ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ

A software analysis tool for Energy and Time-aware function placement on the Edge

Διπλωματική Εργασία

Γιάννος Γαβριηλίδης

Επιβλέπων Καθηγητής

Δημήτριος Σούντρης
Καθηγητής

Αθήνα, Φεβρουάριος 2022

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ

A software analysis tool for Energy and Time-aware function placement on the Edge

Διπλωματική Εργασία

Γιάννος Γαβριηλίδης

Επιβλέπων Καθηγητής

Δημήτριος Σούντρης
Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 3η Φεβρουαρίου 2022.

.....	.....	.....
Δημήτριος Σούντρης Καθηγητής Ε.Μ.Π.	Παναγιώτης Τσανάκας Καθηγητής Ε.Μ.Π.	Διονύσιος Πνευματικάτος Καθηγητής Ε.Μ.Π.

……

Γιάννος Γαβριηλίδης

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

Περίληψη

Ο όρος και συγκεκριμένα η τεχνολογία του cloud computing μοντέλου (υπολογιστικό νέφος) έχει μπει για τα καλά στη ζωή μας τα τελευταία χρόνια και βλέπουμε τόσο τη ζήτηση όσο και την προσφορά cloud υπηρεσιών να αυξάνεται με ταχύτατους ρυθμούς. Η ανάγκη όμως για αποθήκευση και επεξεργασία μεγάλων όγκων δεδομένων σε συνδυασμό με την ένταξη όλο και περισσότερων IoT συσκευών στο διαδίκτυο, αυξάνει δραματικά την συνολική κατανάλωση ενέργειας στα datacenters και παράλληλα φέρνει σε κίνδυνο υποβάθμισης της συνολικής Ποιότητας Υπηρεσίας (QoS) και της Ποιότητας Εμπειρίας (QoE) του χρήστη.

Το εναλλακτικό υπολογιστικό μοντέλο edge computing προτάθηκε και χρησιμοποιήθηκε επιτυχώς στα IoT περιβάλλοντα, με σκοπό να φέρει τους υπολογισμούς πιο κοντά στον τελικό χρήστη μειώνοντας τον όγκο των δεδομένων που αποστέλλονται στο νέφος και κατά συνέπεια τις καθυστερήσεις επικοινωνίας. Χρειάζεται λοιπόν να υπάρχει μια ευελιξία στον τρόπο που αναπτύσσεται ένα λογισμικό με σκοπό να τρέξει σε περιβάλλον νέφους γι’ αυτό και τα «containers» είναι πολύ διαδεδομένα στις μέρες μας. Τα «containers» εισάγουν στο μοντέλο αυτό τα «microservices» και την έννοια του serverless computing σύμφωνα με την οποία κάθε επιμέρους λειτουργικότητα μπορεί να υλοποιηθεί αυτόνομα και ανεξάρτητα χωρίς να απαιτείται προκατανομή υπολογιστικών πόρων.

Σε αυτή τη διπλωματική εργασία, παρουσιάζουμε ένα εργαλείο το οποίο προτείνει ένα τρόπο με τον οποίο θα μπορούσαν να τρέξουν οι επιμέρους συναρτήσεις ενός μονολιθικού κώδικα ο οποίος πρέπει να τρέξει σε serverless περιβάλλον, έτσι ώστε με δεδομένο κάποιο ανώτατο όριο χρόνου εκτέλεσης να επιτευχθεί η ελάχιστη δυνατή κατανάλωση ενέργειας στα μηχανήματα που αποτελούν το περιβάλλον cloud-edge το οποίο χρησιμοποιείται. Το περιβάλλον αυτό ενορχηστρώνεται από τον κυβερνήτη και για την διαχείριση των επιμέρους συναρτήσεων χρησιμοποιείται η πλατφόρμα ανοιχτού κώδικα OpenFaaS. Το εργαλείο μας χρησιμοποιεί τεχνικές μηχανικής μάθησης για τις προβλέψεις ενέργειας και χρόνου και αναλύει το αρχείο κώδικα που του δίνεται ως προς τη μνήμη και το χρόνο που απαιτεί κάθε επιμέρους συνάρτηση του. Η αξιοποίηση της πληροφορίας αυτής και η απόφαση για την τελική πρόταση γίνεται με τη βοήθεια ενός αλγορίθμου ελαχιστοποίησης με προσεγγιστική λύση.

Λέξεις Κλειδιά: <<Διαχείριση πόρων, Κυβερνήτες, OpenFaaS, Ενορχήστρωση πακέτων, Διαδίκτυο των Πραγμάτων, Edge Computing, Serverless Computing>>

Abstract

The technology of the cloud computing model has entered our lives for good in recent years and we can easily see both the demand and the supply of public and private cloud services growing rapidly. However, the need to store and process large volumes of data in combination with the integration of more and more IoT devices on the Internet increases dramatically the total power consumption in datacenters and at the same time puts at risk the overall Quality of Service (QoS) and Quality of Experience (QoE).

Edge computing model was proposed and used successfully in IoT environments, in order to bring the computations and the process of the data closer to the end user by reducing the volume of data sent to the cloud and reduce communication delays. So there needs to be a flexibility in the way cloud-native software is developed and for this "containers" are very common nowadays. "Containers" introduced "microservices" in this model and also the concept of serverless computing according to which each individual functionality can be implemented independently, without the need for any pre-allocation of computing resources.

In this thesis, we present a tool that proposes a way in which the individual functions of a monolithic code could run in a serverless environment, so that given a maximum runtime threshold, the minimum possible energy consumption in devices is achieved. These devices consist our serverless infrastructure (cluster), they are orchestrated by Kubernetes and the deployment of our code is managed by a scalable, fault-tolerant event-driven serverless platform called OpenFaaS. Our tool uses machine learning techniques for energy and time predictions and analyzes (profiles) the code file given in terms of memory allocation and the run-time required for each of its individual functions. The utilization of this information and the decision for the final proposal is achieved with the help of our self-developed minimization algorithm with approximate solution.

Keywords: << Resource Management, Kubernetes, OpenFaaS, Container Orchestration, Internet of Things, Edge Computing, Serverless Computing >>

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 με εμπιστεύτηκε από την πρώτη στιγμή για την εκπόνηση αυτής της διπλωματικής εργασίας μέσα από την οποία έμαθα πολλά, ανέπτυξα δεξιότητες και εμπλούτισα τις γνώσεις μου. Επίσης, θα ήθελα να ευχαριστήσω το εργαστήριο της σχολής, Microlab, για τους πόρους που διέθεσε όπως επίσης ευχαριστώ τους υποψήφιους διδάκτορες Χαράλαμπο Μαράντο και Αχιλλέα Τζενετόπουλο για τον χρόνο τους, την πολύτιμη βοήθεια, υπομονή και καθοδήγησή τους.

Ένα τεράστιο ευχαριστώ στους ανθρώπους που στάθηκαν δίπλα μου όλα αυτά τα όμορφα φοιτητικά χρόνια και ο κάθε ένας με τον δικό του τρόπου συνέβαλε στο να γίνω αυτός που είμαι σήμερα. Τέλος, το πιο μεγάλο ευχαριστώ στους ανθρώπους που βρίσκονται πάντα δίπλα μου και με στηρίζουν με την αγάπη και την εμπιστοσύνη που μου δείχνουν, στην οικογένεια μου, που χωρίς αυτούς δεν θα έφτανα μέχρι εδώ.

Αφιερωμένο στους γονείς μου, Μαρία και Ευαγόρα, και στον αδερφό μου Παναγιώτη.

Περιεχόμενα *(Table of Contents)*

1 Εκτεταμένη Ελληνική Περίληψη 15

1.1 Η συνεισφορά μας 19

1.2 Πειραματικό περιβάλλον 20

1.3 Υλοποίηση 22

1.4 Αποτελέσματα και Αξιολόγηση 24

2 Introduction 28

2.1 Internet of Things 28

2.2 Edge Computing 30

2.3 Serverless Computing 35

2.4 Virtualization and Deployment 38

2.4.1 Virtual Machines 38

2.4.2 Containers 39

2.5 Thesis Overview 41

3 Related Work 42

4 Technical background 45

4.1 Docker: A container runtime 45

4.1.1 Docker Client and Server 46

4.1.2 Docker Images 46

4.1.3 Docker Registry 47

4.1.4 Docker Containers 47

4.2 Kubernetes: A container orchestrator 47

4.2.1 Control Plane (Master) Components 49

4.2.2 Node (Worker) components 50

4.2.3 k8s vs k3s 51

4.3 OpenFaaS 53

4.3.1 OpenFaaS Design & Architecture 54

4.4 Machine Learning – Predictive Modeling....58
4.4.1 Linear Regression....60
4.4.2 Decision Tree Regression....61
5 Proposed approach....64
5.1 Profiling....64
5.1.1 1st Step – Memory profiling....65
5.1.2 2nd Step – Time profiling....66
5.2 Energy and Time predictions....67
5.2.1 Building the datasets....68
5.2.2 Prediction Model Decision....69
5.3 Placing algorithm....76
5.3.1 Knapsack and Multiple knapsack problem....76
5.3.2 Problem Definition....78
5.3.3 Algorithm....78
5.3.4 Algorithm Implementation....79
6 Experimental Evaluation....82
6.1 Experimental Setup....82
6.1.1 Edge Devices....83
6.1.2 Evaluated Application....86
6.2 Evaluation....88
6.2.1 Experimental Procedure....89
6.2.2 Results and Kubernetes Comparison....90
7 Conclusion and Future Work....93
7.1 Summary....93
7.2 Future Work....93
8 Βιβλιογραφία / References....95

1

Εκτεταμένη Ελληνική Περίληψη

Τα τελευταία χρόνια, η ραγδαία ανάπτυξη της τεχνολογίας φέρνει όλο και περισσότερες ηλεκτρονικές συσκευές στην καθημερινότητα μας. Οι περισσότερες από αυτές 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 πρότυπο 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 δυνατότητα σύνδεσης στο διαδίκτυο, κάτι που επιτρέπει σε αυτά τα αντικείμενα να συλλέγουν, να ανταλλάσσουν δεδομένα και να επικοινωνούν γενικότερα μεταξύ τους. 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 (νέφος) δεν μπορεί να διαχειριστεί αποδοτικά. Ο όγκος αυτός θα συνεχίσει να μεγαλώνει, καθώς τα δίκτυα 5G αναμένεται να αυξήσουν ακόμη περισσότερο τον αριθμό των συνδεδεμένων mobile συσκευών. Ακόμα, η ασφάλεια και η ταχύτητα επικοινωνίας με το νέφος, γίνεται όλο και πιο δύσκολη όσο πληθαίνουν οι χρήστες και η κλίμακα της κατανομής των συσκευών αυξάνεται. Προσθέτοντας σε αυτά τα προβλήματα, οι απαιτήσεις (QualityOfService), από άποψη πόρων και καθυστέρησης, των εφαρμογών έχουν αυξηθεί σε πολύ μεγάλο βαθμό με τις εφαρμογές πλέον να χρησιμοποιούν κατά κύριο λόγω ισχυρά μοντέλα Μηχανικής Μάθησης, Νευρωνικών Δικτύων και Τεχνητής Νοημοσύνης.

Λύση στα προβλήματα που παρουσιάζει πλέον το υπολογιστικό νέφος έρχεται να δώσει η τεχνολογία του Edge Computing όπου οι υπολογισμοί πλέον γίνονται στο άκρο του δικτύου, εκεί όπου αρχικά δημιουργούνται. Το Edge Computing είναι η διαδικασία της αποκέντρωσης των IT υποδομών και η τοποθέτησής τους στην πηγή των δεδομένων, δηλαδή στο «άκρο» του δικτύου. Έτσι, ελαχιστοποιούνται οι ανάγκες για επικοινωνίες μεγάλων αποστάσεων μεταξύ client (αιτητή) και server (εξυπηρετητή), γεγονός που μειώνει την καθυστέρηση και τη χρήση του bandwidth.

