Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Υπολογιστών και Πληροφορικής

Resource-aware container orchestration on Fog Computing environments

Διπλωματική Εργασία

Βασίλειος Κ. Μιχαλακόπουλος

Επιβλέπων Καθηγητής

Δημήτριος Σούντρης
Καθηγητής

Αθήνα, Μάιος 2021

Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Υπολογιστών και Πληροφορικής

Resource-aware container orchestration on Fog Computing environments

Διπλωματική Εργασία

Βασίλειος Κ. Μιχαλακόπουλος

Επιβλέπων Καθηγητής

Δημήτριος Σούντρης
Καθηγητής

Εγκρίθηκε από την εξεταστική τριμελή επιτροπή στις 25 Μαρτίου 2021

.....	.....	.....
Δημήτριος Σούντρης	Γεώργιος Γκούμας	Διονύσιος Πνευματικάτος
Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Copyright © Βασίλειος Κ. Μιχαλακόπουλος,2021
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.

.....

Βασίλειος Κ. Μιχαλακόπουλος

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

Περίληψη

Τη σημερινή εποχή, τα "containers" είναι πολύ διαδεδομένα κυρίως λόγω της ευελιξίας που τα χαρακτηρίζει. Επιπλέον, υπάρχει ένας συνεχώς αυξανόμενος αριθμός ισχυρών φορτίων και δεδομένων που ωθούνται στο νέφος. Παράλληλα με αυτόν τον αριθμό, αυξάνονται και οι συσκευές του Διαδικτύου των Πραγμάτων (IoT), καθιστώντας τις κλασσικές υπηρεσίες νέφους μη βιώσιμες. Έτσι, γεννιέται μια καινούργια αρχιτεκτονική ονόματι "Fog Computing", αξιοποιώντας την υπολογιστική δύναμη των συσκευών στην άκρη του δικτύου, βοηθώντας και επεκτείνοντας το νέφος.

Για να δουλέψει αποδοτικά αυτό το μοντέλο, πρέπει να κληρονομηθούν τεχνικές διαχείρισης από το νέφος. Τα "containers", όντας πιο αποδοτικά και ελαφριά, χρησιμοποιούνται για να φέρουν την λογική των "microservices" σε αυτή την αρχιτεκτονική. Όμως, παρά το γεγονός ότι η επεξεργαστική δύναμη και οι πόροι των συσκευών στην άκρη του δικτυού αυξάνονται, δεν μπορεί ακόμα να συγκριθεί με αυτή των συσκευών του νέφους. Συνεπώς, μια προσέγγιση που εστιάζει περισσότερο στην σωστή διαχείριση τον πόρων πρέπει να χρησιμοποιηθεί, ώστε το "Fog Computing" να γίνει πιο αποδοτικό.

Σε αυτή την διπλωματική εργασία, παρουσιάζουμε έναν ενορχηστρωτή πακέτων, ειδικά σχεδιασμένο για συσκευές στην άκρη του δικτύου. Ενσωματώνουμε τη λύση μας με τους Κυβερνήτες, που είναι η κατά κόρον χρησιμοποιούμενη εφαρμογή ενορχήστρωσης σε περιβάλλοντα νέφους. Επίσης, μια υπερσύγχρονη πλατφόρμα ονόματι "KubeEdge", ειδικά σχεδιασμένη για συσκευές στην άκρη του δικτύου, χρησιμοποιείται. Τέλος, δείχνουμε ότι μπορούμε να πετύχουμε υψηλότερη ταχύτητα εξυπηρέτησης των εφαρμογών από τις συσκευές, χρησιμοποιώντας λιγότερους πόρους και κατά συνέπεια λιγότερη ενέργεια για μια πληθώρα εφαρμογών Νευρωνικών Δικτύων.

Λέξεις Κλειδιά: Διαχείριση πόρων, Κυβερνήτες , KubeEdge, Ενορχήστρωση πακέτων, Διαδίκτυο των Πραγμάτων, resource-aware, Fog Computing.

Abstract

These days, containers are extremely popular, mostly due to their isolated, scalable and versatility nature. Furthermore, there is an ever-increasing number of Machine Learning (ML) and Artificial Intelligence (AI) workloads driven to the cloud, in the form of microservices/containers. This number has augmented critically, in parallel with the number of Internet of Things (IOT) devices/sensors, slowly making Cloud services untenable. Thus, emerges a new computing paradigm named "Fog Computing", which leverages computing at the Edge, assisting and extending the Cloud.

In order for this model to work, techniques and experience must be inherited from the Cloud. Containers, being the most efficient and light form of virtualization, bring microservices logic to the Edge. Although, despite the fact that Edge devices are getting more powerful by the day, their computing power and resources cannot compare to the ones at the Cloud. Thus, a more resource-aware approach must come and alter the existing techniques, for Fog Computing to be optimized.

In this Thesis, we present a resource aware container orchestrator, specifically designed for Aarch64 devices located at the Edge. We integrate our solution with Kubernetes, one of the most widely used cloud orchestration frameworks nowadays. Also, a state-of-the-art Edge framework which is directly connected to the Kubernetes is used, named KubeEdge. We show that our custom scheduler can achieve better Quality of Service (QoS) whilst using fewer resources and thus less power, for a variety of ML workloads.

Keywords: Fog Computing, Edge Computing, IoT, Multivariable Polynomial Regression, Scheduling, Kubernetes, KubeEdge, resource-aware, Aarch64, resource management, MLPerf

Ευχαριστίες

Φτάνοντας εδώ θα ήθελα να ευχαριστήσω τα άτομα τα οποία μου στάθηκαν και με βοήθησαν όλον αυτό τον καιρό. Αρχικά, θα ήθελα να ευχαριστήσω τον καθηγητή Δημήτρη Σούντρη ο οποίος ήταν υπεύθυνος για τη διπλωματική μου εργασία. Με τον τρόπο του με ώθησε στο να γίνω πιο παραγωγικός και να καταλάβω τι σημαίνει έρευνα στη πράξη. Επίσης, θα ήθελα να ευχαριστήσω το εργαστήριο Microlab για τους πόρους που διέθεσε. Πιο συγκεκριμένα, ευχαριστώ τους υποψήφιους διδάκτορες Δημοσθένη Μασούρο και Μανώλη Κατσαραγάκη για τον χρόνο τους και βεβαίως για την πολύτιμη βοήθεια, υπομονή και καθοδήγησή τους.

Ένα τεράστιο ευχαριστώ, στους ανθρώπους που συνέβαλλαν στο να γίνουν τα φοιτητικά χρόνια το πιο όμορφο και ποικιλόχρωμο ταξίδι της ζωής μου, μέχρι στιγμής. Τέλος, ένα ευχαριστώ στην οικογένεια μου για την ανιδιοτελή εμπιστοσύνη και αγάπη τους. Αφιερωμένο στις γιαγιάδες μου Αγγελικούλα και Ελένη.

Περιεχόμενα

Κεφάλαιο 1

Εκτεταμένη Ελληνική Περίληψη

Τα τελευταία χρόνια, η ραγδαία ανάπτυξη της τεχνολογίας φέρνει όλο και περισσότερες ηλεκτρονικές συσκευές στην καθημερινότητα μας. Οι περισσότερες εξαυτών είναι συνεχώς συνδεδεμένες στο διαδίκτυο και προσφέρουν στον χρήστη μια πληθώρα εφαρμογών. Ως εκ τούτου, υπάρχει η ανάγκη για ένα νέο πρότυπο στην επικοινωνία Machine2Machine που επιτρέπει τη συνδεσιμότητα των "Πραγμάτων" στο Παγκόσμιο Διαδίκτυο. Αυτό το παράδειγμα είναι γνωστό με τον όρο IoT.

Το Internet of Things (IoT) είναι ένα δίκτυο φυσικών αντικειμένων, συσκευών, οχημάτων, κτιρίων αλλά και άλλων αντικειμένων τα οποία περιέχουν ενσωματωμένα ηλεκτρονικά συστήματα, λογισμικά, αισθητήρες και διαδικτυακή δυνατότητα σύνδεσης – κάτι που επιτρέπει σε αυτά τα αντικείμενα να συλλέγουν και να ανταλλάσσουν δεδομένα. Το IoT δίνει τη δυνατότητα στα αντικείμενα αυτά να ελέγχονται εξ'αποστάσεως μέσω της υπάρχουσας δικτυακής υποδομής δημιουργώντας ευκαιρίες άμεσης ενσωμάτωσης του φυσικού κόσμου με τα υπολογιστικά συστήματα έχοντας ως αποτέλεσμα τη βελτίωση της αποτελεσματικότητας και της ακρίβειας αλλά και τη μείωση του κόστους. Από την στιγμή μάλιστα που το IoT εξοπλίζεται με αισθητήρες αποτελεί μέρος έξυπνων συστημάτων της καθημερινότητας όπως είναι τα έξυπνα σπίτια, οχήματα και πόλεις. Κάθε αντικείμενο αναγνωρίζεται μοναδικά από το ενσωματωμένο υπολογιστικό σύστημα και μπορεί να λειτουργεί τόσο αυτόνομα όσο και σε συνεργασία με την υπόλοιπη διαδικτυακή υποδομή.

Ως συνέπεια, δημιουργείται καθημερινά ένας τεράστιος όγκος δεδομένων που το παρόν μοντέλο του cloud computing δε μπορεί να διαχειριστεί αποδοτικά. Ακόμα, η ασφάλεια και η ταχύτητα επικοινωνίας με το νέφος, γίνεται όλο και πιο δύσκολη όσο πληθαίνουν οι χρήστες και η κλίμακα της γεοκατανομής των συσκευών αυξάνεται. Προσθέτοντας σε αυτά τα προβλήματα, οι απαιτήσεις , από άποψη πόρων και καθυστέρησης, των εφαρμογών έχουν αυξηθεί σε πολύ μεγάλο βαθμό με τις εφαρμογές πλέον να χρησιμοποιούν κατά κύριο λόγω ισχυρά μοντέλα Νευρωνικών Δικτύων και Τεχνητής Νοημοσύνης.

Τη λύση σε αυτά τα προβλήματα, του μοντέλου νέφους, έρχεται να λύσει μια καινούργια πλατφόρμα ονόματι Fog Computing. Αυτή η αρχιτεκτονική υπολογιστικής ομίχλης , όπως αποκαλείται στη διεθνή βιβλιογραφία, εκμεταλλεύεται τους πλέον αρκετούς υπολογιστικούς πόρους στην άκρη του δικτύου, με την επεξεργασία των δεδομένων να συμβαίνει ακριβώς εκεί που παράγονται. Όπως γίνεται κατανοητό, τα προβλήματα καθυστέρησης και

απόδοσης λόγω απόστασης και γεοκατανομής των συσκευών με τα κέντρα νέφους, απαλείφεται.

Πάραυτα προκύπτουν προβλήματα, καθώς οι συσκευές στην άκρη του δικτύου είναι πολύ πιο περιορισμένες, από άποψη πόρων, από αυτές του νέφους. Επίσης, μιας και υπάρχει κάθε είδους συσκευή στην άκρη του δικτύου (τηλεοράσεις, κινητά, ψυγεία, αυτοκίνητα κτλ.) η ανάγκη για εικονοποίηση των εφαρμογών σε αυτό το περιβάλλον είναι επιτακτική. Η παρούσα τεχνική που χρησιμοποιεί εικονικές μηχανές (Virtual Machines) δε θα μπορούσε να δουλέψει σε ένα τέτοιο περιβάλλον λόγω των υψηλών απαιτήσεων της σε μνήμη καθώς και επεξεργαστική δύναμη. Συνεπώς, επιλέγεται μια καινούργια τεχνική, ονόματι Containerization.

Όπως αναφέραμε, τα Containers είναι ένα τρόπος εικονοποίησης μιας εφαρμογής. Το κύριο χαρακτηριστικό τους, που τα κάνει να επιλέγονται από τα Virtual Machines, είναι κυρίως η ελαφρότητα και η ευελιξία τους. Πιάνουν πολύ λιγότερο χώρο και είναι πολύ πιο γρήγορα και αποδοτικά. Επιπρόσθετα, λειτουργούν πάνω στον πυρήνα του συστήματος του οικοδεσπότη και χρησιμοποιούν απευθείας πόρους και βιβλιοθήκες του συστήματος, χωρίς κάποιο επιπλέον στρώμα λογισμικού. Έτσι έρχεται και η λογική των microservices στο Fog Computing, επεκτείνοντας το νέφος. Όπως αναφέραμε και νωρίτερα όμως, το πλήθος των εφαρμογών είναι εξαιρετικά μεγάλο και σχεδόν αδύνατο να το διαχειριστεί κάποιος. Επομένως, δημιουργήθηκαν δομές για container orchestration, με κορυφαία εξαυτών τους Κυβερνήτες (Kubernetes).

Οι Κυβερνήτες είναι ένας ενορχηστρωτής πακέτων που χρησιμοποιείται κατά κόρον σε όλα τα υπολογιστικά συστήματα. Η αναγνώριση από τον κόσμο του προγραμματισμού καθώς και οι δυνατότητες του, μοιάζουν απεριόριστες. Είναι σχεδιασμένοι για να διαχειρίζονται τεράστιο πλήθος συσκευών καθώς και εφαρμογών. Με άλλα λόγια, είναι ειδικά σχεδιασμένοι για περιβάλλοντα νέφους. Αυτό το χαρακτηριστικό, δημιουργεί ένα μειονέκτημα σε περιβάλλοντα Fog Computing, όπου οι συσκευές είναι πιο περιορισμένες από άποψη πόρων. Το KubeEdge, είναι ένα πιο ελαφρύ εργαλείο που χρησιμοποιεί τις βασικές λειτουργίες των κυβερνητών, αλλά είναι ειδικά σχεδιασμένο για το Edge, προσπαθώντας να λύσει κάποια από αυτά τα προβλήματα.

Ένα μεγάλο πρόβλημα λοιπόν που απασχολεί μια πληθώρα ερευνητών, είναι η σωστή ενορχήστρωση πακέτων σε συσκευές που βρίσκονται στην άκρη του δικτύου. Όπως αναφέραμε, το KubeEdge, αν και ειδικά σχεδιασμένο για το Edge, χρησιμοποιεί τη διαδικασία ενορχήστρωσης των Κυβερνητών, η οποία είναι σχεδιασμένη για πιο δυνατές συσκευές. Το κύριο αντικείμενο μελέτης λοιπόν, είναι να βρεθεί η κατάλληλη τεχνική ώστε η ενορχήστρωση πακέτων σε αδύναμες συσκευές από άποψη πόρων, να γίνει πιο αποδοτική.

