management Symposium (NOMS), pages 204–212. IEEE, 2012.
[12] P. Padala, K.-Y. Hou, K.G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In ACM Europ Conf on Computer systems, pages 13–26, 2009
[13] E. Kalyvianaki, TheT.mistoklis Charalambous, and S. Hand. Self adaptive and self-confifigured cpu resource provisioning for virtualized servers using kalman fifilters. In Proceedings of the 6th international conference on Autonomic computing, pages 117–126. ACM, 2009.
[14] J. Huang, C. Li, and J. Yu. Resource prediction based on double exponential smoothing in cloud computing. In Intl Conf on Consumer Electronics, Communications and Networks, pages 2056–2060, 2012.
[15] H. Mi, H. Wang, G. Yin, Y. Zhou, D. Shi, and L. Yuan. Online self reconfifiguration with performance guarantee for energy-efficient large scale cloud computing data centers. In Services Computing (SCC), 2010 IEEE International Conference on, pages 514–521. IEEE, 2010.

[16] N. Roy, A. Dubey, and A. Gokhale. Effificient autoscaling in the cloud using predictive models for workload forecasting. In Intl Conf on Cloud Computing (CLOUD), pages 500–507, 2011.
[17] Z. Gong, X. Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In Network and Service Management (CNSM), 2010 International Conference on, pages 9–16. IEEE, 2010.
[18] R.S. Sutton and A.G. Barto. Reinforcement learning: An introduction. MIT press Cambridge, 1998.
[19] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, and I. Truck. Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflflow. In International Conference on Autonomic and Autonomous Systems, pages 67–74, 2011
[20] G. Tesauro, N.K. Jong, R. Das, and M.N. Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In IEEE Intl Conference on Autonomic Computing, pages 65–73, 2006.
[21]Adan, Ivo, and Jacques Resing. "Queueing theory." (2002): 104-106.
[22]M. A. Kaboudan, “A dynamic-server queuing simulation,” Comput. Oper. Res., vol. 25, no. 6, pp. 431–439, 1998.
[23]V. V. Mazalov and A. Gurtov, “Queuing system with on-demand number of servers,” Mathematica Applicanda, vol. 40, no. 2, pp. 1–12, 2012.
[24]Y.-H. Jia, L.-X. Tang, Z. G. Zhang, and X.-F. Chen, “MMPP/M/C queue with congestion-based staffifing policy and applications in oper ations of steel industry,” Springer J. Iron Steel Res. Int., vol. 26, no. 7, pp. 659–668, 2018.
[25]M. Wajahat, A. Gandhi, A. A. Karve, and A. Kochut, “Using machine learning for black-box autoscaling,” in Proc. IEEE 7th Int. Green Sustain. Comput. Conf. (IGSC), 2016, pp. 1–8.
[26]J. Rahman and P. Lama, “Predicting the end-to-end tail latency of containerized microservices in the cloud,” in Proc. IEEE Int. Conf. Cloud Eng. (IC2E), 2019, pp. 200–210.
[27]G. Yu, P. Chen, and Z. Zheng, “Microscaler: Automatic scaling for microservices with an online learning approach,” in Proc. IEEE Int. Conf. Web Serv. (ICWS), 2019, pp. 68–75.
[28]K. Rzadca et al., “Autopilot: Workload autoscaling at Google,” in Proc. 15th EuroSys Conf., 2020, pp. 1–16.
[29]S. Li, D. C. Wunsch, E. A. O’Hair, and M. G. Giesselmann, “Using neural networks to estimate wind turbine power generation,” IEEE Trans. Energy Convers., vol. 16, no. 3, pp. 276–282, Sep. 2001.
[30]J. P. S. Catalão, S. J. P. S. Mariano, V. M. F. Mendes, and L. A. F. M. Ferreira, “Short-term electricity prices forecasting in a com petitive market: A neural network approach,” Elect. Power Syst. Res., vol. 77, no. 10, pp. 1297–1304, 2007.
[31]G. Chen, “Energy-aware server provisioning and load dispatching for connection-intensive Internet services,” in Proc. 5th USENIX Symp. Netw. Syst. Design Implement. (NSDI), 2008, pp. 337–350.
[32]Z. Gong, X. Gu, and J. Wilkes, “PRESS: PRedictive elastic resource scaling for cloud systems,” in Proc. IEEE 6th Int. Conf. Netw. Serv. Manag. (CNSM), 2010, pp. 9–16.
[33]S. Islam, J. Keung, K. Lee, and A. Liu, “Empirical prediction models for adaptive resource provisioning in the cloud,” Elsevier Future Gener. Comput. Syst., vol. 28, no. 1, pp. 155–162, 2012.

[34]V. R. Messias, J. C. Estrella, R. Ehlers, M. J. Santana, R. C. Santana, and S. Reiff-Marganiec, “Combining time series prediction models using genetic algorithm to autoscaling Web applications hosted in the cloud infrastructure,” Springer Neural Comput. Appl., vol. 27, no. 8, pp. 2383–2406, 2016.
[35]“What is a Container?,” Docker. https://www.docker.com/resources/what-container.
[36]“Production-Grade Container Orchestration,” Kubernetes. https://kubernetes.io/
[37]“Kubernetes Components,” Kubernetes. https://kubernetes.io/docs/concepts/overview/components/
[38]“Pods,” *Kubernetes*. https://kubernetes.io/docs/concepts/workloads/pods/
[39]https://kubernetes.io/docs/concepts/services-networking/service/
[40]https://spot.io/resources/kubernetes-autoscaling-3-methods-and-how-to-make-them-great/
[41]https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
[42]https://custom-pod-autoscaler.readthedocs.io/en/latest/
[43]https://custom-pod-autoscaler.readthedocs.io/en/latest/user-guide/getting-started/
[44]https://prometheus.io/docs/introduction/overview/
[45]https://github.com/prometheus-operator/prometheus-operator
[46]https://grafana.com/docs/grafana/latest/introduction/
[47]https://docs.python.org/3/tutorial/index.html
[48]https://docs.python.org/3/tutorial/modules.html
[49]https://pandas.pydata.org/docs/getting_started/overview.html
[50]https://www.dataquest.io/blog/sci-kit-learn-tutorial/
[51]Breiman, Leo. "Random forests." *Machine learning* 45.1 (2001): 5-32
[52]Draper, Norman R., and Harry Smith. *Applied regression analysis*. Vol. 326. John Wiley & Sons, 1998.
[53]Noble, William S. "What is a support vector machine?." *Nature biotechnology* 24.12 (2006): 1565-1567.
[54]Rish, Irina. "An empirical study of the naive Bayes classifier." *IJCAI 2001 workshop on empirical methods in artificial intelligence*. Vol. 3. No. 22. 2001.
[55]Guo, Gongde, et al. "KNN model-based approach in classification." *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, Berlin, Heidelberg, 2003.
[56]https://minikube.sigs.k8s.io/docs/start/
[57]https://github.com/prometheus-operator/prometheus-operator
[58]https://github.com/kubernetes-sigs/prometheus-adapter