Το Edge Computing προσφέρει μια αποτελεσματική εναλλακτική: Τα δεδομένα γίνονται αντικείμενο επεξεργασίας και ανάλυσης πολύ πιο κοντά στο σημείο που έχουν δημιουργηθεί. Και δεδομένου πως δεν χρειάζεται να μετακινηθούν στο cloud ή σε datacenter για επεξεργασία, η καθυστέρηση είναι σημαντικά λιγότερη. Το Edge Computing (και το mobile Edge Computing στα δίκτυα 5G) επιτρέπει ταχύτερη και πιο ολοκληρωμένη ανάλυση δεδομένων, παρέχοντας ευκαιρίες για πιο αξιόπιστα insights, ταχύτερους χρόνους απόκρισης και βελτιωμένη εμπειρία πελατών.

Τα πλεονεκτήματα του Edge Computing είναι αδιαμφισβήτητα η μείωση κόστους, η μείωση πολυπλοκότητας της αρχιτεκτονικής των υποδομών και η μείωση ανάγκης διαχείρισης. Η ανάπτυξη της αγοράς του Edge Computing είναι σαφής και ανοδική. Μάλιστα, η Gartner εκτιμά ότι μέχρι το 2025 το 75% των δεδομένων θα επεξεργάζονται εκτός του παραδοσιακού Data Centre ή του cloud [1].

Παράλληλα με το Edge Computing ξεκίνησε να αναπτύσσεται ακόμη μια καινούργια τεχνολογία που ονομάζεται Fog Computing (υπολογιστική ομίχλη). Το Fog Computing αποτελεί ένα επίπεδο υπολογισμού μεταξύ του νέφους και του edge. Όταν το Edge Computing προσπαθεί να στείλει τεράστιες ροές δεδομένων απευθείας στο cloud, το Fog Computing μπορεί να λάβει τα δεδομένα από το επίπεδο Edge πριν φτάσουν στο cloud και στη συνέχεια να αποφασίσει τι είναι σχετικό και τι όχι. Τα σχετικά δεδομένα αποθηκεύονται - μεταφέρονται στο cloud, ενώ τα άσχετα δεδομένα μπορούν να διαγραφούν ή να αναλυθούν στο επίπεδο του Fog για remote access ή για ενημέρωση τοπικών μοντέλων μηχανικής μάθησης. Πιο κάτω στο σχήμα 1.1 γίνεται ξεκάθαρη η ιεραρχία που επικρατεί στα περισσότερα IoT περιβάλλοντα σήμερα.

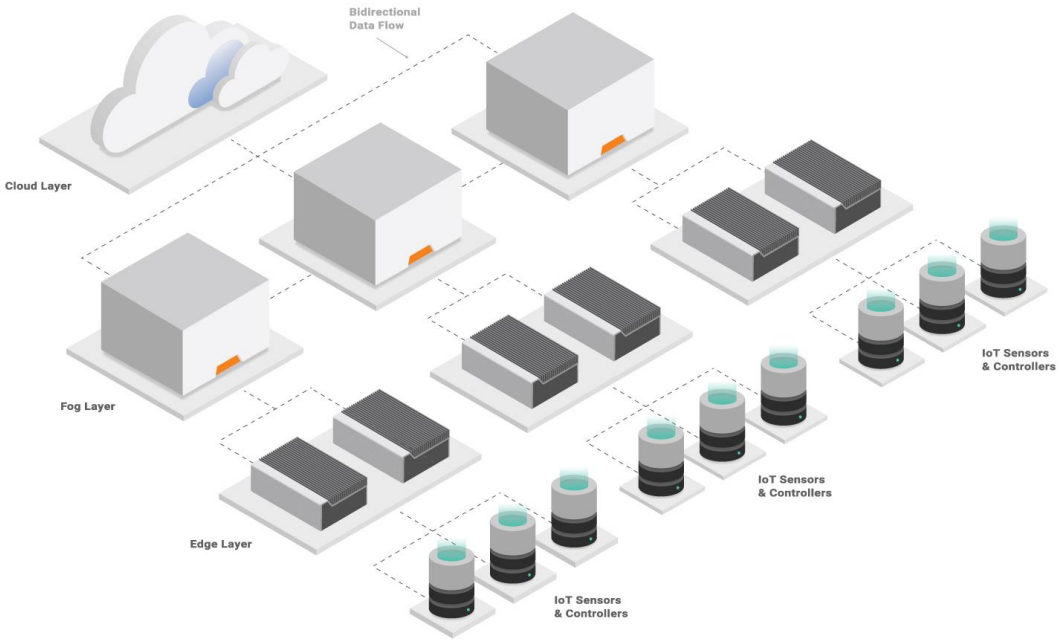


Fig 1.1: Cloud, Fog and Edge Computing Structure [2].

Παρ όλα αυτά προκύπτουν άλλα προβλήματα, καθώς οι συσκευές στην άκρη του δικτύου είναι πολύ πιο περιορισμένες, σχετικά με τους πόρους που διαθέτουν, από αυτές του νέφους. Επίσης, μιας και υπάρχει κάθε είδους συσκευή στην άκρη του δικτύου (τηλεοράσεις, κινητά, ψυγεία, αυτοκίνητα κτλ.) η εικονοποίηση (virtualization) των εφαρμογών σε αυτό το περιβάλλον προσφέρει την καταλληλότερη λύση. 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 που λέγεται Containerization.

Τα Containers λοιπόν είναι ένα τρόπος εικονοποίησης μιας εφαρμογής. Το κύριο χαρακτηριστικό, που τα κάνει να επιλέγονται από τα Virtual Machines, είναι κυρίως η «ελαφρότητα» και η ευελιξία τους. Δεσμεύουν πολύ λιγότερο χώρο και είναι πολύ πιο γρήγορα και αποδοτικά. Επιπρόσθετα, λειτουργούν πάνω στον πυρήνα του συστήματος του host machine (οικοδεσπότη) και χρησιμοποιούν απευθείας πόρους και βιβλιοθήκες του συστήματος, χωρίς κάποιο επιπλέον στρώμα λογισμικού. Έτσι έρχεται και η λογική των microservices. 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, για εργαλεία δηλαδή που διαχειρίζονται αυτές τις τεχνολογίες, με κορυφαία και επικρατέστερη τους Κυβερνήτες (Kubernetes).

Ο Κυβερνήτης (k8s και k3s) 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 και εφαρμογών. Με άλλα λόγια, είναι ειδικά σχεδιασμένοι για περιβάλλοντα νέφους.

Μετά την αξιοσημείωτη συνεισφορά του virtualization και των containers στην βιομηχανία του software development δημιουργήθηκε πλέον η ανάγκη του pay-per-use μοντέλου, δηλαδή της παροχή υπηρεσιών νέφους με τέτοιο τρόπο ώστε να «επιβαρύνεται» ο χρήστης με το κόστος χρήσης μιας παροχής, υπηρεσίας, εφαρμογής μόνο για όσο χρόνο τη χρησιμοποιεί ή ακόμη να του δίνεται και η ευκαιρία να χρησιμοποιεί μόνο όσα features - χαρακτηριστικά της εφαρμογής χρειάζεται για όσο χρόνο τα χρειάζεται.

Έτσι λοιπόν αναπτύχθηκε το μοντέλο του Serverless Computing σύμφωνα με το οποίο ο εκάστοτε developer μπορεί να γράψει τον κώδικα του χωρίς να έχει να διαχειριστεί κάποιο server (όλη η διαχείριση γίνεται από τον cloud service provider) και στη συνέχεια ο κώδικας αυτός μπορεί να ενσωματωθεί σαν ένα container και να εκτελείται μόνο κάθε φορά που καλείται. Μπορεί επίσης να γίνει scale-up or down σύμφωνα με τις ανάγκες που υπάρχουν κάτι που ενισχύει την απλότητα κατά τη διάρκεια ανάπτυξης του λογισμικού. Το εκτελεστικό μοντέλο αυτό ονομάζεται Function-as-a-Service (FaaS) όπου ουσιαστικά οι προγραμματιστές εξακολουθούν να γράφουν με custom server-side λογική, αλλά η διαχείριση γίνεται με containers τα οποία διαχειρίζεται πλήρως ένας πάροχος υπηρεσιών cloud.

Η διαχείριση ενέργειας ωστόσο στα Datacenters των πάροχων υπηρεσιών cloud αποτελεί ένα σημαντικό παράγοντα επίβλεψης του QoS των υπηρεσιών τους αλλά και της μείωση τόσο του περιβαλλοντικού αποτυπώματος τους όσο και του κόστους συντήρησης του εξοπλισμού τους. Η ραγδαία αύξηση κινητικότητας και ανάγκης για υπηρεσίες cloud τοποθετεί την επίβλεψη και την σωστή διαχείριση της ενέργειας πρώτη στη λίστα των πάροχων cloud υπηρεσιών αναφορικά με τη βιωσιμότητα τους. Το κύριο αντικείμενο της μελέτης μας λοιπόν, είναι να βρεθεί ή έστω να προσεγγιστεί μια καλή τεχνική ώστε σε ένα serverless περιβάλλον που αποτελείται από διάφορων ειδών συσκευές και ενορχηστρώνεται με κυβερνήτη, η τοποθέτηση των εκάστοτε containers – serverless functions στις συσκευές αυτές να γίνεται με τον πιο ενεργητικά αποδοτικό τρόπο.

1.1 Η συνεισφορά μας

Αρχικά δημιουργήσαμε ένα εργαλείο με το οποίο κάνουμε profiling τον κώδικα που μας δίνεται. Η ανάλυση αυτή γίνεται στο design time (κατά τον σχεδιασμό της εφαρμογής) και σε μηχάνημα διαφορετικό από τις συσκευές ακμής. Πιο συγκεκριμένα το profiling γίνεται σε Intel x86 μηχάνημα σε αντίθεση με τις ετερογενές συσκευές ακμής που αποτελούν το edge περιβάλλον μας. Γραμμή ανά γραμμή ελέγχουμε το μέγεθος μνήμης που απαιτεί η κάθε εντολή αλλά και το χρόνο που χρειάζεται για να τρέξει στο κοινό μηχάνημα που χρησιμοποιούμε για την ανάλυση, καταλήγοντας έτσι στη συνολική μνήμη και χρόνο που χρειάζεται κάθε συνάρτηση του μονολιθικού κώδικα. Στη συνέχεια με μοντέλα πρόβλεψης (Linear και Decision Tree Regression) προβλέπουμε το χρόνο που θα χρειαστεί ο κώδικας να τρέξει σε κάθε συσκευή του cluster μας αλλά και την ενέργεια που θα καταναλώσει σε κάθε μια από αυτές. Τέλος, τα δεδομένα αυτά τα επεξεργαζόμαστε μέσω ενός αλγορίθμου που αναπτύξαμε με τελικό στόχο την τοποθέτηση των εκάστοτε συναρτήσεων με τον βέλτιστο ενεργειακό τρόπο διασφαλίζοντας το επιθυμητό όριο συνολικού χρόνου που ορίζει ο χρήστης.