1.1 Η Συνεισφορά μας

Αρχικά, δημιουργήσαμε έναν ενορχηστρωτή ο οποίος σε πραγματικό χρόνο γνωρίζει για τις συνθήκες κάποιων μετρικών στις συσκευές και προβλέπει τις μετρικές κατά τη διάρκεια εκτέλεσης. Στη συνέχεια, προβλέπει την επιλεγμένη μετρική απόδοσης (Queries per Second) και επιλέγει τη συσκευή που προβλέπεται να έχει υψηλότερη απόδοση στη συγκεκριμένη περίπτωση. Αυτό επιτυγχάνεται με ένα διπλό στρώμα Polynomial Regression. Δείχνουμε ότι αυτή η προσέγγιση είναι πιο αποδοτική σε περιβάλλοντα Fog, καθώς επιτυγχάνεται μεγαλύτερη απόδοση κατά τη διάρκεια εκτέλεσης με σημαντικά λιγότερους πόρους σε σχέση με την προκαθορισμένη.

1.2 Πειραματικό Περιβάλλον

Το σύστημα που δημιουργήθηκε για την αξιολόγηση του δρομολογητή μας αποτελείται από μια εικονική μηχανή και δύο Edge συσκευές. Οι δύο αυτές συσκευές χρησιμοποιούνται ως κόμβοι εργάτες στον Κυβερνήτη, ενώ η εικονική μηχανή, όντας η πιο δυνατή, χρησιμοποιείται ως κόμβος διαχειριστής. Επίσης, είναι πολύ σημαντικό να αναφερθεί ότι οι τρείς συσκευές διαφέρουν και σε αρχιτεκτονική επεξεργαστή. Οι μεν δύο Edge συσκευές χρησιμοποιούν την αρχιτεκτονική Aarch64 ή Arm64, ενώ η εικονική μηχανή στηρίζεται σε μηχάνημα που τρέχει στο κλασσικό x86_64. Η βασική τους διαφορά είναι ότι οι Arm επεξεργαστές είναι τύπου RISC (Reduced Instruction Set Computer) και επομένως πολύ πιο ταιριαστοί και διαδεδομένοι σε ενσωματωμένα συστήματα.

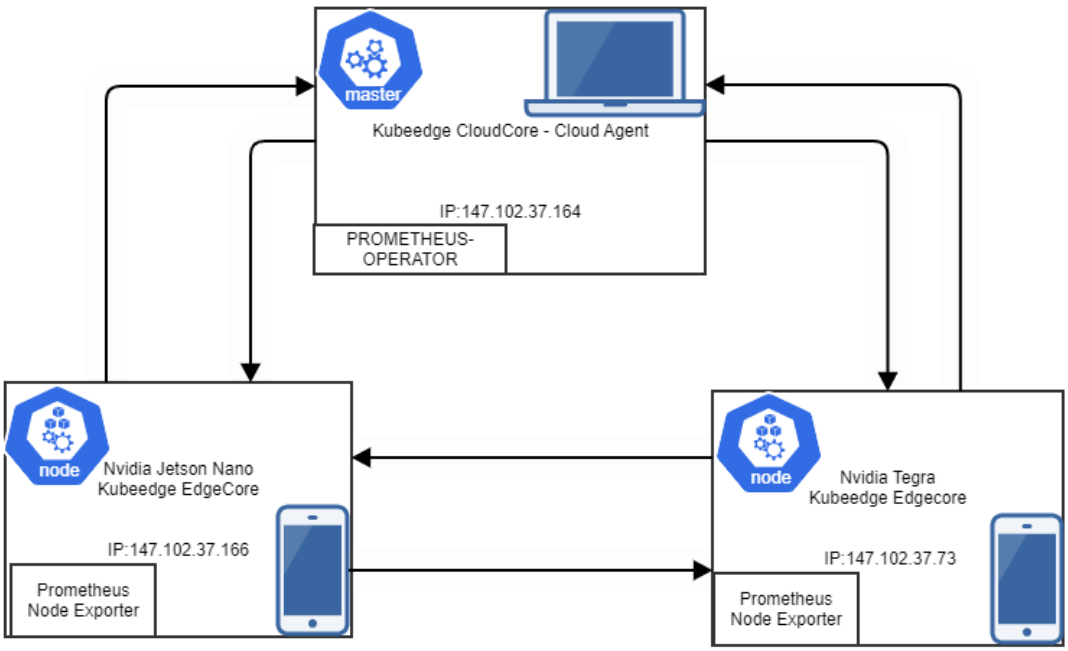
Στη συνέχεια ενσωματώσαμε στον Κυβερνήτη το τελευταίας τεχνολογίας Edge framework το KubeEdge. Το συγκεκριμένο εργαλείο όπως αναφέρθηκε προηγουμένως, είναι ειδικά σχεδιασμένο για συσκευές με περιορισμένους πόρους. Χρησιμοποιεί έναν πράκτορα πολύ πιο ελαφρύ για να επικοινωνεί με αυτές τις συσκευές. Επίσης, συνδέεται άμεσα με τον Κυβερνήτη, με τον πράκτορα νέφους που διαθέτει, και παρέχει στο χρήστη το ίδιο API (Application Programming Interface) και τις ίδιες λειτουργικότητες.

Για την εξαγωγή μετρικών από τις συσκευές χρησιμοποιείται ένα ευρέως διαδεδομένο πρόγραμμα παρακολούθησης συσκευών, ο Προμηθέας (Prometheus). Στην ουσία ο Προμηθέας είναι μια βάση δεδομένων για χρονοσειρές, οι οποίες λαμβάνονται από τις συσκευές ανά τακτά χρονικά διαστήματα. Στη συνέχεια μέσω της γλώσσας PromQL, ο χρήστης μπορεί να διαχειριστεί τα δεδομένα και να πάρει τις μετρικές που τον ενδιαφέρουν. Οι μετρικές αυτές χρησιμοποιούνται αρχικά για την πρόβλεψη των ίδιων μετρικών

κατά τη διάρκεια εκτέλεσης, αλλά και για την πρόβλεψη της μετρικής απόδοσης της κάθε εφαρμογής.

Το τελευταίο και αναγκαίο συστατικό των πειραμάτων ήταν οι εφαρμογές. Πλέον, η πληθώρα των εφαρμογών οι οποίες χρησιμοποιούνται σε κάθε στάδιο Computing, είναι πλέον Τεχνητής Νοημοσύνης ή και Μηχανικής Μάθησης. Για αυτό τον λόγο, επιλέξαμε τη σουίτα συμπερασμάτων(inference) της MLPerf. Αυτό το υποσύνολο της σουίτας έχει ως στόχο να μετρήσει το πόσο γρήγορα και αποδοτικά κάποια συστήματα επεξεργάζονται εισόδους και παράγουν αποτελέσματα χρησιμοποιώντας ένα εκπαιδευμένο μοντέλο. Οι εφαρμογές που χρησιμοποιήσαμε αφορούν τα πεδία image classification και object detection. Κάθε workload αποτελείται από διαφορετικά Inference Engines που χρησιμοποιούν διαφορετικά μοντέλα, σύνολα δεδομένων, backend (ONNX Runtime, Pytorch, Tensorflow κ.λ.π) και προκαθορισμένο σενάριο.

Το περιβάλλον που δημιουργήθηκε εν τέλει, είναι όπως φαίνεται στο Σχήμα 1.2 και οι εφαρμογές παρουσιάζονται πιο αναλυτικά στον Πίνακα 1.2:



Σχήμα 1.2: Το Σύστημά μας.

Area	Task	Model	Dataset	Quality	Backend	Accuracy
Vision-Heavy	Image Classification	Resnet50-v1.5	ImageNet (224x224)	fp32	Tensorflow	76.456%
Vision-Light	Image Classification	Mobilenet-v1 224	ImageNet (224x224)	fp32	Tensorflow	71.676%
Vision-Light	Object Detection	Ssd-Mobilenet-v1	COCO(300x300)	fp32	Tensorflow	mAP 0.23

Πίνακας 1.2: MLPerf εφαρμογές.

1.3 Η Υλοποίησή μας

Η διαδικασία που χρησιμοποιείται για τη δρομολόγηση των εφαρμογών στην ουσία, μιμείται σε δομή αυτή του Kube-Scheduler. Αρχικά, υπάρχει ένας "Listener", ο οποίος εντοπίζει εισερχόμενα Pod (εφαρμογές) στο default namespace και όλους τους κόμβους και τις ενημερώσεις τους. Από τη στιγμή που εντοπιστεί ένα Pod έτοιμο να δρομολογηθεί εισέρχεται σε μια ουρά προτεραιότητας. Η προτεραιότητα των εφαρμογών εξαρτάται από τις ήδη υπάρχουσες εφαρμογές στην ουρά. Με άλλα λόγια, υλοποιήσαμε μια ουρά FIFO. Στη συνέχεια, η εφαρμογή με τη χαμηλότερη προτεραιότητα εξέρχεται -δηλαδή η εφαρμογή που εισήλθε πρώτη στην ουρά- και εισέρχεται στην διαδικασία δρομολόγησης.

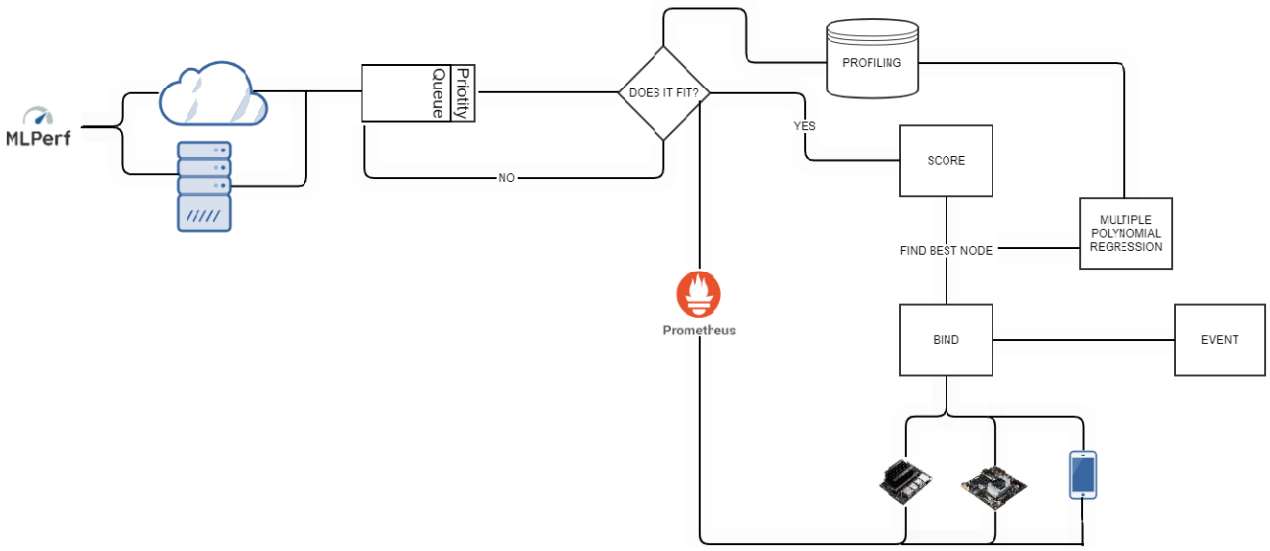
Ο κύκλος αυτός, ξεκινά με την αξιολόγηση των συσκευών, ώστε ο δρομολογητής μας να κρίνει, εάν οι συσκευές είναι κατάλληλες για να σηκώσουν το φορτίο ή όχι. Αρχικά, ελέγχεται άμα οι συσκευές είναι διαθέσιμες στο cluster μας μέσω κλήσης στο API. Στη συνέχεια, μέσω του Προμηθέα κοιτάμε εάν οι συσκευές έχουν χρήση $CPU < \alpha\%$, $Memory < \beta\%$ και $Δίσκος \geq \gamma\ Gb$. Ύστερα από αρκετά πειράματα, καταλήξαμε ότι οι κατάλληλες τιμές για αυτές τις μεταβλητές στην περίπτωση μας, είναι : $\alpha = \beta = 80$ και $\gamma = 2$. Όποιοι κόμβοι πληρούν αυτές τις προϋποθέσεις μεταβαίνουν στο επόμενο στάδιο της δρομολόγησης, το Scoring. Εκεί βρίσκεται και το βασικό κομμάτι του αλγορίθμου μας.

Τώρα ο δρομολογητής καλείται να αποφασίσει ποια εκ των φιλτραρισμένων συσκευών, θα αποδώσει καλύτερα με τη συγκεκριμένη εφαρμογή. Αρχικά, ανάλογα την αίτηση μνήμης της εφαρμογής την κατατάσσει σε μια κατηγορία. Οι κατηγορίες προκύπτουν από τις ήδη δοκιμασμένες εφαρμογές όπως φαίνονται στον Πίνακα 1.2. Στην ουσία υπάρχουν τρείς κατηγορίες, μία για κάθε εφαρμογή και χαρακτηρίζονται από έναν αριθμό, την

αίτηση μνήμης της εφαρμογής της κατηγορίας. Το ζευγάρι με την μικρότερη ευκλείδεια απόσταση επιλέγεται. Στη συνέχεια, ανάλογα με την κατηγορία, έχοντας ως είσοδο τρείς μετρικές, την χρήση CPU , την χρήση της μνήμης και την θερμοκρασία της CPU, προβλέπει τις τιμές αυτών των μετρικών κατά τη διάρκεια εκτέλεσης της εφαρμογής. Αυτή η πρόβλεψη στην ουσία είναι ένα Single Variable Polynomial Regression που κάθε φορά έχει ως εξαρτώμενη μεταβλητή την τιμή της μετρικής κατά τη διάρκεια εκτέλεσης και ως ανεξάρτητη την τιμή της μετρικής πριν την δρομολόγηση. Τώρα αφού έχουμε προβλέψει τις τιμές των μετρικών κατά τη διάρκεια εκτέλεσης, πρέπει να υποθέσουμε ποια εκ των φιλτραρισμένων συσκευών είναι η πλέον καταλληλότερη για την εκτέλεση της εφαρμογής. Η τεχνική που χρησιμοποιείται σε αυτό το στάδιο, ονομάζεται Multivariable Polynomial Regression και με βάση τις τιμές των τριών μετρικών προβλέπει το τελικό QPS της εφαρμογής στην κάθε συσκευή. Η μετρική QPS, ολογράφως Queries per Second, είναι στην ουσία μια μετρική που δείχνει την καθυστέρηση για την εκτέλεση της εφαρμογής και πιο γενικά την απόδοση της εφαρμογής.

Τέλος, η συσκευή που επιλέγεται είναι αυτή που εν δυνάμει θα έχει τον μεγαλύτερο αριθμό QPS και άρα την καλύτερη απόδοση. Μετά η εφαρμογή δένεται με την επιλεγμένη συσκευή στο cluster και ένα event (γεγονός) εκπέμπεται ώστε ο χρήστης να μπορεί να ακολουθήσει την πορεία της ζωής της εφαρμογής. Απαραίτητο είναι να σημειωθεί ότι ο δρομολογητής μας ζητάει από τον χρήστη την τοποθεσία του συστήματος αποθήκευσης του Docker στις συσκευές και την απαίτηση μνήμης της κάθε εφαρμογής.

Τα παραπάνω στάδια δρομολόγησης μπορούμε να τα δούμε και στο Σχήμα 1.3 :



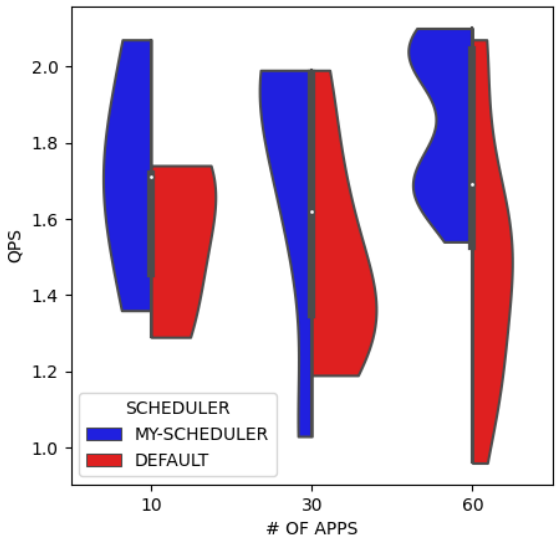
Σχήμα 1.3: Ο δικός μας κύκλος δρομολόγησης.

1.4 Αποτελέσματα και Αξιολόγηση

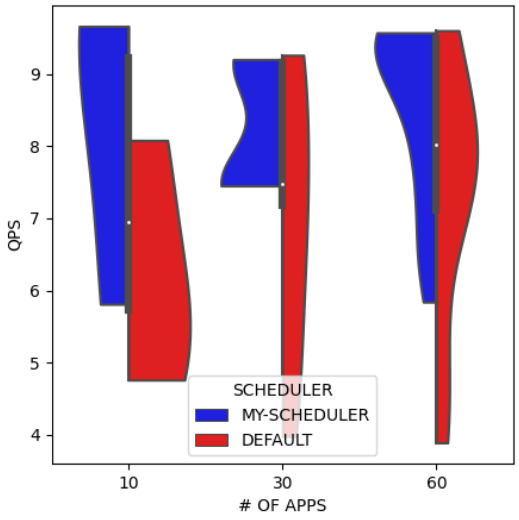
Για την αξιολόγηση του δρομολογητή, δημιουργήσαμε μια σουίτα σε Python και Bash και εκτελέσαμε διάφορα πειράματα. Τα πειράματα διέφεραν από άποψη έντασης και ποικιλίας εφαρμογών. Σε κάθε πείραμα αλλάζαμε το workload που αποτελούταν κάθε φορά από διαφορετικό αριθμό εφαρμογών. Οι εφαρμογές προέκυπταν τυχαία. Επίσης, όπως είπαμε, τα πειράματα διέφεραν και από άποψη έντασης. Αυτό προκύπτει από τους διαφορετικούς χρόνους ανάμεσα στις αφίξεις δύο εφαρμογών. Με πιο απλά λόγια, κάθε πείραμα σαν είσοδο έπαιρνε μια πληθώρα εφαρμογών, διαφορετικές μεταξύ τους, και ένα χρονικό διάστημα (π.χ. 2-5 λεπτά) το οποίο συμβολίζει τον ελάχιστο και τον μέγιστο χρόνο που μπορούμε να περιμένουμε ώστε να στείλουμε μια εφαρμογή για δρομολόγηση.

Οι εφαρμογές στην ουσία είναι ένα από τα inference engines του Πίνακα 1.2. Για να δοκιμαστούν στις συσκευές δημιουργήσαμε κάποια docker images τα οποία πακετάραμε μέσα σε κάποια Kubernetes Jobs, αντίστοιχα. Τα Jobs με τη σειρά τους, αντιστοιχούνται σε ένα Pod, τα οποία είναι ο κατεξοχήν τρόπος εκτέλεσης εφαρμογών σε περιβάλλοντα Κυβερνητών. Οι μετρικές απόδοσης που επιλέχθηκαν είναι , από άποψη απόδοσης τα QPS (Queries per Second) κάθε εφαρμογής και από άποψη κατανάλωσης πόρων ο μέσος όρος χρησιμοποίησης CPU και μνήμης κατά τη διάρκεια εκτέλεσης, καθώς και η θερμοκρασία της CPU.

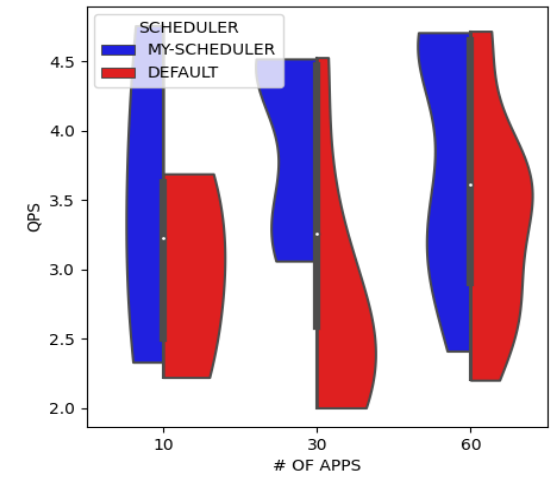
Τα πειράματα που θα περιγράψουμε αποτελούνται από δέκα, τριάντα και εξήντα εφαρμογές αντίστοιχα, ενώ όλα είναι για χρονικό διάστημα δύο με πέντε λεπτών. Τα παρακάτω σχήματα είναι διαχωρισμένα ανά εφαρμογή και περιγράφουν την κατανομή των QPS, σε κάθε πείραμα. Όπως φαίνεται στο Σχήμα 1.4.1, για το inference engine που στηρίζεται στο μοντέλο Resnet, ο δικός μας δρομολογητής καταφέρνει, στην πληθώρα των περιπτώσεων, να βελτιώσει την καθυστέρηση κάθε εφαρμογής. Αντίστοιχα φαίνεται στο Σχήμα 1.4.2 για το inference engine που στηρίζεται στο μοντέλο Mobilenet και στο Σχήμα 1.4.5 για το Ssd-Mobilenet. Η βελτίωση των συνολικών QPS του κάθε workload, είναι κάθε φορά της τάξης του 25%.



Σχήμα 1.4.1: Κατανομή QPS για το inference engine με το μοντέλο Resnet.

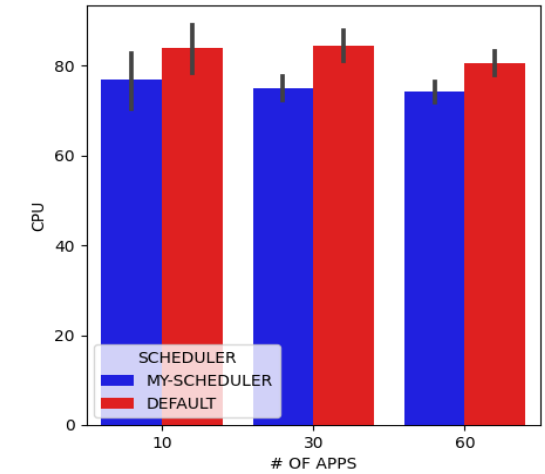


Σχήμα 1.4.2: Κατανομή QPS για το inference engine με το μοντέλο Mobilenet

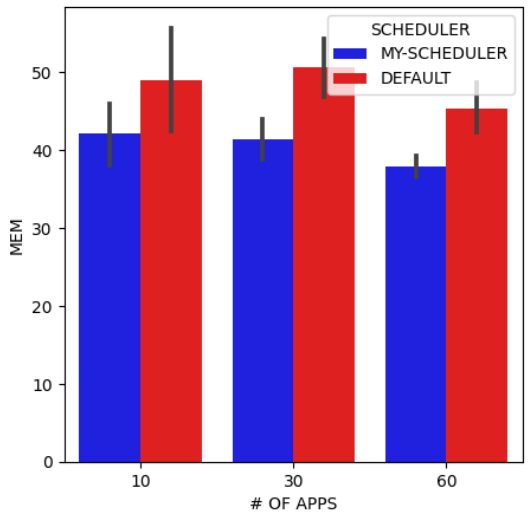


Σχήμα 1.4.3: Κατανομή QPS για το inference engine με το μοντέλο Ssd-Mobilenet.

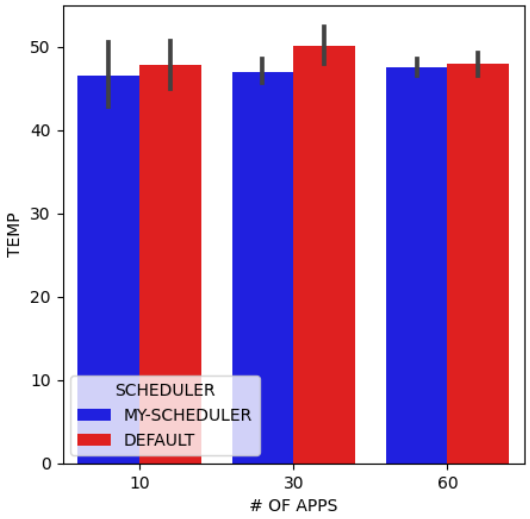
Τώρα για τα ίδια πειράματα θα δείξουμε τη μέση τιμή της χρήσης CPU , τη μέση τιμή χρήσης της μνήμης καθώς και την μέση τιμή των θερμοκρασιών της CPU. Όπως φαίνεται στο Σχήμα 1.4.4, έχουμε μια μείωση στη μέση τιμή, της τάξης του 10% , στη χρήση της CPU στα πειράματά μας. Την ίδια βελτίωση, της τάξης του 10%, παρατηρούμε και στη χρήση της μνήμης στο Σχήμα 1.4.5. Τέλος, υπάρχεο μια μείωση της τάξης των $2°C$ στη μέση τιμή των θερμοκρασιών, όπως φαίνεται στο Σχήμα 1.4.6. Η τάξη των δύο βαθμών Κελσίου ίσως δε φαίνεται τόσο σημαντική, αλλά υπενθυμίζουμε ότι σημαίνει αρκετά σημαντική μείωση στην ισχύ που καταναλώνουν οι συσκευές κατά τη διάρκεια των πειραμάτων.



Σχήμα 1.4.4: Μέση τιμή χρήσης της CPU.



Σχήμα 1.4.5: Μέση τιμή χρήσης της μνήμης.



Σχήμα 1.4.4: Μέση τιμή των θερμοκρασιών στην CPU.

Chapter 2

Introduction

2.1 Internet of Things

To begin with, Internet of Things (IoT) has become one of the most rapidly increasing technologies of the 21st century. Now that we can connect everyday objects – kitchen appliances, cars, thermostats, cameras, pills – to the internet via embedded devices, seamless communication is possible between people, processes and things. Simply put, IoT is a network interface of dedicated physical objects (things) that contain embedded technology to communicate and sense or interact with their internal stages or external environment in order to reach a common goal. So, it is quite a simple concept, it means taking all the things in the world and connecting them to the Internet.

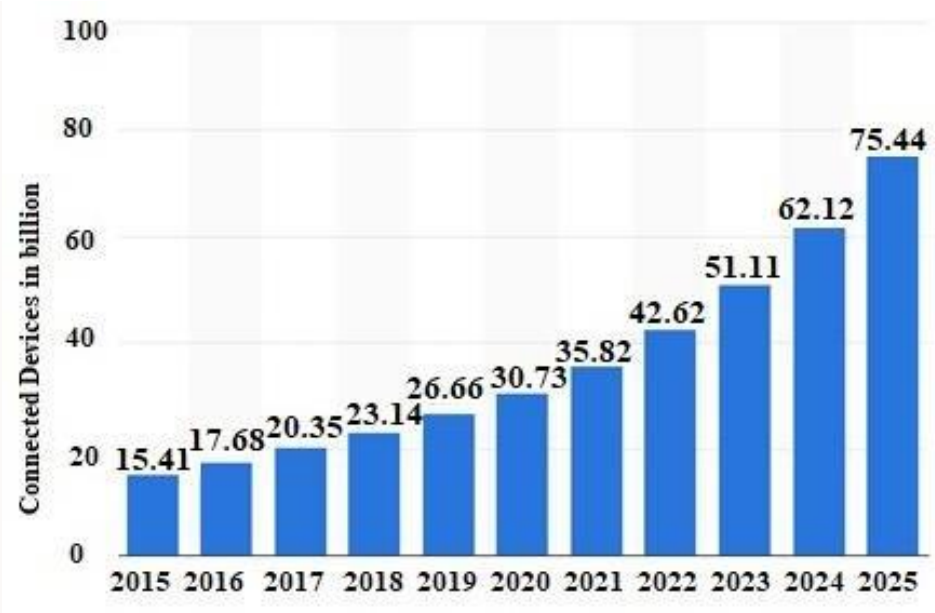


Fig 2.1: Evolution of IoT devices. [1]

Taking a look back, we can see the exponential growth of IoT devices, as shown in Fig 2.1. As the years go by, more and more suchlike devices will become a reality, producing enormous amounts of data in need to be processed. All these data

and the way that these can be managed, bring along questions and challenges to be advised.

Challenges in IoT:

- Privacy & Security: Security is one the biggest issues within IoT. These sensors are collecting in many cases extremely sensitive data – what you say or do in your own home, for example. Keeping that secure is vital to consumer trust, but so far the IoT's security track record has been extremely poor.

- Latency & Bandwidth: Cloud infrastructures are physically located far away from where the data is produced. So, large amounts of data cannot travel all at once via the current ways.

- Quality of Service (QoS): Customers are expecting high latency and throughput to their devices, an expectation hard to meet.

- Compatibility: New waves of technology often feature a large stable of competitors jockeying for market share, and IoT is certainly no exception.

These problems cannot be addressed by the common cloud computing techniques. For industrial and academic purposes, new computing paradigms have emerged to deal with these challenges such as Fog and Edge Computing.

2.2 Edge Computing

Edge computing is a networking philosophy focused on bringing computing as close to the source of the data as possible in order to reduce latency and bandwidth utilization. Simply put, Edge computing's main objective is the decongestion of Cloud infrastructures; by providing an intensive part of the computation locally. Bringing computation to the network's edge minimizes the amount of long-distance communication that has to happen between a client and a server.