1.2 Πειραματικό περιβάλλον

Το σύστημα που δημιουργήθηκε για την αξιολόγηση του αλγορίθμου μας αποτελείται από μια εικονική μηχανή (VM) και τρείς Edge συσκευές. Οι τρείς αυτές συσκευές χρησιμοποιούνται ως κόμβοι εργάτες (worker nodes) στον (lightweight) Κυβερνήτη - k3s, ενώ η εικονική μηχανή, όντας η πιο δυνατή, χρησιμοποιείται ως κόμβος διαχειριστής. Επίσης, είναι πολύ σημαντικό να αναφερθεί ότι οι τέσσερεις συσκευές διαφέρουν και σε αρχιτεκτονική επεξεργαστή. Οι τρείς 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 OpenFaaS, μια πλατφόρμα ανοιχτού κώδικα, δημιουργίας και διαχείρισης serverless εφαρμογών. Με τη βοήθεια του Docker, μιας επίσης πλατφόρμας ανοιχτού κώδικα ιδανική για ενσωμάτωση εφαρμογών σε containers, το OpenFaaS παρέχει τη δυνατότητα ενσωμάτωσης, εκτέλεσης και διαχείρισης οποιασδήποτε εφαρμογής σαν serverless συνάρτηση. Ουσιαστικά πρόκειται για ένα εργαλείο αποσύνθεσης εφαρμογών σε μικρότερες μονάδες εργασίας.

Το τελευταίο και αναγκαίο συστατικό των πειραμάτων ήταν η δοκιμαστική εφαρμογή και η αξιολόγηση της. Πλέον, η πληθώρα των εφαρμογών οι οποίες χρησιμοποιούνται σε κάθε στάδιο Computing, είναι Τεχνητής Νοημοσύνης ή και Μηχανικής Μάθησης. Για αυτό τον λόγο, επιλέξαμε τη βιβλιοθήκη μηχανικής μάθησης της python, scikit-learn . Η Scikit-learn (Sklearn) είναι μια πολύ χρήσιμη και ισχυρή βιβλιοθήκη μηχανικής μάθησης στην python και παρέχει μια επιλογή εργαλείων τόσο για μηχανική μάθηση όσο και για στατιστική μοντελοποίηση. Η python ως γλώσσα προγραμματισμού επιλέχθηκε καθώς αποτελεί μία από τις ισχυρότερες επιλογές σε τεχνολογίες μηχανικής μάθησης και χρησιμοποιείται αρκετά σε serverless περιβάλλοντα.

Το περιβάλλον που δημιουργήθηκε απεικονίζεται στο Σχήμα 1.2:

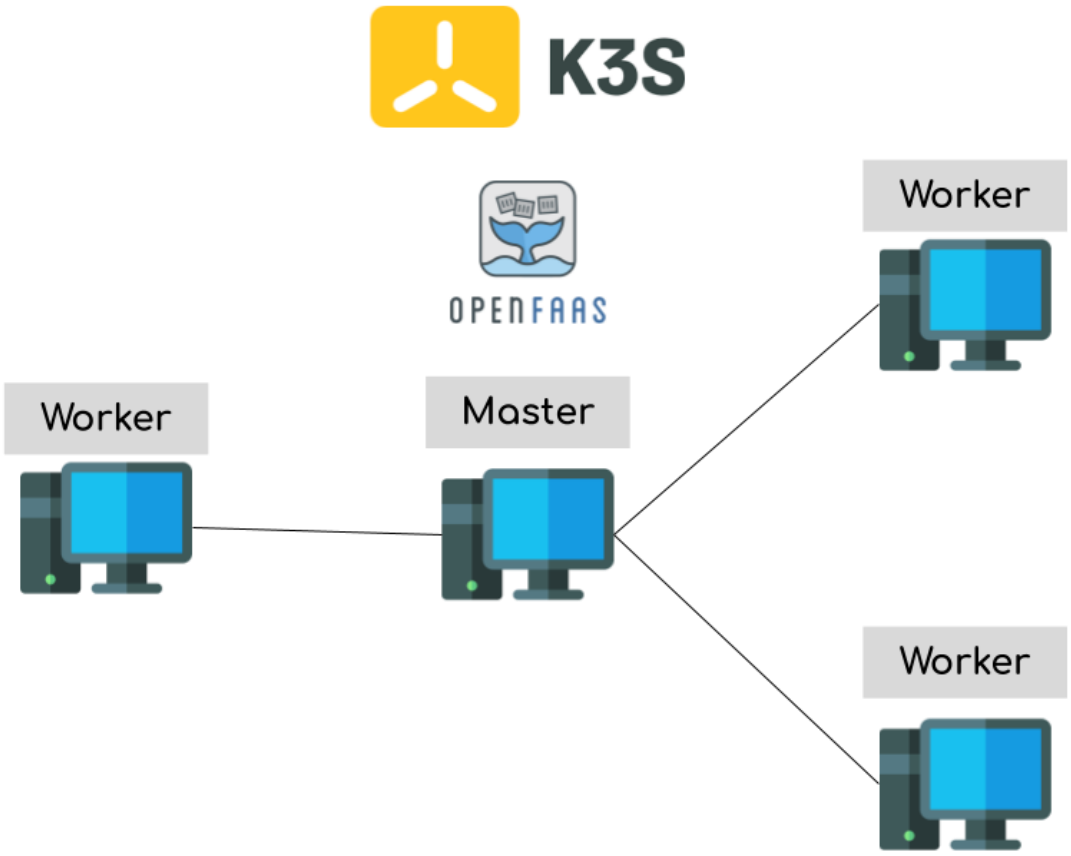


Fig 1.2: Infrastructure.

1.3 Υλοποίηση

Αφού λοιπόν ετοιμάσαμε το περιβάλλον στο οποίο θα αξιολογήσουμε τη λύση μας, περιγράφουμε τη διαδικασία που ακολουθήσαμε για να τρέξουμε την εφαρμογή όπως φαίνεται και στο Σχήμα 1.3.1.

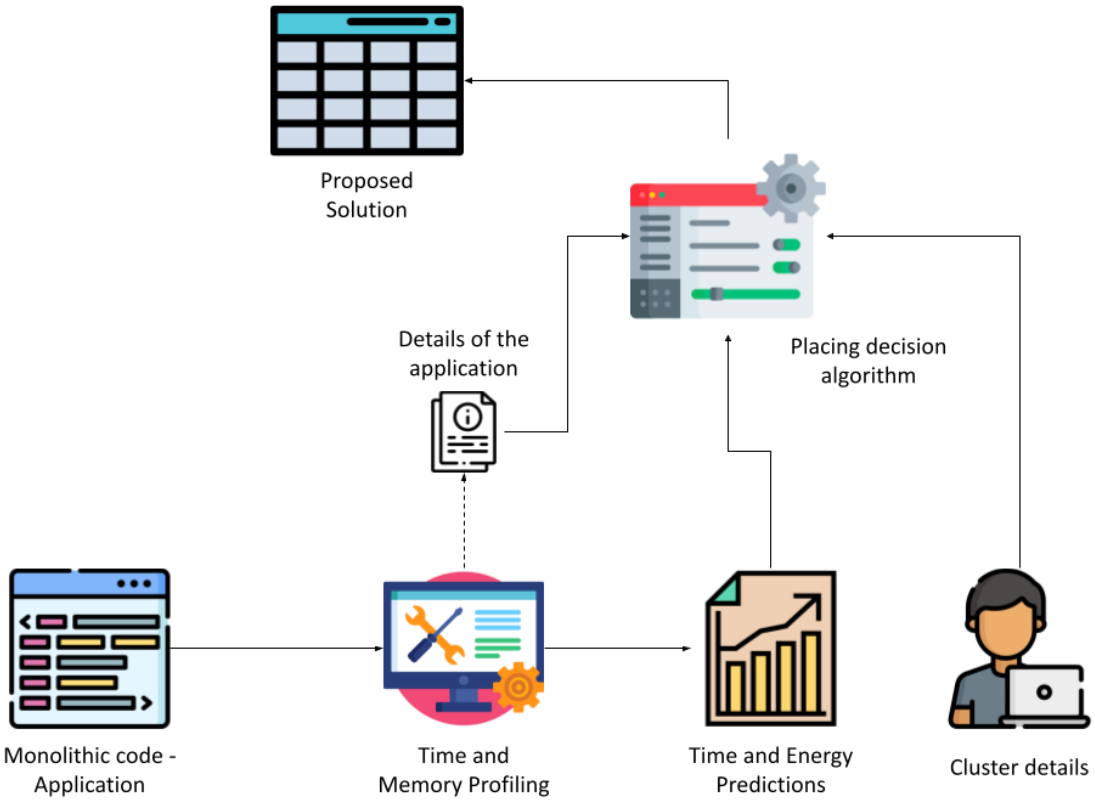


Fig 1.3.1: Evaluation procedure.

Αρχικά εισάγουμε την εφαρμογή ως ένα μονολιθικό κώδικα με διάφορες συναρτήσεις και κάνουμε το βασικό profiling χρησιμοποιώντας εργαλεία της python, τον python memory profiler για το profiling της μνήμης που χρειάζεται κάθε συνάρτηση και τον cProfiler για να δούμε το χρόνο που χρειάζεται η συνάρτηση να τρέξει στο host μηχάνημα που εκτελεί το profiling.

Στη συνέχεια χρησιμοποιούμε μοντέλα πρόβλεψης για να εκτιμήσουμε την ενέργεια που θα χρειαστεί κάθε συνάρτηση στην κάθε συσκευή του περιβάλλοντος μας. Την ίδια διαδικασία ακολουθούμε και για την εκτίμηση του χρόνου που θα χρειαστεί κάθε συνάρτηση για να τρέξει σε κάθε συσκευή.

Ο χρήστης εισάγει στον αλγόριθμο λεπτομέρειες σχετικά με το περιβάλλον (αριθμό και τύπω συσκευών που αποτελούν το serverless περιβάλλον για το οποίο θέλει να μάθει τη βέλτιστη τοποθέτηση των συναρτήσεων), το όριο χρόνου που επιθυμεί έτσι ώστε να έχει ολοκληρωθεί η εκτέλεση ολόκληρου του κώδικα και τέλος την επιλογή αν θέλει να επεξεργαστεί η διαδικασία εκτέλεσης των συναρτήσεων ως παράλληλα εκτελέσιμη ή σειριακά.

Τέλος ο αλγόριθμος αφού έχει όλα τα παραπάνω σαν είσοδο, εμπνευσμένος από τον simulated annealing algorithm (αλγόριθμος βελτιστοποιημένης αναζήτησης), δοκιμάζει αρκετούς συνδυασμούς, μέσα από πράξεις πινάκων, και καταλήγει στη λύση η οποία εξασφαλίζει την ελάχιστη δυνατή προβλεπόμενη κατανάλωση ενέργειας η οποία ικανοποιεί το χρονικό όριο που έθεσε ο χρήστης. Η προτεινόμενη λύση του αλγόριθμου παρουσιάζεται σαν ένας binary map table με στήλες που ορίζουν τα nodes και γραμμές που ορίζουν τις συναρτήσεις. Ο πίνακας, όπως φαίνεται και στο Σχήμα 1.3.2, έχει μόνο ένα «1» στη κάθε γραμμή καθώς κάθε συνάρτηση μπορεί να τοποθετηθεί μόνο μία φορά και μόνο σε ένα κόμβο. Η στήλη που έχει «1» ορίζεται ως ο κόμβος στον οποίο θα τροχοδρομηθεί η συνάρτηση που αντιστοιχεί στην συγκεκριμένη γραμμή.

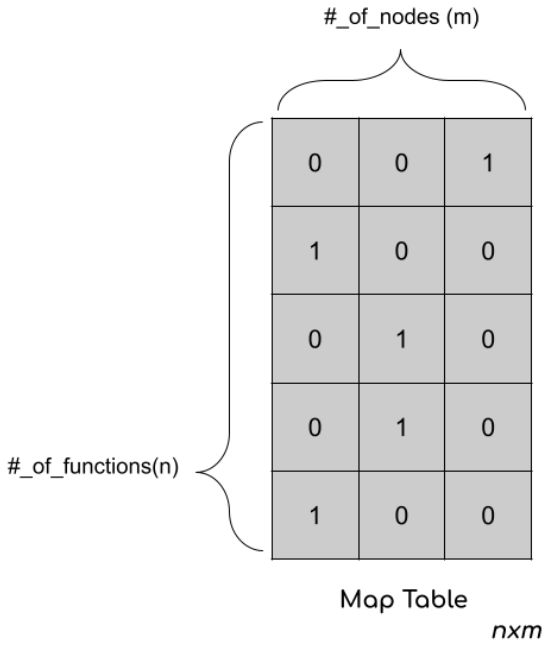


Fig 1.3.2: Proposed solution map table.

1.4 Αποτελέσματα και Αξιολόγηση

Για να αξιολογήσουμε τον αλγόριθμό μας, χρησιμοποιήσαμε μια εφαρμογή που αποτελείται από 25 συναρτήσεις μηχανικής μάθησης, τις οποίες συναντάμε στη βιβλιοθήκη της python, scikit-learn. Αυτή η μονολιθική εφαρμογή με τις επιμέρους λειτουργίες της πρόκειται να χωριστεί σε 25 ανεξάρτητα serverless functions και να εκτελεστεί στο περιβάλλον του OpenFaaS με τον πιο ενεργειακά αποδοτικό τρόπο.

Ο αλγόριθμός μας κάνει επίσης προτάσεις ανάλογα με τον αποδεκτό συνολικό χρόνο εκτέλεσης που δίνεται από την χρήστη, επομένως είναι ενδιαφέρον να δοκιμάσουμε τη συμπεριφορά των προτάσεών μας όταν ο χρήστης προσπαθεί να μειώσει τον συνολικό χρόνο εκτέλεσης και στο τέλος να προκύψει ο ελάχιστος χρόνος εκτέλεσης, δηλαδή ο ελάχιστος χρόνος που μπορεί να τρέξει η εφαρμογή στο συγκεκριμένο περιβάλλον.

Η διαδικασία λοιπόν που ακολουθήσαμε είναι απλή. Πρώτα, τρέξαμε τον αλγόριθμο και πήραμε την πρώτη πρόταση. Στη συνέχεια, συνεχίσαμε να μειώνουμε τον επιθυμητό συνολικό χρόνο εκτέλεσης μέχρι να φτάσουμε στην πρόταση με τον ελάχιστο χρόνο. Ο αλγόριθμος μας έδωσε 9 προτάσεις, η καθεμία με μικρότερο συνολικό χρόνο από την προηγούμενη.

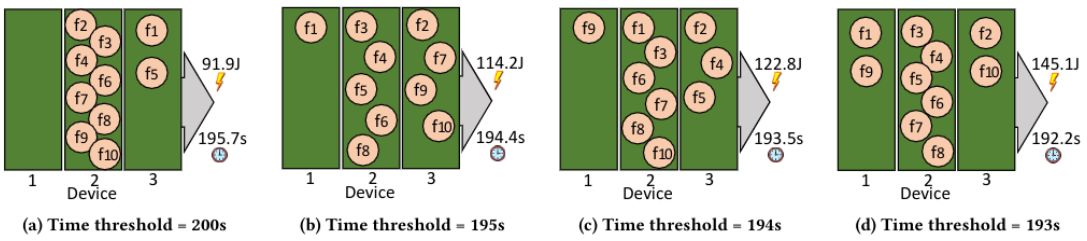


Fig 1.4.1: Παράδειγμα τοποθέτησης συναρτήσεων μειώνοντας τον επιθυμητό συνολικό χρόνο εκτέλεσης

Τα αποτελέσματα που μας δόθηκαν από τον αλγόριθμο είναι αυτά που περιμέναμε. Όπως φαίνεται στο Σχήμα 1.4.1, ο αλγόριθμός μας πρότεινε αρχικά μια λύση όπου όλη η τοποθέτηση γινόταν στις πιο ενεργειακά αποδοτικές και «γρήγορες» συσκευές (συσκευές 2 και 3 - xavier). Όταν όμως χρειάστηκε να μειώσουμε τον συνολικό χρόνο εκτέλεσης, ο αλγόριθμος, προκειμένου να επιτύχει παραλληλισμό, στη δεύτερη πρόταση χρησιμοποίησε και την «πιο αργή» και όχι τόσο ενεργειακά αποδοτική συσκευή (συσκευή 1 - nano) για την

εκτέλεση μιας από τις συναρτήσεις. Το αποτέλεσμα ήταν να μειωθεί ο συνολικός χρόνος όπως είναι λογικό αλλά παράλληλα να αυξηθεί η συνολική κατανάλωση ενέργειας. Η ίδια συμπεριφορά έχει παρατηρηθεί και για τις επόμενες προτάσεις μέχρι να φτάσουμε στον ελάχιστο χρόνο εκτέλεσης που θα μπορούσε να τρέξει η εφαρμογή μας.

Έτσι λοιπόν εκτελέσαμε τα πειράματα των οποίων τα δύο γραφήματα της καταγεγραμμένης συνολικής κατανάλωσης ενέργειας και του συνολικού χρόνου εκτέλεσης κάθε πρότασης παρουσιάζονται πιο κάτω. Παράλληλα συγκρίνουμε αυτά τα αποτελέσματα με την πρόβλεψη που δίνεται από τον αλγόριθμο και την τοποθέτηση του κυβερνήτη.

Κατά την εκτέλεση των πειραμάτων μας, παρατηρήσαμε ότι οι θεωρητικά προβλεπόμενες τιμές τόσο της ενέργειας όσο και του χρόνου είχαν παρόμοια τάση και συμπεριφορά αλλά υπήρξε κάποια διαφορά ανάμεσα στις πραγματικές τιμές που μετρήθηκαν από τα εργαλεία μας κατά την εκτέλεση των συναρτήσεων και στις προβλεπόμενες τιμές από το εργαλείο μας. Χρησιμοποιώντας λοιπόν μερικά απλά εργαλεία, μπορέσαμε να υπολογίσουμε αυτό το κόστος που θεωρείται επιβάρυνση που προκαλείται από τον κυβερνήτη και το OpenFaaS. Πιο συγκεκριμένα, παρατηρήσαμε ότι υπάρχει ένας τύπος θορύβου στις μετρήσεις που προέρχεται από την ίδια τη συσκευή και φαίνεται να είναι αυξημένος λόγω της επίδρασης του κυβερνήτη και του τρόπου με τον οποίο το OpenFaaS επικοινωνεί με τον κύριο κόμβο (master node).

Έτσι λοιπόν, για να είμαστε πιο ακριβείς στα διαγράμματα αξιολόγησης, αντί για τις προβλεπόμενες τιμές που δίνει ο αλγόριθμος χρησιμοποιήσαμε το μαθηματικό μοντέλο επιβάρυνσης κόστους που δεν είναι τίποτα άλλο από την πρόσθεση των προβλεπόμενων τιμών με το κόστος-θόρυβο που υπολογίσαμε πιο πριν.

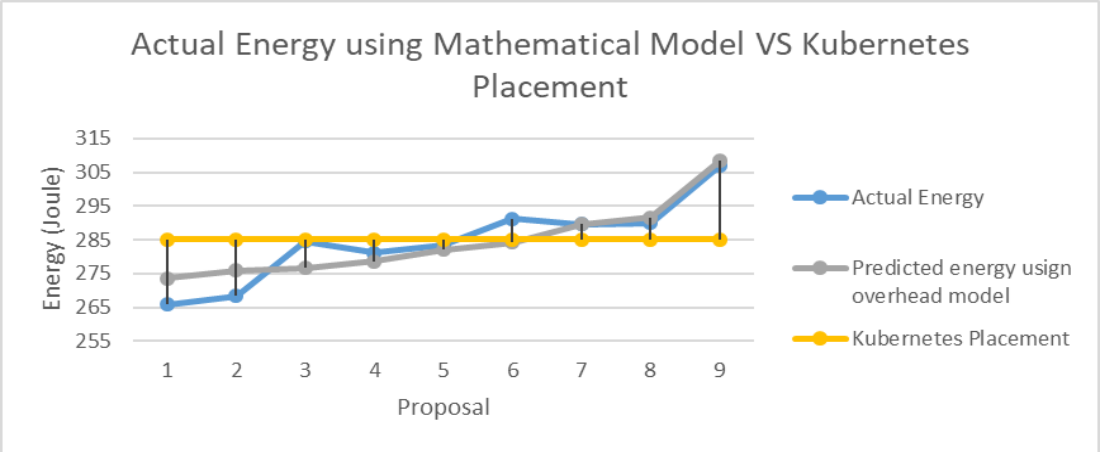


Fig 1.4.2 Energy Evaluation chart

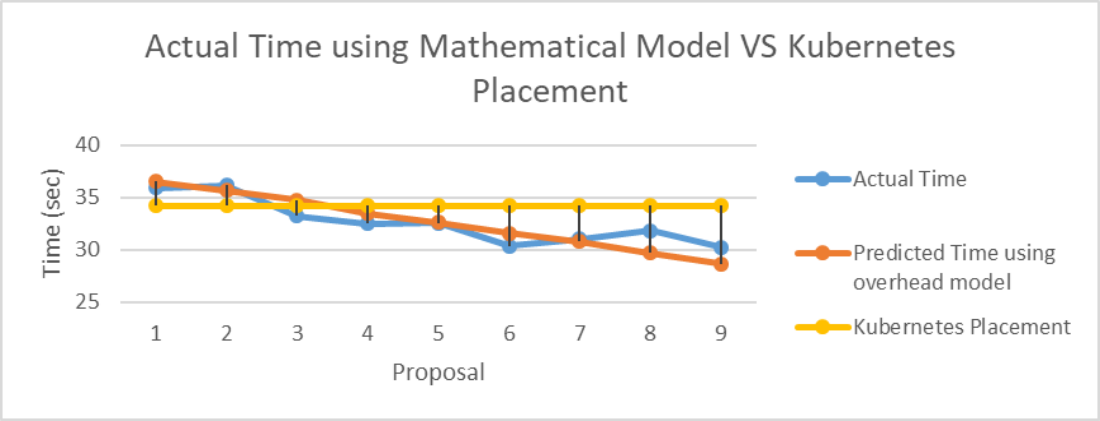


Fig 1.4.3 Time Evaluation chart

Στο Σχήμα 1.4.2 και στο Σχήμα 1.4.3 παρατηρούμε αρχικά ότι υπάρχει ένα μικρό σφάλμα μεταξύ των προβλεπόμενων και των πραγματικών τιμών, το οποίο είναι αρκετά αναμενόμενο καθώς υπάρχει πάντα χώρος βελτίωσης όταν πρόκειται για μοντέλα πρόβλεψης, αλλά μπορούμε να πούμε ότι και οι δύο γραμμές και στις δύο γραφήματα φαίνεται να ακολουθούν την ίδια τάση και συμπεριφορά κάτι που είναι πολύ σημαντικό καθώς επιβεβαιώνεται η ορθή λειτουργία του μοντέλου πρόβλεψης.

Επίσης, μπορούμε να δούμε ότι καθώς μειώνουμε συνεχώς τον επιθυμητό χρόνο λειτουργίας, η κατανάλωση ενέργειας αυξάνεται, όπως ήταν επίσης αναμενόμενο καθώς ο αλγόριθμος ξεκινάει να τοποθετεί συναρτήσεις στο λιγότερο energy efficient nano device για να πετύχει παραλληλισμό.