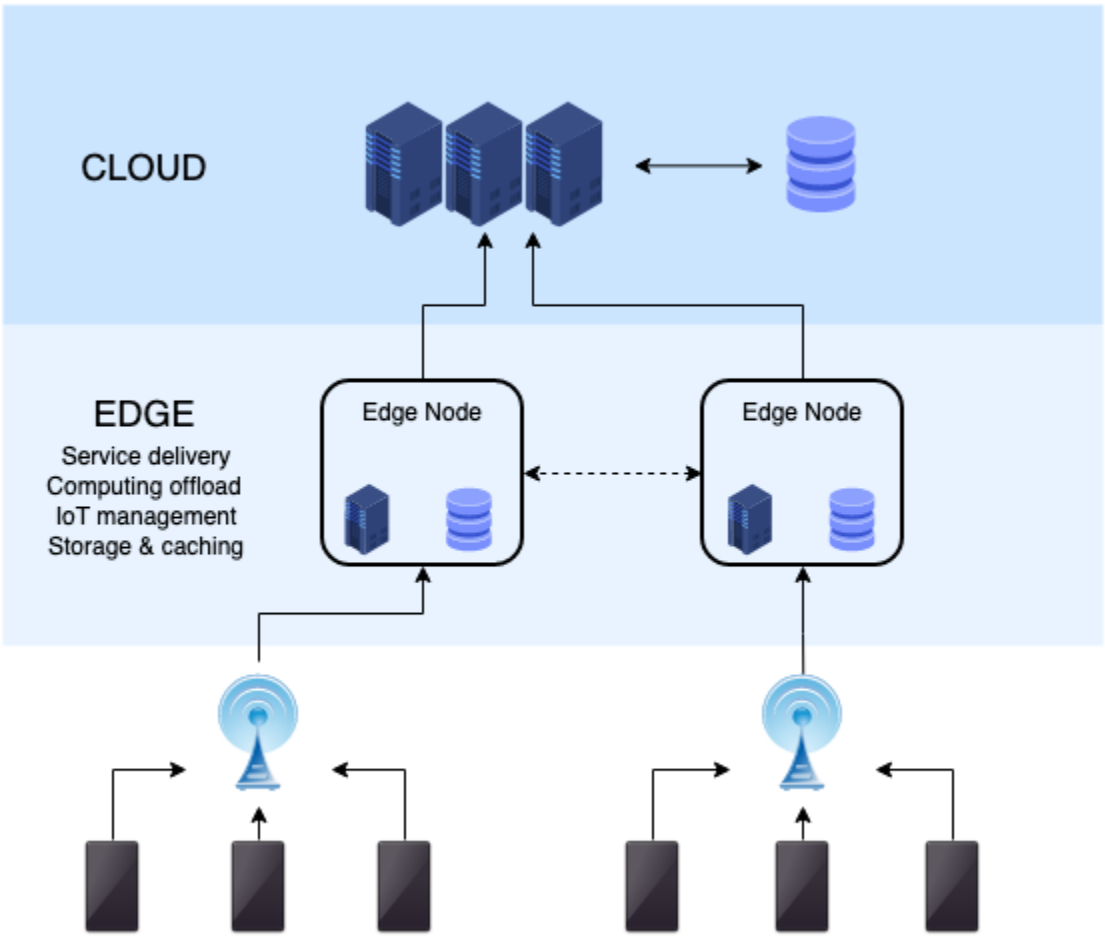


Fig 2.2 Edge Computing architecture. [27]

As shown in Fig 2.2, Edge computing's main goal is to relief the cloud servers by taking a chunk of the computational load needed. As a technological and computational paradigm, edge computing may be characterized as autonomous, distributed computing.

Edge Computing comes with limitations on the technologies and platforms used. So, everything used at the edge must be designed, developed and configured specifically for edge computing purposes.

As mentioned the target of edge computing is to solve the problems that comes with the increase of Internet of Things devices connected. So, the main concept is that any application or functionality will be running closer to the place where embedded technologies interact with the physical world. Edge Computing uses no centralized Cloud, but it uses a similar architecture closer to the Edge.

The benefits of Edge Computing to industry are quite big as QoS (Quality of Service) has been increased and Cost has been decreased as well. The main pros are mentioned below:

- Heterogeneity
- Support to mobiles
- Low Energy Consumption
- Reduces Latency
- Saves bandwidth
- Real time interaction
- Security and Privacy
- Increased QoS
- Decreased Cost

These benefits still remain a challenge and their composition creates space for a huge amount of research. Edge Computing is still trying to improve and new paradigms are emerging to create a whole new aspect of distributed systems.

2.2 Fog Computing

Fog computing may be seen as an architecture that uses edge devices to carry out a bit amount of storage, computation and communication locally. It is an in-between layer to the Cloud and the Edge. Similar to Edge Computing, Fog brings computational power and data mining capabilities closer to the location of the end-user.

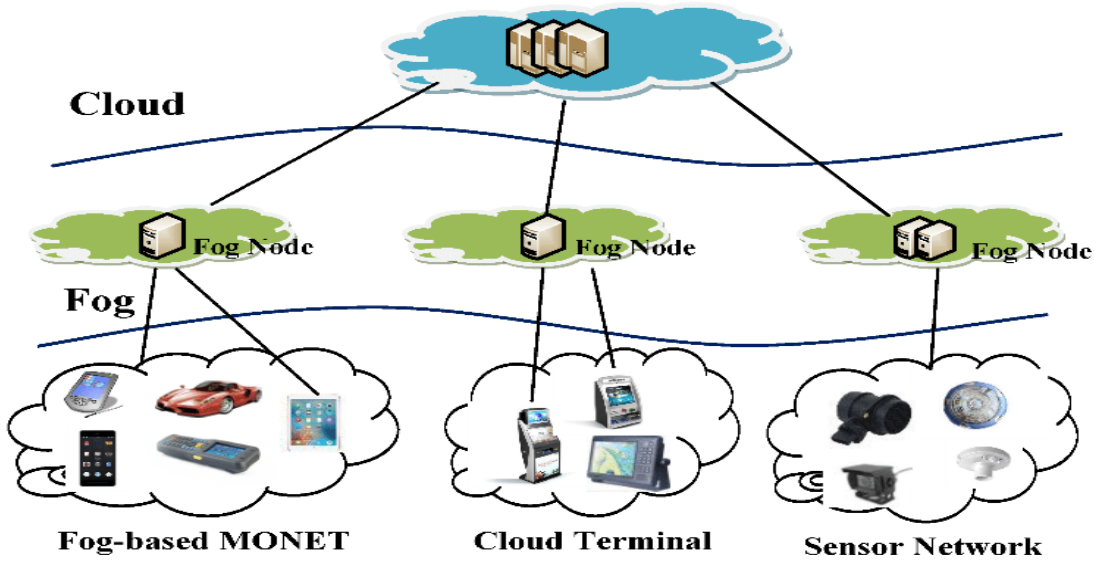


Fig 2.3 Three layer Fog Computing Architecture [28]

The three layers are as shown in Fig 2.3:

- Cloud Layer: The cloud computing layer consists of multiple and different high-performance devices with great storage capabilities. It is the strongest of the three layers when it comes to computing power and storage.
- Fog Layer: It consists of fog nodes which include routers, gateways, switchers and specific embedded devices. Fog nodes provide end-users with storage and computational power. It is closer to a group of Edge devices, so every fog node will attend to the geographically closer devices. Then, all the fog nodes will communicate in a non-centralized way and ask—if necessary, the Cloud servers for computing resources.
- Edge layer: This is the layer closest to the edge and it consists of all different kinds of IoT devices. They are geographically distributed and the ones that are close will communicate with one of the Fog nodes. Even though this layer has almost no computing and storage power, Fog nodes will come to assist these devices, taking the computing workload to the layer above.

2.3 Virtualization and Deployment

In the previous sections, Fog and Edge computing paradigms have been introduced. In this kind of architectures, someone might wonder how these applications will be packaged and deployed in such a heterogeneous environment. The answer is through Virtualization.

Virtualization relies on software to simulate hardware functionality, such as computing and storage resources, and create a virtual computing system in top of an already existing one. Therefore, it provides the opportunity to run more than one virtual systems, with even different operating systems in the same physical device.

There are two ways for someone to perform virtualization and will be described below.

2.3.1 Virtual Machines

Virtual Machines is the most widely used way of deploying an application to a device that you know nothing about. The architecture behind every VM is as shown below.

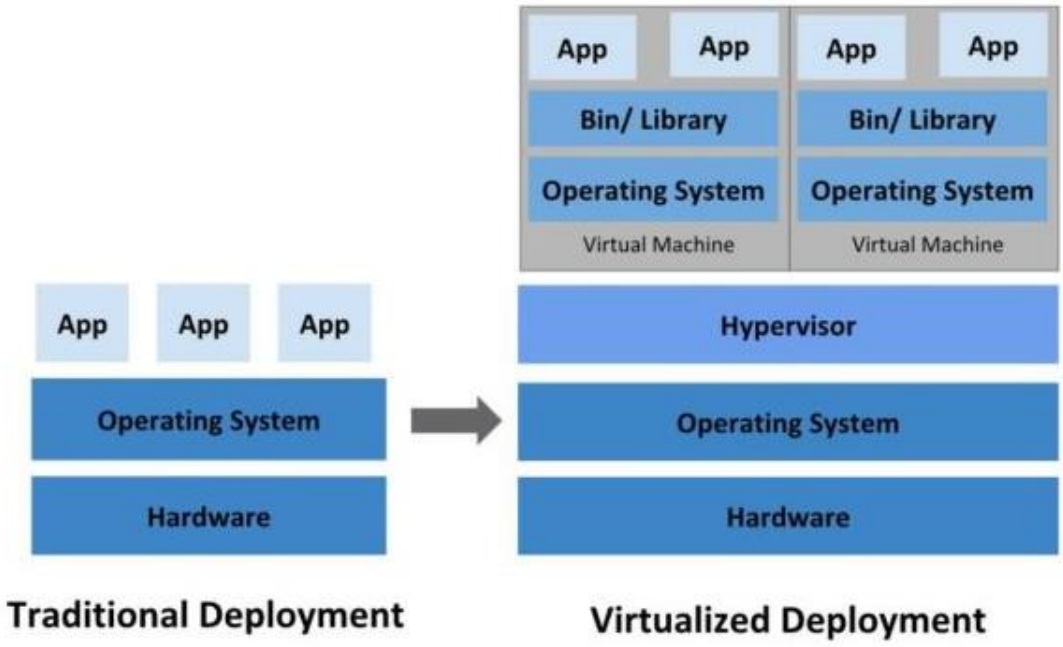


Fig 2.4.1 Traditional Deployment Vs Virtual Machines [29]

Applications require their Libraries and executables in order to run as designed in a completely different system than host machine they were initially developed and tested. This is achieved through containers as seen above. They have a completely different OS (Operating System), and in that OS you can package your application and be sure that it will run as promised everywhere.

Each VM has a hypervisor, or a virtual machine monitor, that is the only thing that sits between the hardware and the VM and its necessary to virtualize the server.

Furthermore, each VM has its own libraries, binaries and applications and the VM may be many GB's in size. This may raise significant problems in a constrained resource environment.

2.3.2 Containers

In recent years, using containers as a virtualization method has become quite popular to all kinds of users. Containers –named after the well-known containers from the shipping industry- are a solution to the problem of how to get software-apps to run reliable and as designed, when moved from one computing environment to another.

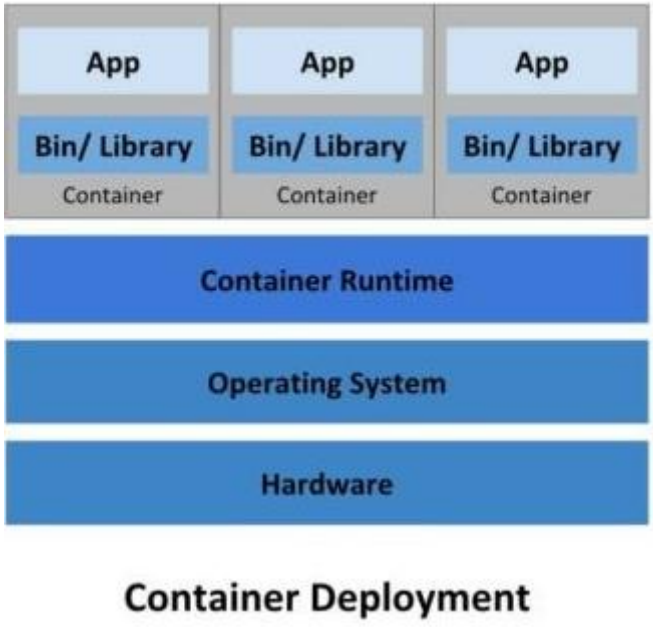


Fig 2.4.2 Containers [29]

To avoid all the drawbacks of the VMs, containers leverage one OS, increasing deployment speed and portability with lower costs and memory footprint. Containers sit on top of a physical server and its host machine's OS. Each container shares the hosts OS kernel, and can also share its binaries and libraries.

Those aspects make containers extremely lighter than VMs. So, this is the main reason that they are the virtualization method chosen for environments and architectures such as Fog and Edge Computing.

Furthermore, containers bring micro-services logic to the Edge due to their modularity. Rather than run an entire complex application inside a single container, the application can be split in to modules (such as the database, the application front end, and so on).

Container Runtimes and ways to orchestrate these microservices will be explained in chapter 3.

2.3 Overview

In this thesis, we present a resource aware container orchestrator based on real-time metrics monitoring. Using a unique two layer Polynomial Regression model we achieve an augmentation in application throughput, whilst using less resources. Furthermore, in order to evaluate our scheduler, we used Aarch64 Edge devices, revealing the default Kubernetes scheduler's weaknesses in resource constrained environments. Finally, using state-of-the-art frameworks such as KubeEdge, Kubernetes and Prometheus shows that our work is quite promising and to be continued.

The rest of this thesis is organized as follows. In Chapter 3, we present related work regarding resource management on Edge computing and container orchestration, while on the same time we highlight the scientific gaps throughout the literature. In Chapter 4, we present Docker as a container runtime, Kubernetes as a container orchestrator and KubeEdge as an Edge framework. In Chapter 5, we present our two layer regression approach to the stated problem. In Chapter 6, we present our experimental infrastructure, in order to evaluate our proposal. Also, Prometheus as a metrics exporter is introduced as well as MLPerf Benchmark Inference Suite. In chapter 7, the experimental evaluation of our approach is presented. Finally, in chapter 8 we conclude and propose future work in order to improve our work.

Chapter 3

Related Work

3.1 Job Scheduling at the Edge

In the past years, several studies have been conducted on job and task scheduling in Fog and Edge Computing environments. As it has already been stated, resource management is critical in deploying applications at the Edge. Kuljeet Kaur et al [2] address the problem of carbon emission footprints, interference and energy consumption in this kind of environments. They propose a solution through Integer Linear Programming, based on a multi objective optimization problem. They are using Kubernetes, but in a different way, adding an extra layer between the cluster and the Edge.

Authors of [3], aim to tackle the problem of limited bandwidth resources in Edge computing environments. Also in [4], authors target the network latency optimization as an optimization objective. They succeeded to reduce network latency up to 80% compared to the Default Kubernetes Scheduler, showing that awareness of the environment makes a big difference.

Showing respect to energy constraints in IoT environments, in [5] authors propose an algorithm for energy consumption and precision optimization. In the same approach, based on the algorithm, in [6] we can see that data placing algorithms can be used, always respecting the constraints in Edge computing devices. Furthermore, a different approach to resource allocation through economic model analysis is proposed in [7]. Although this work is about resource allocation, it cannot be integrated into a Kubernetes environment and plenty of limitations are occurred. Finally, in [8] they propose a Deep Reinforcement Learning (DLR), as a solution to dynamic user requests on top of resource constrained devices. Their DRL model learns to select the optimal allocation policy, saving energy whilst reducing response time and therefore upgrading user experience.

In overview of the literature, resource allocation is a quite popular problem and lots of researchers are trying to bring solutions to the table. However, predicting the throughput of an application before it is offloaded, through a two-layer Polynomial Regression classification, whilst paying respect to energy and resource constraints have never been proposed. Adding to this, we are using KubeEdge, a very promising Edge Framework based on Kubernetes, which can be used in lots of IIoT and IoT scenarios.

Chapter 4

KubeEdge: A Kubernetes based Edge Computing platform

In this chapter we will explain everything that is needed, in order to understand how KubeEdge works.

4.1 Docker: A container runtime

Docker is the industry's De Facto container runtime in most operating systems. In simple terms, Docker is a software platform that simplifies the process of building, running and managing distributing applications. From another perspective, Docker is a set of platform as a service (Paas) products that use OS-level virtualization in containers [9].

Now we are going to break Docker down in its major components as shown in Fig 4.1:

- **Docker client:** It is the primary way that Docker users interact with the whole platform. The Docker client communicates with docker daemon with API calls, through the docker command. This client can communicate with more than one daemon.

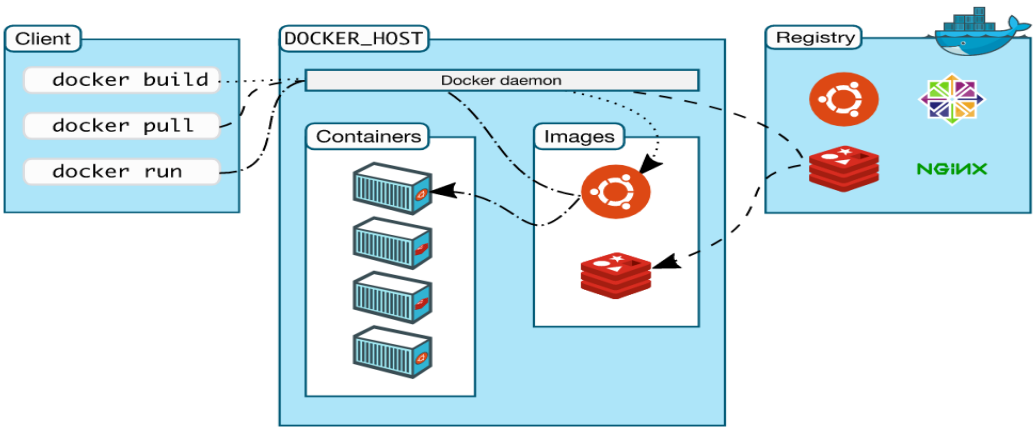


Fig 4.1 Docker architecture [8]

- **Docker Daemon (dockerd) :** The docker daemon listens for Docker API requests and manages Docker objects such as images , containers, networks and volumes. A daemon can also communicate with other daemons to manage Services [10].

- **Docker Registry:** Docker registry is a stateless, high scalable server side application that stores Docker images. All users can store their images in order to manage them from multiple hosts. Docker Hub is the most common interface for users, due to its privacy and support from the Docker community.

- **Images:** Containers are the running instance of an image. So, images are a read-only template with instructions to create a Docker container. A unique docker image can be created either based on an existing one, or enriching it with everything deemed necessary.

- **Dockerfiles:** Dockerfiles are simple text documents that contain all the commands a user could call on the command line to assemble an image [11]. They give birth to a docker image. So, through Dockerfiles, any environment can be created or reproduced.

4.2 Kubernetes: A container orchestrator

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, which facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services support and tools are widely available [12]. So, in simpler terms, Kubernetes can be used in order to manage huge workloads in enormous clusters with ease.

Let's make it a little bit more comprehensible with an example. Consider there are 1000 nodes-devices located in a server and your task is to place a very intense workload in order for an application to be deployed. Where every microservice should be executed? How will the app's network be configured? How microservices are going to communicate with each other and share data? Kubernetes answers all of these questions and a user can notice through a very well built API.

Now, we are going to break down Kubernetes on its main components. As shown in Fig 4.2, Kubernetes is divided in Control Plane Components and Node Components. Only Control Plane Components which are the main parts of the Kubernetes orchestrator will be described.

- **Kube Api Server**: Kubernetes API is actually the front end of the control plane and, by extension, of the whole environment. It is designed to scale horizontally and be friendly to the user.

- **ETCD**: It is the main storage and key-value store of the whole cluster. Most notably, it manages the configuration data, state data, and metadata for Kubernetes.

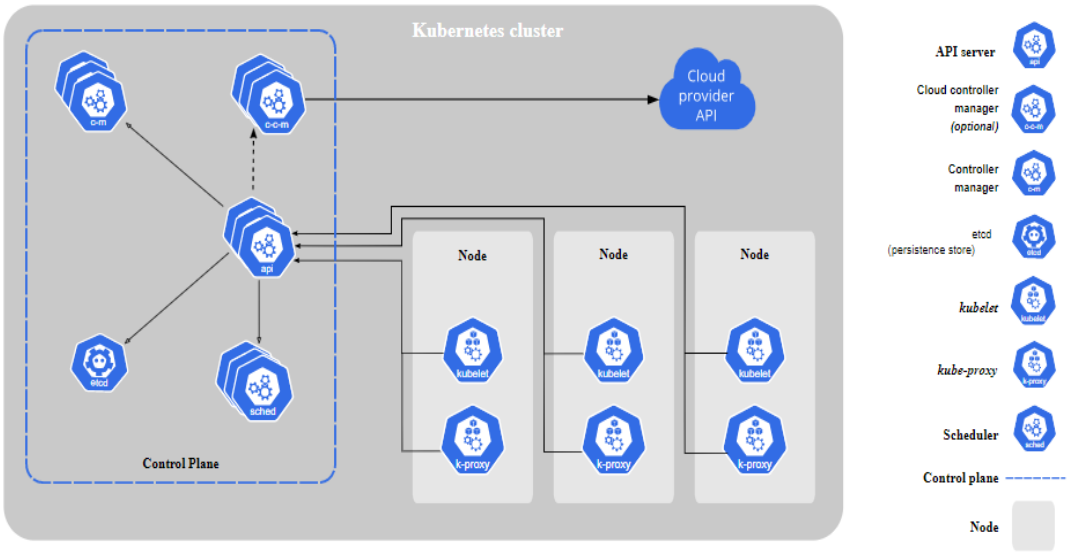


Fig 4.2: Kubernetes Components [13]

- **Kube Controller-Manager**: This is a single binary that runs multiple controller processes. You can break it down into Node Controller, Job Controller, Endpoint Controller and Service Controller. Each one is responsible for Nodes, Pods-Jobs, Services-Pod's Network and API access to different namespaces in that order.

Last, but not least, component of the Control Plane will be discussed in Paragraph 4.2.1, analyzing its whole process.

4.2.1 Kube Scheduler

Kube Scheduler is the control plane component-process which assigns Pods to Nodes. In order to explain the scheduler framework, we first have to define Pods. Pods are the smallest and most common deployable units that anyone can create in a Kubernetes Cluster. In a Pod there can be one or multiple containers with shared storage and network resources. Pods also have a unique IP address so the cluster can identify them as different objects in the cluster. As an analogy, Pods in a Docker environment would be the containers.

As we mentioned above, the main reason of the scheduler's existence is to assign Pods to Nodes. The scheduler achieves that through a scheduling cycle, in which Pods are exposed to. It is designed to handle lots and lots of nodes as well as Pods. So as we can see in Figure 4.2.1, Scheduler Framework has 2 stages: Scheduling Cycle and Binding Cycle.

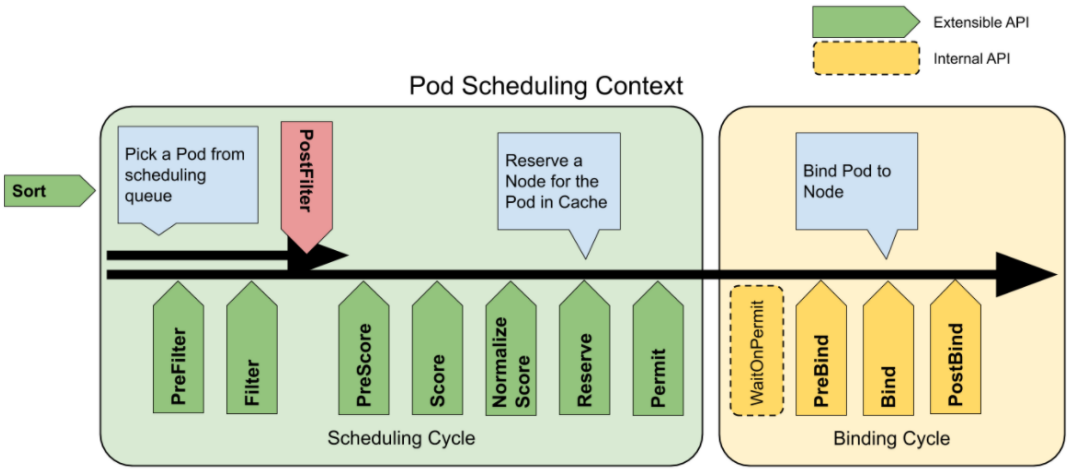


Fig 4.2.1: Scheduling Framework Extension Points [14]

Scheduling Cycle will be broken down to its main components, further discussing the plugins that matter the most:

- **QueueSort:** The scheduler implements a queue in order to sort Pods that are going to be inserted in the Scheduling Cycle.

- **Prefilter – Filter - Postfilter:** In this stage of the scheduling cycle, pre-process information about the Pod and its conditions are gathered. Next, the scheduler filters out all nodes in the cluster and according to the Pods conditions, deeming a node as infeasible or not. If any error occurs, the scheduling cycle is aborted.

- **PreScore – Score – NormalizeScore:** The most important part of the scheduling cycle. The scheduler, according to its own algorithm gives scores to Nodes. In reality, it ranks nodes according to their affinity, labels and resources. As resources, the Kubernetes cluster use miliCpus, for CPU cores on each node, and bytes of requested memory or memory available. After the NormalizeScore plugin, the scheduler knows to which Node the API server is going to bind the Pod.

As we mentioned above, Kubernetes is designed for clusters located on servers, and in servers, there are almost unlimited resources. As a result, Kube Scheduler is also manufactured to rank and assign Pods to this kind of nodes. Furthermore, Kube Node Components are quite resource intensive and are also meant to exist in servers. Our case, though, includes resource constrained devices that exist only at the Edge. So, KubeEdge will be presented as a lightweight version of Kubernetes.

4.3 KubeEdge

KubeEdge [15] is an open source system, extending native containerized application orchestration and device management to hosts at the Edge. It is deployed upon an already existing Kubernetes cluster and provides support for application, networking and metadata synchronization between cloud and edge [16]. Another important feature is MQTT communication between Edge devices, enabling the developer to import custom logic in any app.

Extending Kubernetes power at the Edge, KubeEdge takes care of node components resource demands. In simpler terms, KubeEdge is constructed out of two basic components, EdgeCore and CloudCore. These are two binaries, one running at the Edge Nodes and the other constantly running on the Kubernetes Master Node.

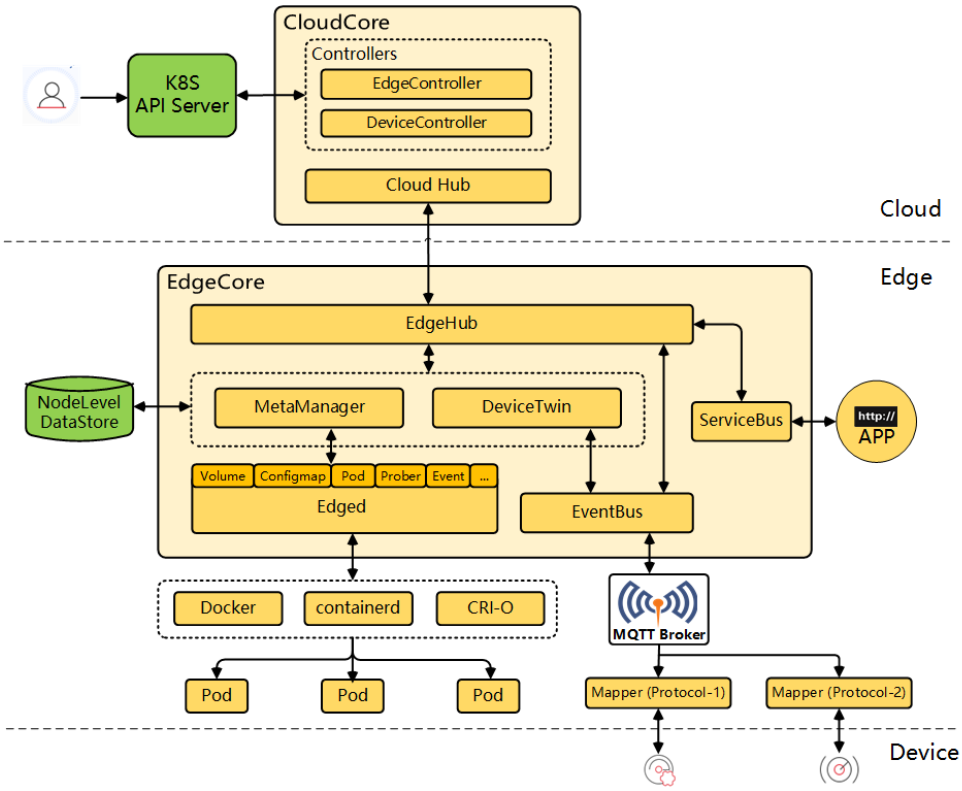


Fig 4.3: KubeEdge Architecture [7]

As depicted in Figure 4.3, Cloud Hub and Edge Hub are the main components of CloudCore and EdgeCore respectively, which are responsible for communication between cloud and edge hosts. CloudCore also contains some controllers in order to talk to Kubernetes API. That aspect gives the user the possibility of taking action on KubeEdge related tasks through K8s (Kubernetes) API. Therefore, the interface of KubeEdge is very similar to Kubernetes therefore it making it easier to utilized by users that are already familiar with the latter.

Now, at the EdgeCore part, lots of components are replacing the Kubelet. Moreover, Edged is for managing containerized applications. Event Bus handles MQTT client and server interaction. The rest are for storage, message processing and http requests. The main differences from Kubelet are the small binary size of EdgeCore, its low demands on resources and MQTT protocol communication between the nodes.