Όταν πρόκειται να αξιολογήσουμε τον αλγόριθμό μας σε σχέση με την τοποθέτηση του κυβερνήτη, είναι σημαντικό να επισημάνουμε ότι από τα δεδομένα των πειραμάτων προκύπτει ότι εάν ορίσουμε ένα χρονικό όριο που είναι μεγαλύτερο από τον χρόνο τοποθέτησης που πετυχαίνει ο κυβερνήτης, επιτυγχάνουμε, με τη λύση μας, λιγότερη κατανάλωση ενέργειας. Επίσης, χρησιμοποιώντας το εργαλείο μας, μπορούμε να ορίσουμε ένα χρονικό όριο χαμηλότερο από την τοποθέτηση του κυβερνήτη, το οποίο δίνει στον χρήστη μας περισσότερο έλεγχο σχετικά με το φορτίο εργασίας (workload), τη ροή εργασίας (workflow) και τη συνολική του εμπειρία (QoS) όταν χρησιμοποιεί την πλατφόρμα του OpenFaaS και την serverless αρχιτεκτονική.

Σε μια βαθύτερη στατιστική ανάλυση, πετύχαμε περίπου έως και 6% λιγότερη κατανάλωση ενέργειας συγκρίνοντας τη λύση μας πάντα με την default τοποθέτηση του κυβερνήτη (όταν

το επιθυμητό χρονικό όριο –time threshold- είναι πάνω από το συνολικό χρόνο εκτέλεσης τοποθέτησης του κυβερνήτη, όπως στο proposal 1) και μείωση του συνολικού χρόνου εκτέλεσης έως και 11% όταν ο χρήστης μας επιθυμεί να εκτελέσει την εφαρμογή με πιο γρήγορο τρόπο, πέρα από τον default τρόπο τοποθέτησης του κυβερνήτη. Συνολικά, συνδυάζοντας την ενεργειακή και χρονική απόδοση και τα αποτελέσματα του εργαλείου μας, καταφέραμε να πετύχουμε μια κατά περίπου 2,6% καλύτερη τοποθέτηση των συναρτήσεων μας στο serverless περιβάλλον, σε σχέση με τον τρόπο που θα τα τοποθετούσε ο ίδιος ο κυβερνήτης.

2

Introduction

2.1 Internet of Things

With no doubt Internet of Things (IoT) has become one of the most rapidly increasing technologies of the 21st century. Now that we want and can connect everyday objects like humidity sensors at crop fields, kitchen appliances in our smarthouses, cars, thermostats, cameras etc. to the internet via embedded devices, we can see the benefits and the huge impact of this technology in our life. 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 (devices) in the world and connecting them to the Internet.

Taking a look back and at the same time trying to predict the number of IoT devices connected to the Internet in the next years, 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.

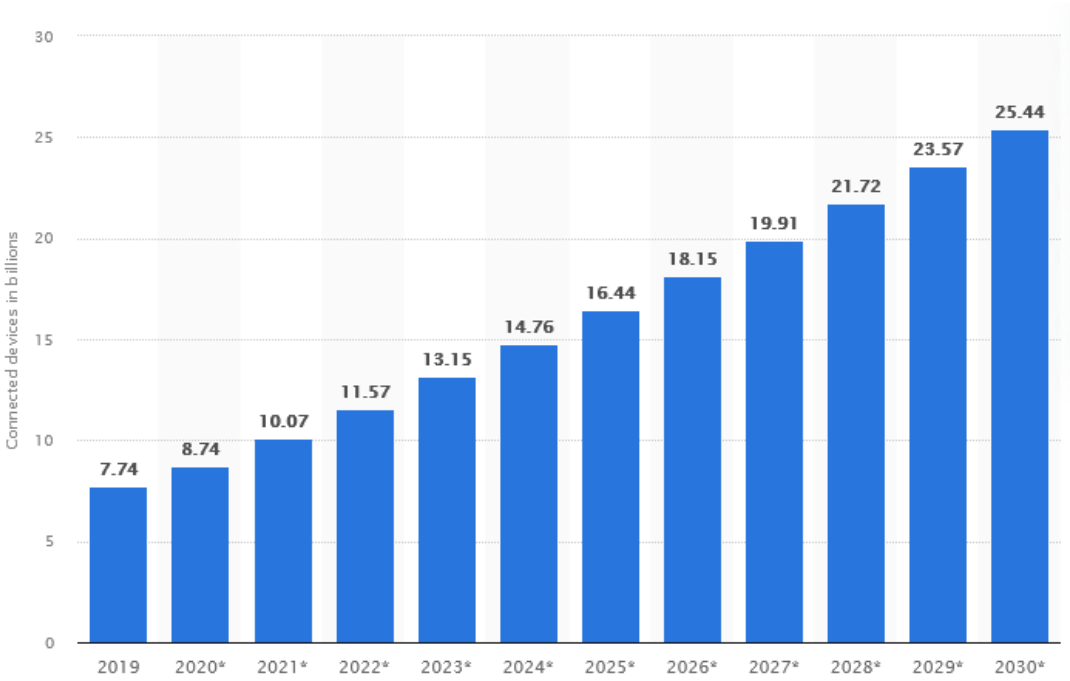


Fig 2.1: Evolution of IoT devices [3].

All these data and the way that they can be managed, bring along questions and challenges to be advised:

- 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

Cloud data providers are continuously expanding their infrastructure as cloud-native applications are becoming more and more popular. This is not a surprise as cloud capabilities benefit any professional business nowadays. Internet is available from any point of the Earth and sometimes in an exceptional speed, something that drove companies to go fully online and offer services to clients directly from a browser or an application on a smart device. The increase of IoT devices at the edge of the network is producing a massive amount of data to be computed at datacenters, pushing network bandwidth requirements to the limit. It also requires the store and process of information into geographically distant clouds so this computing model is not practical for the future, as it is likely to increase communication latencies, degrade the overall Quality-of-Service (QoS) and Quality-of-Experience (QoE).

There was a need of introducing an alternative computing model which will bring computations closer to the end-user. This would reduce the amount of data sent to the cloud, consequently reducing communication latencies and also decentralizing some of the computing resources available in large data centers by distributing them towards the edge of the network. This new computing model that makes use of resources located at the edge of the network (such as routers, gateways, and switches that are augmented with computing capabilities) is referred to as “edge computing”. A model that makes use of both edge resources and the cloud is referred to as “fog computing”.

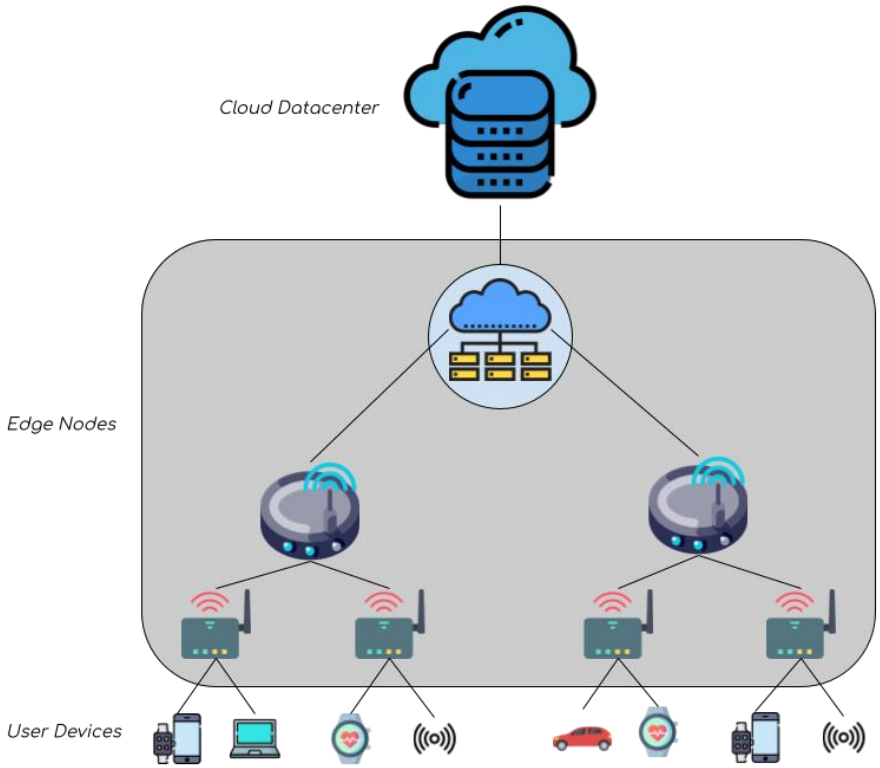


Fig 2.2: Edge Computing architecture.

As shown at Fig 2.2 edge computing forms a computing environment that uses low-power mobile devices, home gateways, home servers, edge ISP servers, and routers. These small-form-factor devices nowadays have competent computing capabilities and are connected to the network. The combination of these small compute servers enables a cloud computing environment that can be leveraged by a rich set of applications processing Internet of Things (IoT) and cyber-physical systems (CPS) data.

We can evaluate an edge environment by 5 factors:

1. Privacy and security: The distributed nature of this model introduces a shift in security schemes used in cloud computing. In edge computing, data may travel between different distributed nodes connected through the Internet and thus requires special encryption mechanisms independent of the cloud. Edge nodes may also be resource-constrained devices, limiting the choice in terms of security methods. Moreover, a shift from centralized top-down infrastructure to a decentralized trust model is required. On the other hand, by keeping and processing data at the edge, it is possible to increase privacy by minimizing the transmission of

sensitive information to the cloud. Furthermore, the ownership of collected data shifts from service providers to end-users.

2. Scalability: In scalability we must take into account the heterogeneity of the devices, having different performance and energy constraints, the highly dynamic condition, and the reliability of the connections compared to more robust infrastructure of cloud data centers. Moreover, security requirements may introduce further latency in the communication between nodes, which may slow down the scaling process.

3. Reliability: Error-handling is crucial in order to keep a service alive. If a single edge node goes down and is unreachable, users should still be able to access a service without interruptions. Moreover, edge computing systems must provide actions to recover from a failure and alerting the user about the incident. To this aim, each device must maintain the network topology of the entire distributed system, so that detection of errors and recovery become easily applicable. Other factors that may influence this aspect are the connection technologies in use, which may provide different levels of reliability, and the accuracy of the data produced at the edge that could be unreliable due to particular environment conditions.

4. Speed: Edge computing brings analytical computational resources close to the end users and therefore can increase the responsiveness and throughput of applications. A well-designed edge platform would significantly outperform a traditional cloud-based system. Some applications rely on short response times, making edge computing a significantly more feasible option than cloud computing. Examples range from IIoT to autonomous driving, anything health or human / public safety relevant, or involving human perception such as facial recognition, which typically takes a human between 370-620ms to perform.

5. Efficiency: Due to the nearness of the analytical resources to the end users, sophisticated analytical tools and Artificial Intelligence tools can run on the edge of the system. This placement at the edge helps to increase operational efficiency and is responsible for many advantages to the system. Additionally, the usage of edge computing as an intermediate stage between client devices and the wider internet results in efficiency savings that can be demonstrated in the following example: A client device requires computationally intensive processing on video files to be performed on external servers. By using servers located on a local edge network to

perform those computations, the video files only need to be transmitted in the local network. Avoiding transmission over the internet results in significant bandwidth savings and therefore increases efficiency.

Edge devices have smaller processors, limited power and sometimes they differ from each other in terms of computer architecture and OS. Therefore, managing the nodes and their resources is one of the key challenges in edge computing. An edge computing environment is defined by its' architecture model on data, control and tenancy:

1. Data flow architectures: These architectures are based on the direction of movement of workloads and data in the computing ecosystem. For example, workloads could be transferred from the user devices to the edge nodes or alternatively from cloud servers to the edge nodes.

2. Control architectures: These architectures are based on how the resources are controlled in the computing ecosystem. For example, a single controller or central algorithm may be used for managing a number of edge nodes. Alternatively, a distributed approach may be employed.