To conclude, KubeEdge is quite new to the world and it can provide what K8S provide to the cloud, to the Edge. It is easy to get and deploy existing complicated machine learning algorithms, exactly where the data is produced. Also, the heterogeneity, the scalability and the cross platform design will help considerably in the future. Though KubeEdge is way lighter than Kubernetes, it still uses Kube Scheduler and all of Kubernetes core components.

Chapter 5

Resource Aware Orchestration

In this chapter, we present the proposed algorithm and the mechanisms used in our scheduler for Pod orchestration.

5.1 Requirements and Usage

The scheduler is already built in a docker image, compiled for x86_64 architecture. The user must label all of the Edge nodes with its Docker mount point. That is why, in Edge devices the storage is almost every time constrained and Docker's overlay storage might be mounted in an SD card and not be at "/". Furthermore, the user must label each pod-job-deployment with a "MEM_REQ" label in GBs, so that the scheduler may know the app's requirements. The scheduler will decide, according to the absolute distance of the memory required, to which of the three apps, the current app looks more alike. IP addresses and rest information about the nodes are configured through API calls.

5.2 Polynomial Regression

Polynomial Regression is a form of regression analysis in which the relationship between the independent and dependent variables are modeled in the nth degree polynomial. Polynomial Regression models usually fit with the method of least squares. The least square method minimizes the variance of the coefficients, under the Gauss Markov Theorem. Polynomial Regression is a special case of Linear Regression where we fit the polynomial equation on the data with a curvilinear relationship between the dependent and independent variables [17].

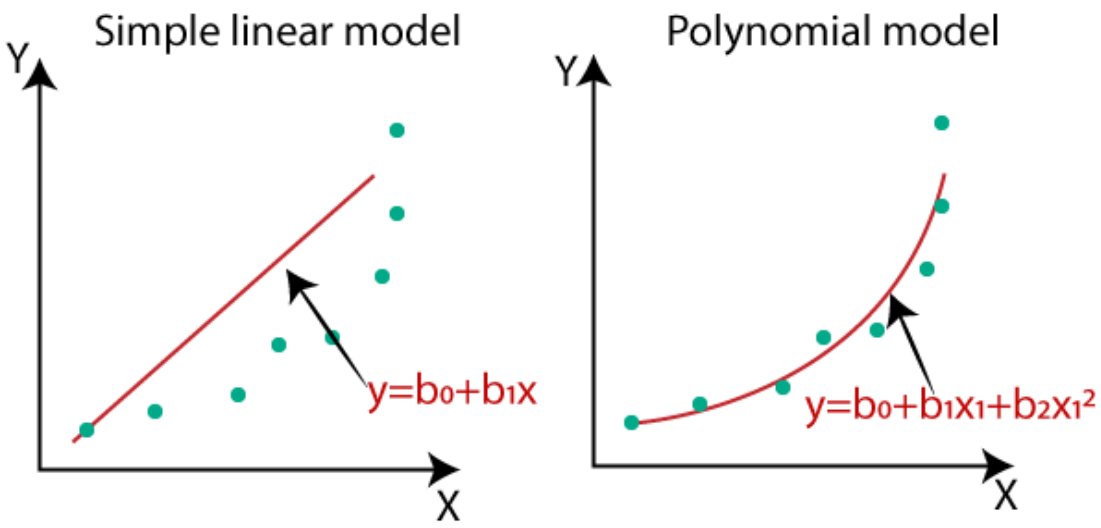


Fig 5.2: Linear vs. Polynomial Regression [27]

In our case, we used two layers of Regression. At first, Single variable Polynomial Regression is utilized to predict the values of the metrics during runtime. The independent variable, in each regression, is each of the metrics (CPU, mem, temperature) 30 seconds before scheduling, the dependent variable being the metric during runtime. There are eighteen regressions in total (# of Apps * # of Metrics * # of Devices) for the first layer. Now for the second layer, Multi variable Polynomial Regression is used to predict the Queries per Second of the app during runtime. The dependent variable this time is QPS and the independent variables are the already predicted values of the metrics during runtime.

In order that the statistical analysis (model) works properly, we operate a device profiling and the results are fed as input data to our model. Thus, we ran dockerized workloads on each device and created the dataset that the model was going to train on. The workloads were different in terms of intensity and the final datasets contain almost two hundred rows. Furthermore, data analysis and normalization were required in order for the data to be suitable for our case. At first, data came with a second decimal point precision. This aspect was not suitable for our situation, as we mean to predict metrics, with the highest accuracy possible, increasing throughput in the highest precision. Therefore, we normalized all of our data, targeting at no decimal point precision for better accuracy and more generalized results. Finally, in order to execute this kind of profiling we created a testing environment written in Python and Bash communicating straight with Docker's API.

5.3 Scheduling Cycle

When a pod requires to be scheduled, our scheduler gets notified through a Kubernetes Informer [18]. After the pod's detection, it is inserted into a reverse priority Queue, with priority being a semaphore protected variable. The protected variable represents the number of pods that are in the Queue at the current time. This way we implement a FIFO queue for our scheduler. We ought to protect this variable because the Kubernetes Informer is actually in a goroutine and at some point our main program and the goroutine might try to write on the same variable, which will lead to a logical error.

Next, the pod with the highest priority enters the scheduling cycle. First, candidate node's health in the cluster, in terms of availability, is checked through Kubernetes API. Then, each healthy node resources must be eligible for some certain boundaries (CPU <= a%, MEM <= b% and DISK_AVAILABLE >= c GB). These boundaries can rotate depending on the devices resources. In our case, after extensive experimentation, we decided that those thresholds should be: $a = b = 80$ & $c = 2$. If at least one node meets the requirements mentioned above we move one to the next extension point, else the pod goes back to the priority queue. If another pod is in the queue, it will have higher priority than the examined pod. That is because, besides the fact that the examined pod exists further in the cluster, it exited the queue and thus in a FIFO queue, such as ours, when it re-enters it will have a lower priority than the already existing ones.

The main part of the algorithm takes place at the Scoring extension of the scheduler. Knowing the pods memory usage, the scheduler, according to the current state of the metrics at the devices, through Polynomial Regression predicts the state of the metrics, in each node, during runtime. Another Multivariable Polynomial Regression model takes place and predicts the QPS in each node. Max QPS predicted indicates the best node and thus, the scheduler decides that this is the node the pod is going to be sent off to. In short, this two layer Regression model, takes as input the current state of the metrics in all of the filtered devices, and predicts the chosen throughput metric (QPS) during runtime.

Finally, the pod gets bound to the node chosen and an event is emitted, so that the user can trace the pod's lifecycle. Our scheduling cycle is as shown in Figure 5.3:

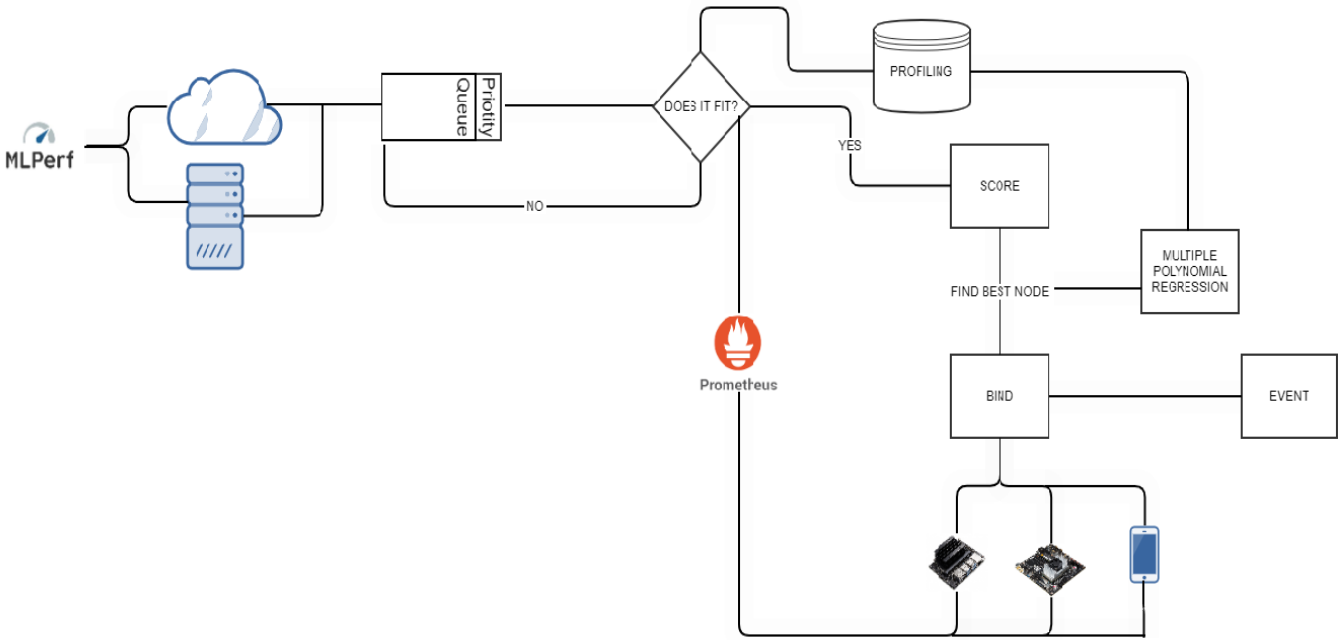


Fig 5.3: My Scheduling Cycle

Chapter 6

Experimental Infrastructure

In this chapter, we are going to describe the Kubernetes cluster created using KubeEdge for our experiments. Also, we present the monitoring system and the benchmarks we used during this whole process.

6.1 System Setup

Our cluster consists of three nodes, two Edge nodes and one Cloud node as shown in Fig 6.1.

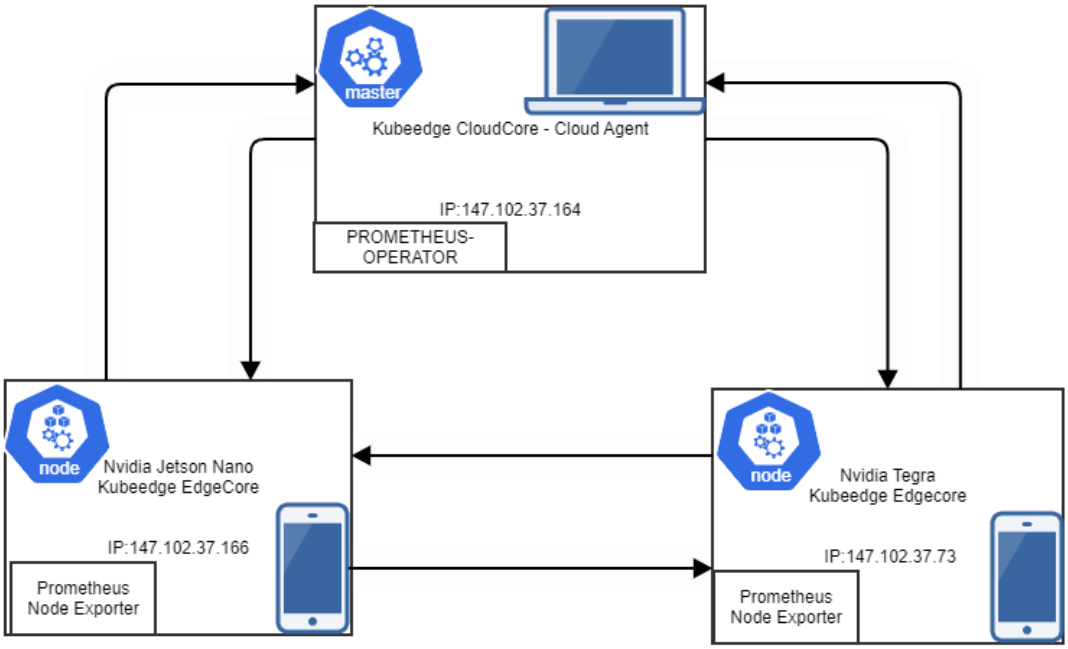


Fig 6.1: System Setup

For the Cloud-Master node in our cluster, we used a virtual machine sitting on top of a machine with 4 Intel Cpu cores and 8 GBs of RAM. The virtual machines OS is Ubuntu 20.04.2 and the architecture is x86_64. Now, for our Edge-Worker nodes we used an NVIDIA Jetson Nano and an NVIDIA Tegra X1 and they are both Aarch64.

Arm architecture is one of the most widely used architectures in Edge computing systems.

As shown in Fig 6.1, we created a single node Kubernetes cluster and then deployed KubeEdge on top of it. Calico was chosen as the Kubernetes clusters CNI (Container Network Interface). Also, Prometheus is used to scrape metrics from the devices and inform about the metrics whereabouts in the cluster. For node to node communication, we used Mosquitto as it is advised by KubeEdge. Most of the components, as well as the devices, will be described below.

So, we created a heterogeneous, secure and scalable cluster which can be found in real life situations. The main idea is that sensors and cameras will be connected at the devices and, ML and AI algorithms can be deployed in order to process data right where they are produced. This could be a solid scenario in an Industry related situation.

6.2 Edge Devices

ARM is an acronym of Advanced RISC Machines and it came to dominate the market. Its reduced instruction set makes it more powerful and efficient for mobile and Edge devices. Furthermore, it offers extremely low power consumption which is the main reason it is so popular in Internet of Things devices.