3. Tenancy architecture: These architectures are based on the support provided for hosting multiple entities in the ecosystem. For example, either a single application or multiple applications could be hosted on an edge node.

Edge computing infrastructure is defined by 3 main components:

1. Hardware: Any device can be equipped with single-board computers (SBCs) that offer considerable computing capabilities and can contribute to a larger cluster of nodes. Hardware consists of both computation devices that manipulate the data and network devices that control the data flow.

2. System software: How a node manages its resources and can operate and communicate directly with other fog/edge devices. System software is responsible for manager the machines' Operating System or the Virtual Machines and containers that are deployed on the machine (known as Virtualization Software).

3. Middleware: Software that runs on top of an operating system and provides complementary services that are not supported by the system software. The

middleware coordinates distributed compute nodes and performs deployment of virtual machines or containers to each fog/edge node.

Recent studies in fog/edge computing suggest exploiting small-form-factor devices such as network gateways, Wi-Fi Access Points (APs), set-top boxes, cars, and even drones as compute servers for resource efficiency. Edge computing also utilizes commodity products such as desktops, laptops, and smartphones.

In order to facilitate edge computing there is a need for implementing several algorithms with the most popular being:

- Discovery: Identifying edge resources within the network that can be used for distributed computation (so that workloads from the clouds or from user devices/ sensors can be deployed on them).

- Benchmarking: Capturing the performance of resources for decision-making in order to maximize the performance of deployments using standard performance evaluation tools. This is done by running sample micro or macro applications that stress-tests each entity to obtain a snapshot of the performance of a Virtual Machine (VM). There is a need for lightweight benchmarking tools for the edge not to over-stress and reduce the performance of a device.

- Load-balancing: Distributing workloads across resources based on different criteria such as priorities, fairness etc.

- Placement: Identifying resources appropriate for deploying a workload and place incoming computation tasks on suitable fog/edge resources.

As the demand of allocating computer resources is increased due to the large amount of data and the computationally heavy algorithms (such as machine learning or artificial intelligence algorithms) deployed and stored in IoT devices, there has been the need of running pieces of code that do not require pre-allocation of any resource. The solution came from the new developed cloud computing model called “serverless computing”.

2.3 Serverless Computing

Serverless Architecture is a relatively new cloud computing model which uses the event-trigger-action sequence model (Fig 2.3.1) and it is used to process requested functionality without pre-allocating computing resources. It delegates the management of the execution environment of an application (in the form of stateless functions) to the infrastructure provider. Containers, managed by the cloud provider, are used to execute functions (often called lambdas) which are event triggered and they last only for one invocation (state-less). Serverless is by no means an indication that there are no servers, it simply means that the developer should leave most operational concerns such as resource provisioning, monitoring, maintenance, scalability, and fault-tolerance to the cloud provider.

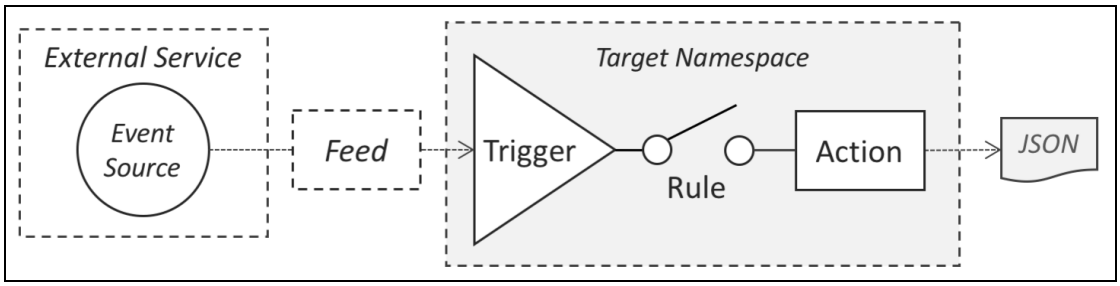


Fig 2.3.1: Serverless Computing Model [4].

- A feed is a convenient way to configure an external event source to fire trigger events that can be consumed by Cloud Functions. A feed is a stream of events that all belong to some trigger. Pre-installed packages, installable packages, and your own custom packages might contain feeds. A feed is controlled by a feed action, which handles creating, deleting, pausing, and resuming the stream of events that comprise a feed. The feed action typically interacts with external services that produce the events, by using a REST API that manages notifications.

- A trigger is a name for a class of events. Each event belongs to exactly one trigger; by analogy, a trigger resembles a topic in topic-based pub-sub systems.

- A rule is used to indicate that whenever an event from trigger arrives, invoke action with the trigger payload.

- An action is a set of instructions to be executed as soon as the previous feed-trigger-rule sequence invoke it.

Serverless architectures refer to the application architecture that abstracts away server management tasks from the developer and enhances development speed and efficiency by dynamically allocating and managing compute resources. Function as a service (FaaS) is a runtime on top of which a serverless architecture can be built.

A very simple example where serverless architecture's capabilities are pointed out is the procedure of automatically ordering new products in a supermarket (Fig 2.3.2). Let's say that there is an automated cloud-native tool that monitors products quantities on the store's shelves. An event can be the monitoring of a low availability on a single brand which will then trigger the action of making a new order for the specific brand to the warehouse.

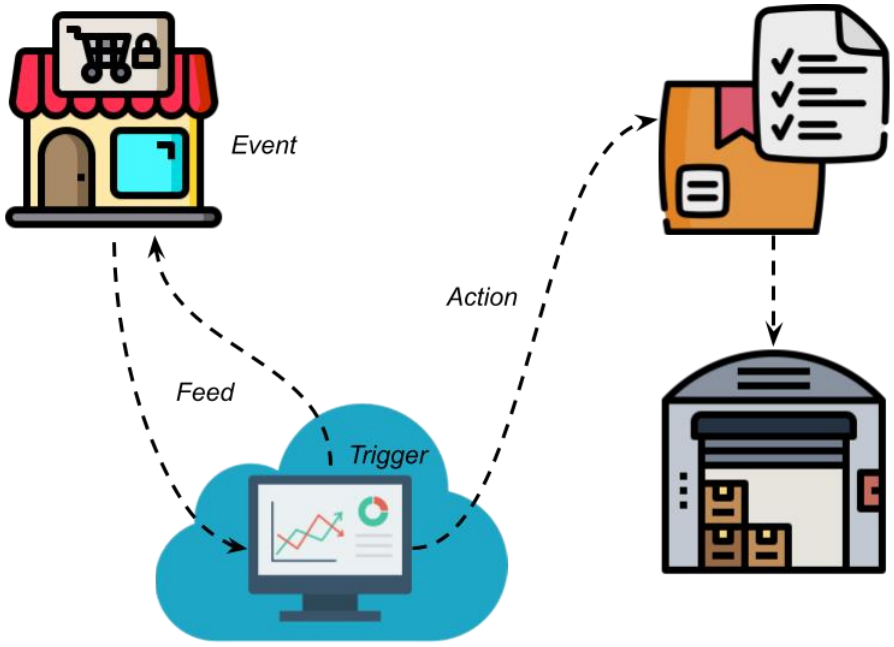


Fig 2.3.2: Serverless example.

This model allows the developer to write and deploy small pieces of cloud-native code that responds to events without considering the runtime environment, resource allocation, load balancing, and scalability and this is the main reason that the term "serverless" is very popular in Google Trends the past 5 years as shown at Fig 2.3.3.

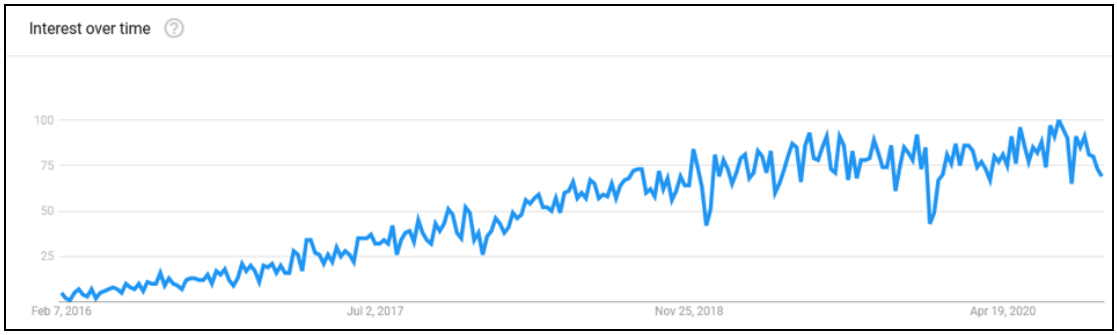


Fig 2.3.3: Popularity of the term "serverless" as reported by Google Trends.

Another major advantage is the pay-per-use billing model in contrast to the usual hour-based billing of virtual machines and containers that cloud providers use. Anyone that uses serverless infrastructure pays only for the time and resources his computations need, as functions share the same runtime environment (typically a pool of containers) while there is no cost to the consumer when the system is idle. Hence, the deployment of a pool of shared containers (workers) on a machine (or a cluster of machines) and the execution of some code onto any of them becomes inexpensive and efficient. It is important to mention that cloud providers report that their customers see cost savings of 4x-10x when moving applications to serverless. Because there is no charge when there are no events that invoke cloud functions, it's possible that serverless could be much less expensive. Moreover, as organizations use serverless more and more, they will be able to predict their serverless computing costs based on history, similar to the way they do today for other utility services, such as electricity.

In the world of cloud computing there are several ways to use a cloud infrastructure for the deployment of cloud native applications, either by allocating a machine-server or a VM (IaaS), a container (PaaS) or just provide your stateless function and the computations will be executed on the runtime and OS you want (SaaS). Serverless Architecture and more specific the Software as a Service model that cloud providers use nowadays give the user scalability (horizontal) on demand and just when it is needed; in those sudden spikes of traffic. The user can also have access to the software or app from any point of the earth and he does not have to worry about updates, new versions of the OS etc.

In this way serverless platforms force application developers to carefully think about the cost of their code when modularizing their applications, rather than latency, scalability, and elasticity, which is where significant development effort has traditionally been spent. On the other hand, deploying such applications in a serverless platform is challenging as developers

have to care fully understand how the platform behaves and design the application around these. It also requires relinquishing to the platform design decisions that concern, among other things, quality-of-service (QoS) monitoring, scaling, and fault-tolerance properties.

2.4 Virtualization and Deployment

In the previous sections, Edge computing paradigm has been introduced. In this kind of architecture, someone might wonder how these applications will be packaged and deployed in such a heterogeneous environment. This is achieved 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.

Virtualization allows multiple operating systems to run on a single physical machine and enables fault and performance isolation between multiple tenants in the edge. It partitions resources for each tenant so one tenant cannot access other tenants' resources. The fault of a tenant, therefore, cannot affect other tenants. Virtualization also limits and accounts for the resource usage of each tenant so a tenant cannot monopolize all the available resources in the system.

There are two ways for someone to perform virtualization which are described below:

2.4.1 Virtual Machines

Virtual Machines is the most widely used way of deploying an application to a device that you know nothing about. A Virtual Machine (VM) is a set of virtualized resources used to emulate a physical computer. Virtualized resources include CPUs, memory, network, storage devices, and even GPUs and FPGAs. The architecture behind every VM is as shown below at Fig 2.4.1.

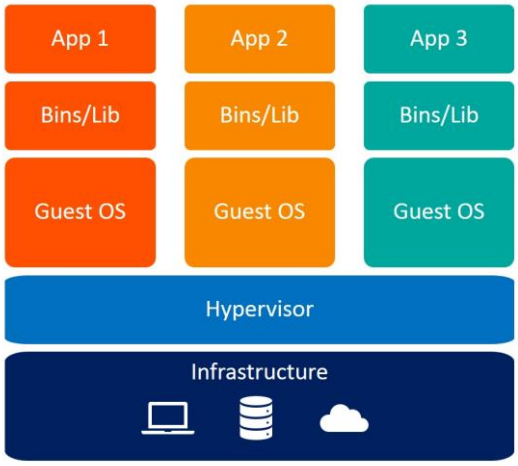


Fig 2.4.1: Virtual Machine Architecture.

Applications have their own dependencies (Libraries, executables etc.) in order to run as designed in a completely different system than host machine they were initially developed and tested. Virtualization software called a hypervisor virtualizes the physical resources and provides the virtualized resources in the form of a VM.

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.

2.4.2 Containers

Containers are an emerging technology for cloud computing that provides process-level lightweight virtualization. Containers are multiplexed by a single Linux kernel so they do not require an additional virtualization layer compared to virtual machines.

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 to run independent, reliable and as designed, when moved from one computing environment to another.

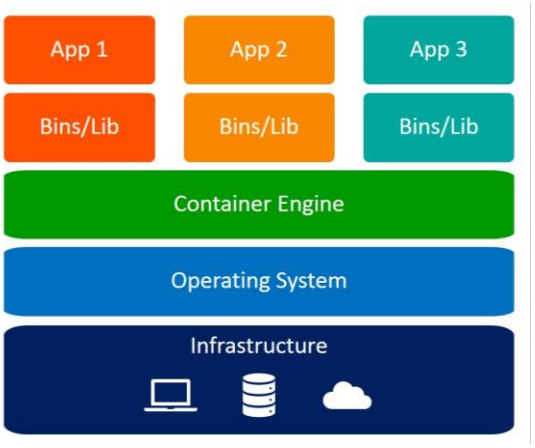


Fig 2.4.2: Containers Architecture.

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.

This lightweight virtualization allows containers to start and stop rapidly and to achieve performance similar to that of the native environment.

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).

2.5 Thesis Overview

In this thesis, we present a tool which proposes a way in which the individual functions of a monolithic code could run in a serverless environment as individual and independent application packaged containers, so that given a maximum runtime threshold, the minimum possible power consumption in our cluster is achieved. Our serverless infrastructure (cluster) consists of Aarch64 Edge devices which are orchestrated by Kubernetes. Finally, using state-of-the-art frameworks such as Openfaas and the open source containerization platform Docker we managed to package our applications into containers and run them in our serverless environment.

Our tool analyzes (profiles) the code file given in terms of memory requirements and the execution time of each of its individual functions on our testing device (personal computer) and then using machine learning techniques (Linear Regression and Decision Tree Regression models) predicts the energy consumption and the execution time of each function at each of our devices. The utilization of this information and the decision for the proposed solution is achieved through our developed algorithm. Furthermore, using Openfaas, Kubernetes and Docker we show 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 Openfaas as a (Function as a Service) framework for building serverless functions on top of containers (using Docker and Kubernetes). In Chapter 5, we present our approach to the stated problem. In Chapter 6, we present our experimental infrastructure and the experimental evaluation of our approach. Finally, in chapter 7 we conclude and propose future work in order to improve our tool.

3

Related Work

Resource management in cloud infrastructure and datacenters has been researched by a lot of scientists and professionals during the past years. As the amount of data being processed at the datacenters all around the world increases the need for high performance while reducing and controlling power consumption is essential. Moving to the fog/edge resources which are typically resource constrained, heterogeneous, and dynamic compared to the cloud, resource management on those devices is also an important challenge that needs to be addressed.

Cheol-Ho Hong and Blesson Varghese [5] review publications to identify and classify the architectures, infrastructure, and underlying algorithms for managing resources in fog/edge computing. They make clear and fully understandable the reason fog/edge computing has gained significant attention over the past few years as an alternative approach to the conventional centralized cloud computing model, while describing in detail resource management architectures, the dataflow, control, tenancy architectures, the infrastructure used for managing resources, such as the hardware, system software, and middleware.

Authors of [6], using static analysis propose a flexible tool that enables the estimation of performance and energy consumption of the application on embedded devices, providing a

complete methodology based on which the user can add estimation models for various platforms. Facing the problem of how to adapt the traditional software development model into the cloud-native model authors at [7] propose a tool for Java source code analysis, transformation and deployment of the model and code automatically in a FaaS infrastructure.

Authors at [8] addresses the technical challenges and open problems of existing serverless platforms from industry, academia, and open source projects, identify key characteristics and use cases.

Babak Bashari Rad et al. research the performance and the containerized development using docker and its tools at [9] while concluding the advantages and disadvantages of the platform.

Authors at [10], discuss the effective placement of virtual machines in a cluster of physical machines in order to optimize the use of computational resources and reduce the probability of virtual machine reallocation. They propose an approach based on the multiple multidimensional knapsack problem, where the main concern is to maximize placement ratio.

Authors at [11], address the problem of growing demand on cloud computing and the need to try whatever is possible to meet the customers' requirements for resources consuming minimum power. They use the modeling of the multiple knapsack problem, with a mechanism for allocating resources, which addresses the issue of energy saving. Furthermore, a comparative analysis of the proposed solution with the original mechanism to evaluate the performance modification is made.

In paper [12], authors propose a methodology for evaluating serverless frameworks deployed on hybrid edge-cloud clusters. The methodology focuses on key performance knobs of the serverless paradigm and applies a systematic way for evaluating these aspects in hybrid edge-cloud environments. Methodology is also applied on three open-source serverless frameworks, OpenFaaS, Openwhisk, and Lean Openwhisk respectively, and they provide key insights regarding their performance implications over resource-constrained edge devices.

In this work [13], we propose a methodology for application decomposition into fine-grained functions and energy-aware function placement on a cluster of edge devices subject to user-specified QoS guarantees. While the main concern in such environments is the minimization of energy consumption, the heterogeneity in compute resources found at the edge may lead to Quality of Service (QoS) violations and at the same time, Serverless computing, the next frontier of Cloud computing has emerged to offer unprecedented elasticity by utilizing fine-grained, stateless functions.

In overview of the literature, resource allocation and energy consumption is a quite popular problem and lots of researchers are trying to bring solutions to the table. However, dividing an application into several serverless functions, predicting the energy and run-time of every function of the application before they are scheduled and proposing the way each one should be placed on the edge devices of a cluster in order to achieve the lowest energy consumption under a specific time threshold have never been proposed. Adding to this, we are using OpenFaaS, a very promising serverless platform, which can be used in lots of IIoT and IoT serverless environment scenarios.

4

Technical background

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

4.1 Docker: A container runtime

Docker is an open-source engine that automates the deployment of applications into containers. Docker actually adds an application deployment engine on top of a virtualized container execution environment and the applications that are built in the docker are packaged with all the supporting dependencies into a standard form (container). These containers keep running in an isolated way on top of the operating system's kernel. The way that docker is designed is to give a quick and a lightweight environment where code can be run efficiently and moreover it provides an extra facility of the proficient work process to take the code from the computer for testing before production.

There are four main internal components of docker, including Docker Client and Server, Docker Images, Docker Registries, and Docker Containers as shown at Fig 4.1. These components will be explained in more detail in the following sections.

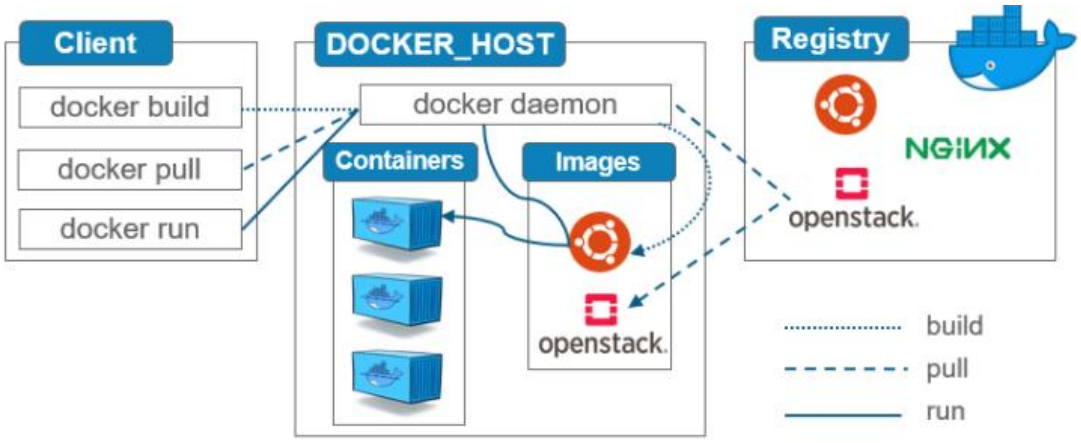


Fig 4.1: Docker Architecture [14].

4.1.1 Docker Client and Server

Docker can be explained as a client and server based application. The docker server gets the request from the docker client through the docker command and then process it accordingly. The complete RESTful (Representational state transfer) API and a command line client binary are shipped by docker. Docker daemon/server and docker client can be run on the same machine or a local docker client can be connected with a remote server or daemon, which is running on another machine. Also the client can communicate with more than one daemon.

4.1.2 Docker Images

There are two methods to build an image. The first one is to build an image by using a read-only template. The foundation of every image is a base image. Operating system images are basically the base images, such as Ubuntu 14.04 LTS, or Fedora 20. The images of operating system create a container with an ability of complete running OS. Base image can also be created from scratch. Required applications can be added to the base image by modifying it, but it is necessary to build a new image. The process of building a new image is called “committing a change”.

The second method is to create a docker file. The docker file contains a list of instructions when “Docker build” command is run from the bash terminal it follows all the instructions given in the docker file and builds an image. This is an automated way of building an image.

4.1.3 Docker Registry

Docker images are placed in docker registries. It works correspondingly to source code repositories where images can be pushed or pulled from a single source. There are two types of registries, public and private. Docker Hub is called a public registry where everyone can pull available images and push their own images without creating an image from scratch.

4.1.4 Docker Containers

Docker image creates a docker container. Containers hold the whole kit required for an application, so the application can be run in an isolated way. For example, suppose there is an image of Ubuntu OS with SQL SERVER, when this image is run with docker run command, then a container will be created and SQL SERVER will be running on Ubuntu OS.

4.2 Kubernetes: A container orchestrator

Kubernetes (or k8s) 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 and has become the standard container orchestration solution. Kubernetes services support and tools are widely available. In simpler terms, Kubernetes can be used in order to manage huge workloads in enormous clusters with ease.

Kubernetes is a system that enables a container-based deployment within Platform-as-a-Service (PaaS) clouds, focusing specifically on cluster-based systems. It can provide a cloud-native application, a distributed and horizontally scalable system composed of

(micro)services, with operational capabilities such as resilience and elasticity support. From an architectural point of view, Kubernetes introduces the *pod* concept, a group of one or more containers (e.g. Docker, or any OCI compliant container system) with shared storage and network.

A Container Management Platform (CMP) (as shown at Fig 4.2.1) contains functional layers that work in order to deliver all the capabilities you need to build and manage a Kubernetes infrastructure.