6.2.1 Tegra X1

Tegra is a SoC developed by NVIDIA and integrates ARM architecture (Aarch64), a graphics processing unit and a common DRAM between the two. It is extremely low power and designed specifically to reduce, as much as possible, the energy and power consumption of the board. NVIDIA talked up the Jetson TX1 a lot for deep learning purposes and being able to fit the JTX1 module on drones and other portable, low-power devices [15]. So, it's composed of:

	JETSON TX1
GPU	1 TFLOP/s 256-core Maxwell
CPU	64-bit ARM A57 CPUs
Memory	4 GB LPDDR4 | 25.6 GB/s
Storage	16 GB eMMC
Wifi/BT	802.11 2x2 ac/BT Ready
Networking	1 Gigabit Ethernet
Size	50mm x 87mm
Interface	400 pin board-to-board connector

Fig 6.2.1: TEGRA X1 [19]

6.2.2 Jetson Nano

At the heart of the Nano module we find Nvidia's "Erista" chip which also powered the Tegra X1. It is a full blown single-board-computer in the form of a module [20]. Also, it is specifically designed for developers to use at home and for every kind of usage in general. The goal of the form-factor is to have the most compact form-factor possible, as it is envisioned to be used in a wide variety of applications where a possible customer will design their own connector boards best fit for their design needs. The specifics are as shown in Fig 6.2.2:

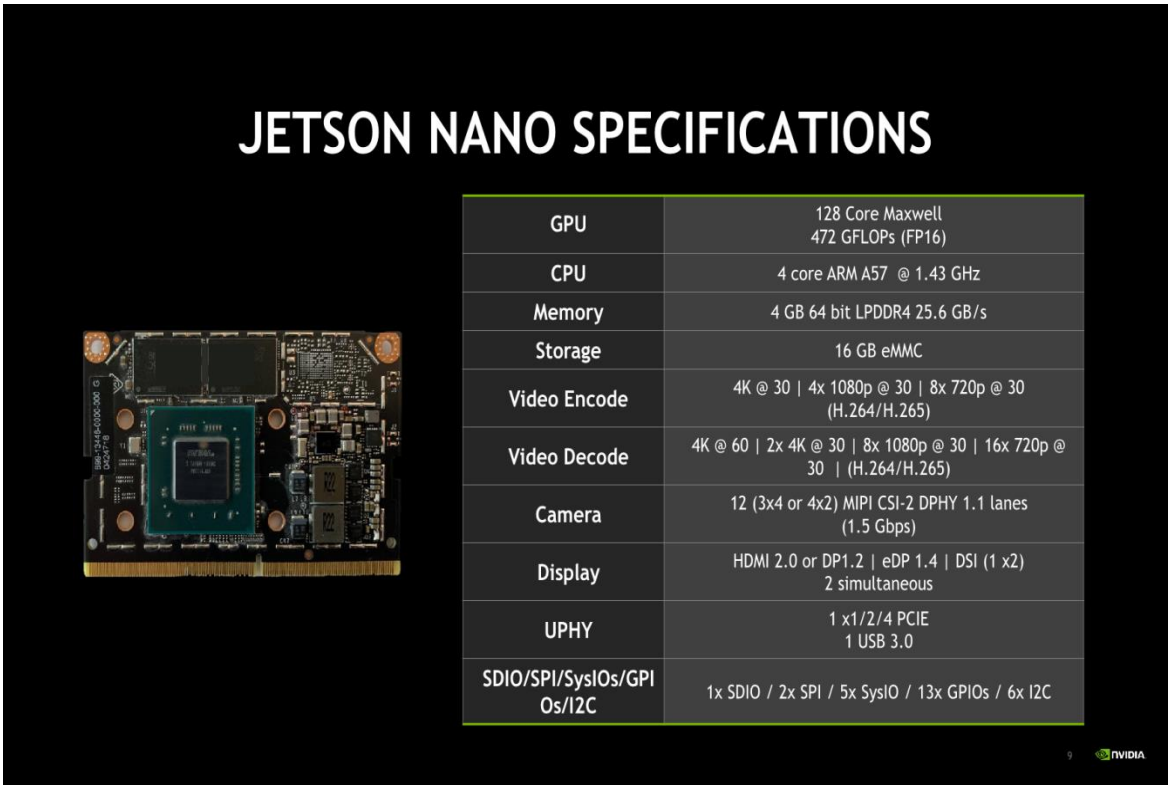


Fig 6.2.2: JETSON NANO [16]

6.3 Monitoring System

As mentioned above, we used Prometheus [21] as the monitoring system to our cluster. Prometheus is an open-source monitoring and alerting toolkits originally built at Soundcloud. Since its inception in 2012, many companies and organizations have adopted Prometheus, and the project has a very active developer and user community. It is now a standalone open source project and maintained independently of any company. To emphasize this, and to clarify the project's governance structure, Prometheus joined the Cloud Native Computing Foundation in 2016 as the second hosted project, after Kubernetes [22].

Prometheus is based on a multi-dimensional data model with time series data identified by metric name and key/value pairs. It provides PromQL, a flexible query language to leverage the dimensionality. Prometheus does not rely on distributed storage hence each single server node is autonomous. The time series collection happens via a pull model over HTTP while the time series pushing is supported via an intermediate gateway. The Prometheus targets are discovered via service discovery or static configuration. Finally, it provides multiple modes of graphing and dashboarding.

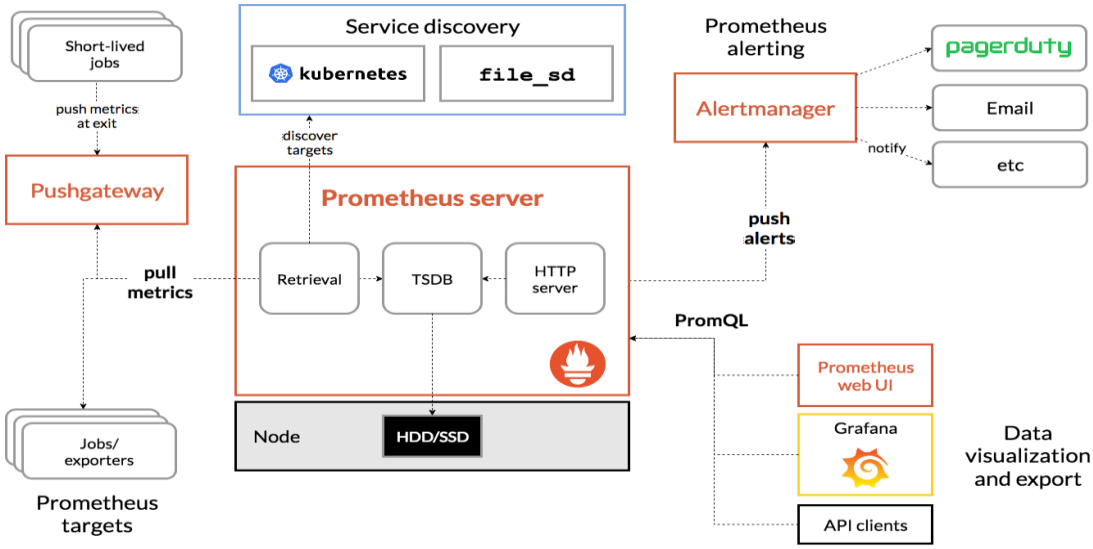


Fig 6.3: Prometheus Architecture [17]

The Prometheus ecosystem consists of multiple components, many of which are optional, as shown in Figure 6.3. The main Prometheus component is the Prometheus server which scrapes and stores time series data. There is a push gateway for supporting short-lived jobs and special-purpose exporters for services like HAProxy, StatsD, Graphite, e.t.c. For the alerts handling an alertmanager component is provided. In addition, Prometheus has client libraries for instrumenting application code while various tools are supported.

In our case, we are using the Prometheus operator, talking straight to the Prometheus server, located on the master node of our cluster. Node exporters are used as Prometheus agents in the Edge nodes. PromQL is used taking advantage of all its aspects (functions and literals) [23]. The scraping period chosen is 5 sec which is the default value, after testing other scraping periods.

6.4 Benchmarks

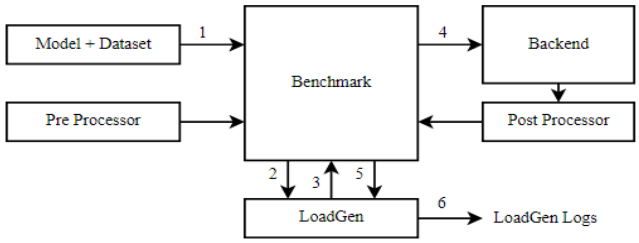
In order to test the scheduler and the environment we needed workloads that consist of latency-critical applications. So we wanted a benchmark suite for measuring how fast systems can process inputs and produce results, using a trained model. That is because in the majority of such systems, Edge users want their data to be produced, processed and analyzed as fast as possible, thus leading to real-time requirements in terms of execution. Moreover ML domain is one of the most widely increasing application domains in Edge computing systems.

MLPerf Inference [24]benchmark suite is the perfect one for the job described above. We used three of the vision-classification and detection benchmarks as described in Table 6.4:

Area	Task	Model	Dataset	Quality	Backend	Accuracy
Vision-Heavy	Image Classification	Resnet50-v1.5	ImageNet (224x224)	fp32	Tensorflow	76.456%
Vision-Light	Image Classification	Mobilenet-v1 224	ImageNet (224x224)	fp32	Tensorflow	71.676%
Vision-Light	Object Detection	Ssd-Mobilenet-v1	COCO(300x300)	fp32	Tensorflow	mAP 0.23

Table 6.4: MLPerf Inference Benchmarks used

The key component of the MLPerf Inference Benchmark is the Load Generator [25]. The Load Generator is a reusable module that efficiently and fairly measures the performance of inference systems. It generates traffic for scenarios as formulated by a diverse set of experts in the MLPerf working group. The scenarios emulate the workloads seen in mobile devices, autonomous vehicles, robotics, and cloud-based setups. Although the Load Generator is not model or dataset aware, its strength is in its reusability. Fig 6.4 shows is a diagram of how the Load Generator can be integrated into an inference system, resembling how the used MLPerf reference models are implemented.



1. Benchmark knows the model, dataset, and preprocessing.
2. Benchmark hands dataset sample IDs to LoadGen.
3. LoadGen starts generating queries of sample IDs.
4. Benchmark creates requests to backend.
5. Result is post processed and forwarded to LoadGen.
6. LoadGen outputs logs for analysis.

Fig 6.4: Integration Example & Flow [20]

Finally, the scenario we used was SingleStream and it is all meant to run only on CPUs. Also, for the apps to require less space at the SDs, we used only 1000 images-queries from the datasets during the experiments. The LoadGen output logs inform the user about the time took to load the dataset, as well as the total QPS (Queries per Second) of the model, the total time, and the mean value.

Chapter 7

Evaluation and Experiments

In this chapter, we use our experimental infrastructure to evaluate our custom scheduler. We used different kind of experiments, based on intensity and diversity. Each of them gives us insight about different metrics concerning the app's throughput and the board's utilization.

7.1 Experiments

In order to evaluate our custom scheduler we execute numerous different experiments. Each of them gives us insights about the advantages and disadvantages compared to the default Kubernetes scheduler.

An experiment is actually a module written in Python programming language, running on the master node of our cluster. The communication with the Kubernetes cluster is through the python Kubernetes client. Our experiment tracks each application from the beginning to the end and keeps logs, for every action occurred. The experiment takes two inputs. The user must specify the range that defines the arrival between two consecutive apps and the folder that contain the yaml files representing our workload.

The workload is thrown at the cluster as Jobs. We decide to use Jobs as they are easier to manage, compared to Deployments and stand alone Pods. Then a Pod is created and each Pod creates a different inference engine by using the MLPerf Inference container. A container has already the backend, the scenario, the dataset and the container's memory usage label defined.

As we mentioned, lots of experiments were executed, changing the workload, as well as the intensity (time range).First, we created a random workload sixty apps strong. This way, we could each time change the size of the workload to ten, thirty and sixty apps. These are the first three different experiments, executed with the same time range, varying in terms of diversity and size. We decided to use a time range between two and five minutes, as the apps range in completion time from three to twelve minutes in our devices, if they stand alone. Furthermore, we reduced the time range in a fourth experiment from five minutes max to three, keeping the lower limit the same at two minutes.

The whole workload consists of nineteen apps using the Mobilenet model, twenty bases on Resnet and twenty one based on SSD-Mobilenet model. For more

information about MLPerf inference apps you can retrace back at Table 6.4. The metrics chosen to evaluate and compare the two schedulers are shown below:

- **QoS metrics**

    1. QPS of the whole workload.
    2. QPS of each app.
    3. # of lost apps, due to node availability.

- **Board Resource Utilization Metrics**

    1. System Average CPU utilization during runtime.
    2. System Average CPU Temperature.
    3. System Average Memory utilization during runtime.

7.2 Results & Scheduler Comparison

In this section, we present the scheduler comparison between our custom scheduler and the Kube-Scheduler (Default). In the figures below, the distribution of the QPS made by each application is shown respectively. We observe that our custom scheduler offers better throughput for the applications than the default scheduler. Queries per Second are a metric that shows the total duration of the application as well as its latency and efficiency. Fig 7.2.1 shows the QPS distribution made for each Resnet model based application, in each of the three experiments made. In Fig 7.2.2 and Fig 7.2.3 the rest of the inference engines, based on Mobilenet and SSD-Mobilenet, are shown respectively. It is necessary to mention that on average, our scheduler achieves an approximately 25% increase in total QPS throughput.

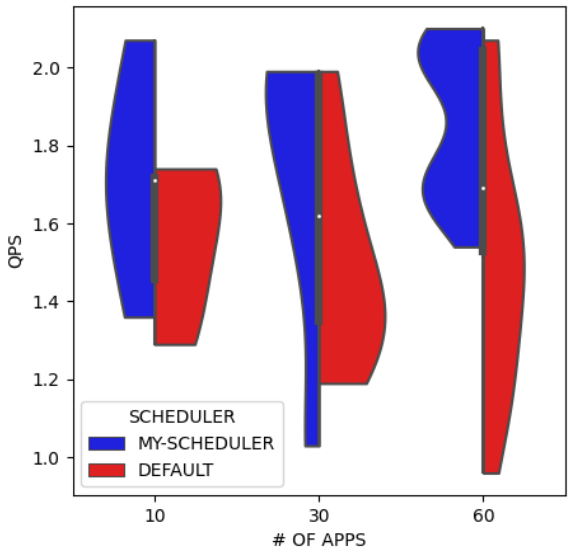


Fig 7.2.1: Resnet QPS distribution

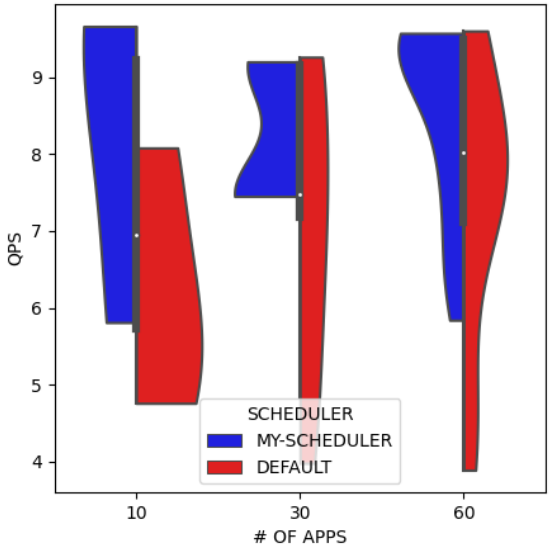


Fig 7.2.2: Mobilenet QPS distribution

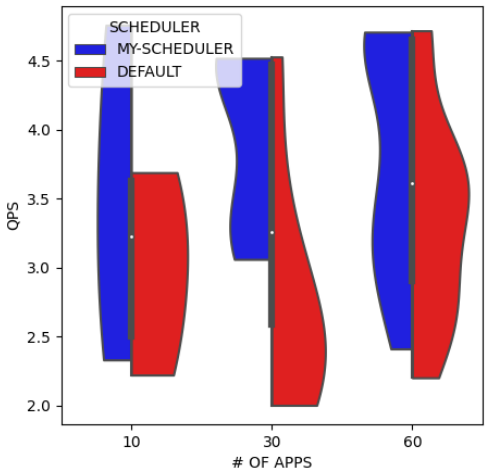


Fig 7.2.3: SSD-Mobilenet QPS distribution

As shown in the next figures, resource utilization is also significantly lower, despite the fact of the workload being quite intense. Therefore, our scheduler can achieve better efficiency, always bearing in mind to the restricted resources of the devices. Furthermore, the default scheduler in all of the experiments made at least once, one of the devices crash, meaning it bound an application to a device that wasn't able to handle the work. So, in Fig 7.2.4 average CPU utilization for each application is shown, and as we can see we decreased utilization for up to 10%. In Fig 7.2.5 the average Memory utilization is shown, achieving the same amount of decrease in average utilization, approximately 10%. Finally, in Fig 7.2.6 the average Temperature of the boards CPUs is shown, achieving a decrease of 1-2 $\mathcal{C}$ in total. In total, the power consumption of the boards decreased.

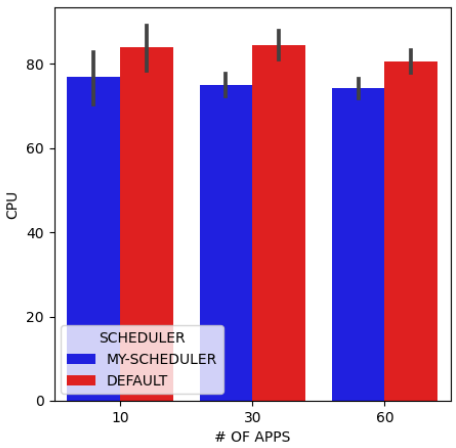


Fig 7.2.4: System average CPU utilization

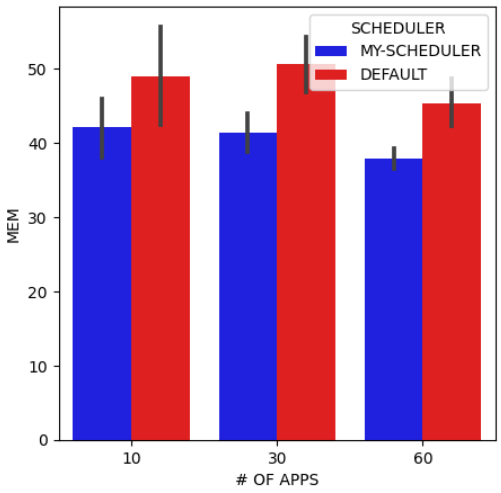


Fig 7.2.5: System average Memory utilization

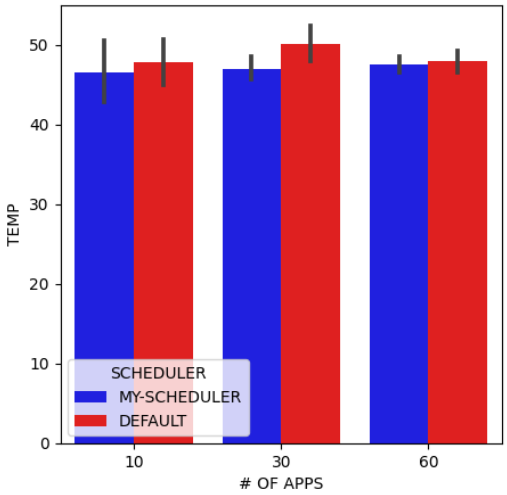


Fig 7.2.6: System average CPU Temperature

In order to examine the behavior of our proposed scheduler in more intensive workloads, we provide as input a more time restricted scenario, by limiting the time range to three minutes. As expected, the default scheduler not knowing about the device's resource restriction is making the devices unavailable to the cluster. While in many cases the devices crash. Our scheduler, on the other hand, handles all of the workload efficiently, not losing a single app. In total the default scheduler loses

16.6% of the workload, except in case of ten apps. In Fig 7.2.7, we can see the QPS throughput for each of the ten apps. We observe that in every scenario our scheduler succeeds a better throughput. The overall improvement is around 26%. In Fig 7.2.8, we can see a little less than 10% reduction in CPU utilization. In Fig 7.2.9, the reduction in memory utilization is again around 10%. Moreover, in Fig 7.2.10 we can see a difference around 1 ℃ in temperature.

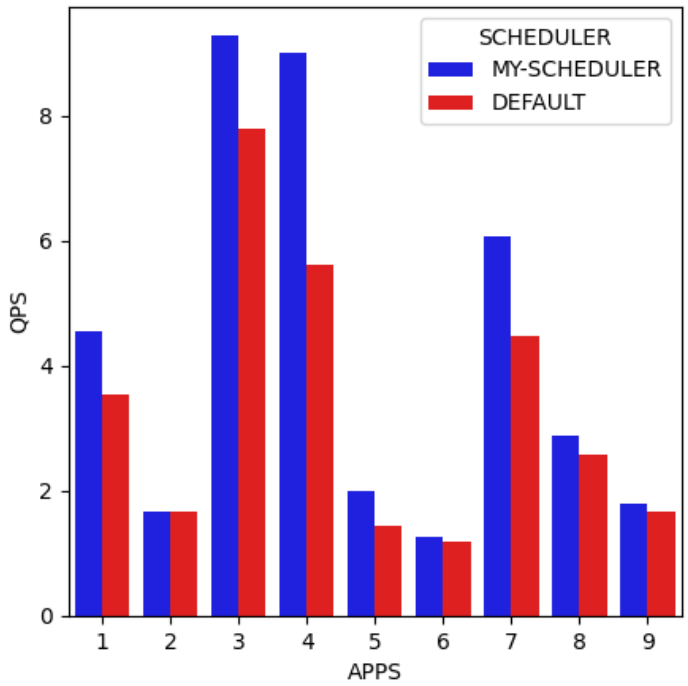


Fig 7.2.7: QPS of each of the ten application in the experiment.

Fig

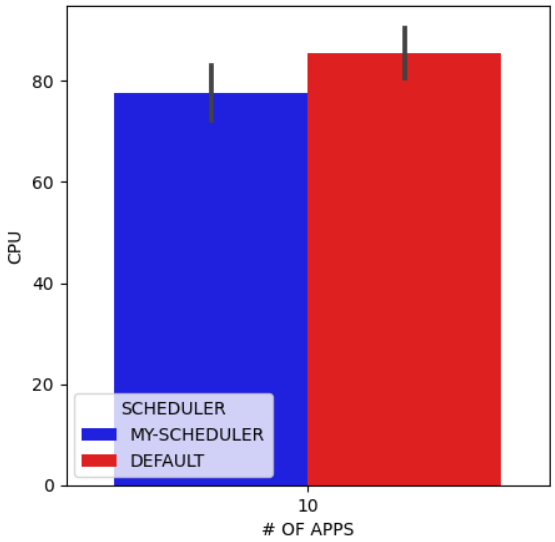


Fig 7.2.8 System average CPU utilization

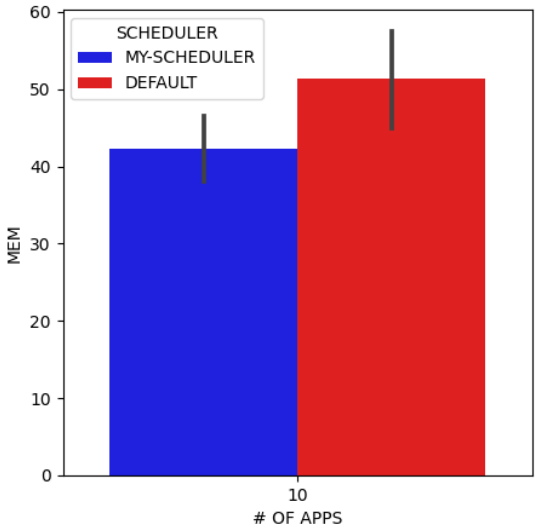


Fig 7.2.9: System average Memory utilization

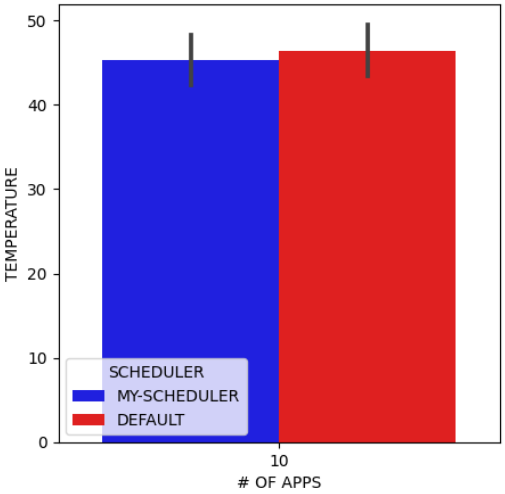


Fig 7.2.10: System average CPU Temperature

Chapter 8

Conclusion and Future Work

8.1 Summary

In this thesis, we design a resource aware container orchestrator that runs on every Kubernetes environment. We used state of the art tools such as KubeEdge and Prometheus. It is very important to mention that our orchestrator is scalable and optimized for fog computing environments. Furthermore, we evaluated the scheduler with MLPerf inference engines, against the Kubernetes default scheduler. As shown, in every scenario our custom scheduler improves the quality of service and achieves this with significantly less resources. Also, our approach is scalable, heterogeneous and able to achieve this result in most resource constrained environments.

8.2 Future Work

This particular subject is very promising due to its versatility and its necessity. The fact that, with an optimal orchestration in this kind of environments, industry and end-users needs and quality of service can be improved dramatically makes it very interesting and exciting.

First of all, for this research, our proposed framework can be evaluated in more heterogeneous devices and applications, so that we can explore the scalability of our approach. Another improvement might be letting the scheduler know about the app's requirements, before the second and most important polynomial regression. Furthermore, Kubernetes Scheduler plugins can be created using our method, integrating with the Kube Scheduler. That way, the advantages of the default scheduler can be combined with the advantages of our scheduler.

In a similar perspective to our approach, we could investigate several ML models as alternative scheduling algorithms, such as deep learning models training during runtime. This way, the model should be more effective and train specifically for each different case. Another idea is to integrate into the scoring algorithm, connection related aspects. Thus, making the scheduler more appropriate for clusters consisted of devices in different locations. Last but not least, in modular

scenarios on Edge computing systems, the behavior of our approach should be evaluated, under geolocated, networking and QPS variations.

Βιβλιογραφία

[1] T. Alam, «A Reliable Communication Framework and Its Use in Internet of Things (IoT),» *JOUR,* 30 May 2018.

[2] S. G. G. K. S. H. A. M. A. Kuljeet Kaur, «KEIDS: Kubernetes-Based Energy and Interference Driven Scheduler for Industrial IoT in Edge-Cloud Ecosystem,» *IEEE Internet of Things Journal,* pp. 4228-4237, May 2020.

[3] V. T. L. B. S. X. D. S. J. H. F. Samie, «Computation offloading and resource allocation for low-power IoT edge devices,» *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT),* pp. 7-12, 14 December 2016.

[4] T. W. B. V. F. D. T. J. Santos, «Towards Network-Aware Resource Provisioning in Kubernetes for Fog Computing Applications,» *2019 IEEE Conference on Network Softwarization (NetSoft),* pp. 351-359, 24 June 2019.

[5] J. W. X. Ding, «Study on Energy Consumption Optimization Scheduling for Internet of Things,» *IEEE Access,* pp. 70574-70583, 29 May 2019.

[6] Z. L. H. W. Z. L. F. J. d. C. L. Z. Sun, «Sensing Cloud Computing in Internet of Things: A Novel Data Scheduling Optimization Algorithm,» *IEEE Access,* pp. 42141-42153, 20 March 2020.

[7] M. Katsaragakis, «DMRM: Distributed Market-Based Resource Management of Edge Computing Systems,» *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE),* pp. 1391-1396, 16 May 2019.

[8] H. S. AbishiChowdhury. Shital A.Raut, «DA-DRLS: Drift adaptive deep reinforcement learning based scheduling for IoT resource management,» *Journal of Network and Computer Applications,* pp. 51-65, 15 July 2019.

[9] «Docker Docs,» [Ηλεκτρονικό]. Available: https://docs.docker.com/get-started/overview/.

[10] «Wikipedia,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Docker_(software).

[11] «Docker Docs,» [Ηλεκτρονικό]. Available: https://docs.docker.com/engine/reference/builder/.

[12] «Kubernetes,» [Ηλεκτρονικό]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

[13] «Kubernetes,» [Ηλεκτρονικό]. Available: https://kubernetes.io/docs/concepts/overview/components/ .

[14] «Kubernetes,» [Ηλεκτρονικό]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/.

[15] «KubeEdge,» [Ηλεκτρονικό]. Available: https://kubeedge.io/en/.

[16] «KubeEdge,» [Ηλεκτρονικό]. Available: https://kubeedge.io/en/docs/kubeedge/ .

[17] [Ηλεκτρονικό]. Available: https://medium.com/analytics-vidhya/understanding-polynomial-regression-5ac25b970e18.

[18] [Ηλεκτρονικό]. Available: https://pkg.go.dev/k8s.io/client-go/informers.

[19] [Ηλεκτρονικό]. Available: https://www.phoronix.com/scan.php?page=article&item=nvidia-jtx1-perf&num=1.

[20] [Ηλεκτρονικό]. Available: https://www.anandtech.com/show/14101/nvidia-announces-jetson-nano.

[21] «Prometheus,» [Ηλεκτρονικό]. Available: https://prometheus.io/.

[22] «PROMETHEUS,» [Ηλεκτρονικό]. Available: https://prometheus.io/docs/introduction/overview/.

[23] [Ηλεκτρονικό]. Available: https://prometheus.io/docs/prometheus/latest/querying/basics/.

[24] «MLPerf Inference,» [Ηλεκτρονικό]. Available: https://github.com/mlcommons/inference.

[25] «LoadGen,» [Ηλεκτρονικό]. Available: https://github.com/mlcommons/inference/tree/master/loadgen.

[26] «Appypie,» [Ηλεκτρονικό]. Available: https://www.appypie.com/everything-you-should-know-about-internet-of-things.

[27] «Cloudflare,» [Ηλεκτρονικό]. Available: https://www.cloudflare.com/learning/serverless/glossary/what-is-edge-computing/.

[28] «Wikipedia,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Edge_computing.

[29] «Researchgate,» [Ηλεκτρονικό]. Available: https://www.researchgate.net/figure/Three-layer-fog-computing-architecture-

5_fig1_325144725.

[30] «VirtualMachines,» [Ηλεκτρονικό]. Available: https://marionoioso.com/2019/07/16/virtual-machines-vs-containers-vs-serverless-computing/[.

[31] S. A. H. S. Abishi Chowdhury.

[32] «IoT Growth,» [Ηλεκτρονικό]. Available: https://i.pinimg.com/originals/8e/19/ca/8e19ca2dd683a50cf4d75c23caa5946a.jpg.