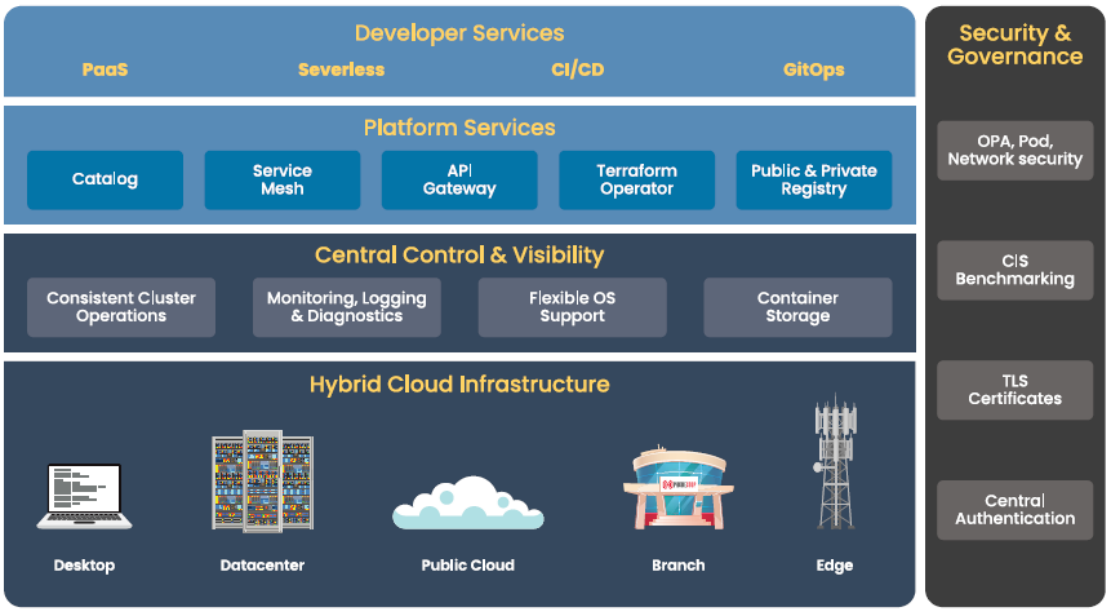


Fig 4.2.1: The anatomy of a container management platform.

A Kubernetes cluster consists of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node. The worker node(s) host the Pods that are the components of the application workload. The Control Plane manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.

Now, we will describe Kubernetes's main components. As shown in Fig 4.2.2, Kubernetes is divided in Control Plane Components and Node Components.

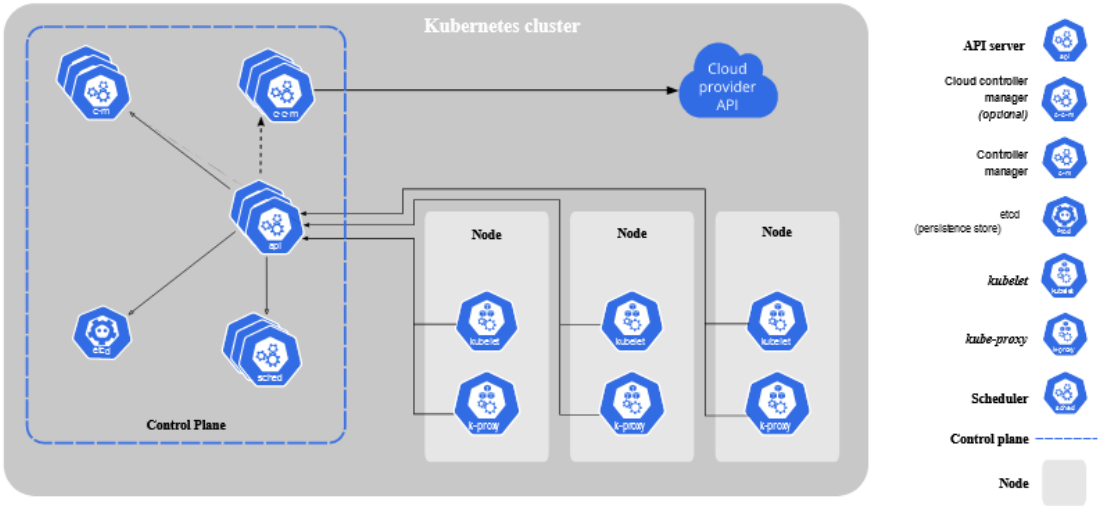


Fig 4.2.2: Kubernetes cluster main components [15].

4.2.1 Control Plane (Master) Components

- ETCD: It is a simple, distributed key value storage which is used to store the Kubernetes cluster data and metadata (such as number of pods, their state, namespace, etc.), API objects and service discovery details. It is only accessible from the API server for security reasons. ETCD enables notifications to the cluster about configuration changes with the help of watchers. Notifications are API requests on each ETCD cluster node to trigger the update of information in the node's storage.

- Kube Api Server: Kubernetes API is actually the front end of the control plane and, by extension, of the whole environment. It is the central management entity that receives all REST requests for modifications (to pods, services, replication sets/controllers and others), serving as frontend to the cluster. Also, this is the only component that communicates with the ETCD cluster, making sure data is stored in ETCD and is in agreement with the service details of the deployed pods. It is also designed to scale horizontally and be friendly to the user

- 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. Kube controller-manager runs a number of distinct controller processes in the background (for example, replication controller controls number of replicas in a pod, endpoints

controller populates endpoint objects like services and pods, and others) to regulate the shared state of the cluster and perform routine tasks. When a change in a service configuration occurs (for example, replacing the image from which the pods are running, or changing parameters in the configuration yaml file), the controller spots the change and starts working towards the new desired state.

- Kube Scheduler: This is the tool responsible to schedule the pods (the co-located group of containers inside which our application processes are running) on the various nodes based on resource utilization. It reads the service's operational requirements and schedules it on the best fit node. For example, if the application needs 2GB of memory and 4 CPU cores, then the pods for that application will be scheduled on a node with at least those resources. Pods are the smallest and most common deployable units that anyone can create in a Kubernetes Cluster. The scheduler runs each time there is a need to schedule pods. The scheduler must know the total resources available as well as resources allocated to existing workloads on each node.

4.2.2 Node (Worker) components

- kubelet: The main service on a node, regularly taking in new or modified pod specifications (primarily through the kube-apiserver) and ensuring that pods and their containers are healthy and running in the desired state. This component also reports to the master on the health of the host where it is running.

- kube-proxy: A proxy service that runs on each worker node to deal with individual host subnetting and expose services to the external world. It performs request forwarding to the correct pods/containers across the various isolated networks in a cluster.

At Fig 4.2.3 we can see the components described above in a graph of a simple example of a kubernetes cluster with 3 devices, 1 master node and 2 worker nodes (workers):

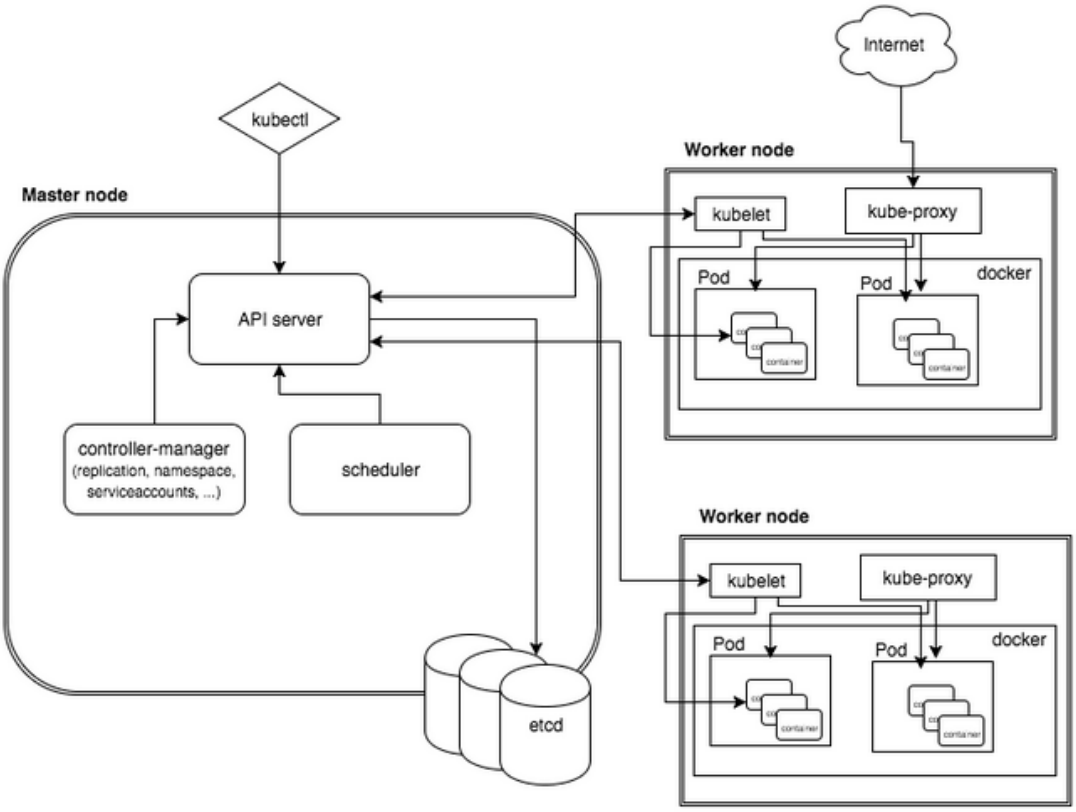


Fig 4.2.3: Kubernetes cluster main components in more detail [16].

4.2.3 k8s vs k3s

While Kubernetes (or k8s) is designed to accommodate large configurations (approximately up to 5000 nodes) and helps deploy applications in production there has been the need for developing a lightweight version of k8s, in order to run clusters in IoT devices such as Raspberry Pi, NVIDIA Jetson, etc. which have limited resources. Normal Kubernetes (or k8s) is not operable in IoT or edge computing devices due to its' heavy size and its' memory footprint. This is why Rancher Labs developed k3s (which is the one we used in our kubernetes cluster) a lightweight Kubernetes distribution, which is a fully certified Kubernetes offering by CNCF (Cloud Native Computing Foundation).

With that being said it is now very interesting to point out some of the major differences between these two versions of Kubernetes:

- K3s are not functionally different from K8s, but they have some differences that make them unique. K3s can deploy applications faster than kubernetes. Not only that, K3s can spin up clusters more quickly than K8s. K8s is a general-purpose container orchestrator, while K3s is a purpose-built container orchestrator for running Kubernetes on bare-metal servers.

- Kubernetes uses kubelet, an agent running on each Kubernetes node to perform a control loop of the containers running on that node. This agent runs inside the container. However, K3s does not use kubelet, but it runs kubelet on the host machine and uses the host's scheduling mechanism to run containers.

- Kubernetes or K8s can host workloads running across multiple environments, while K3s can only host workloads running in a single cloud. It mainly happens because K3s don't contain the capacity to maintain a significant workload on multiple clouds as it is small in size.

- K3s uses kube-proxy to proxy the network connections of the Kubernetes nodes, but K8s uses kube-proxy to proxy the network connections of an individual container. It also uses kube-proxy to set up IP masquerading, while K3s does not use kube-proxy to do that.

- K8s uses kubelet to watch the Kubernetes nodes for changes in the configuration, while K3s does not watch Kubernetes nodes for changes in the configuration. Instead, it receives a deployment manifest containing the configuration information from the Kubernetes control plane and makes changes accordingly.

- K3s can have tighter security deployment than k8s because of their small attack surface area.

- K3s reduces the dependencies and steps needed to install, run or update a Kubernetes cluster (You can install and deploy k3s with one command under 30 seconds).

4.3 OpenFaaS

OpenFaaS (Function as a Service) is a framework for building serverless functions on the top of containers (with docker and kubernetes). With the help of OpenFaaS, it is easy to turn anything that runs on Linux or Windows, into a serverless function, through Docker and Kubernetes. It provides a built-in functionality such as self-healing infrastructure, auto-scaling, and the ability to control every aspect of the cluster. Basically, OpenFaaS is a concept of decomposing applications into a small unit of work.

In order to understand more clearly the innovation of this technology and serverless computing in general we can analyze the 3 main architectural patterns (ways of building systems) as shown at Fig 4.3.1.

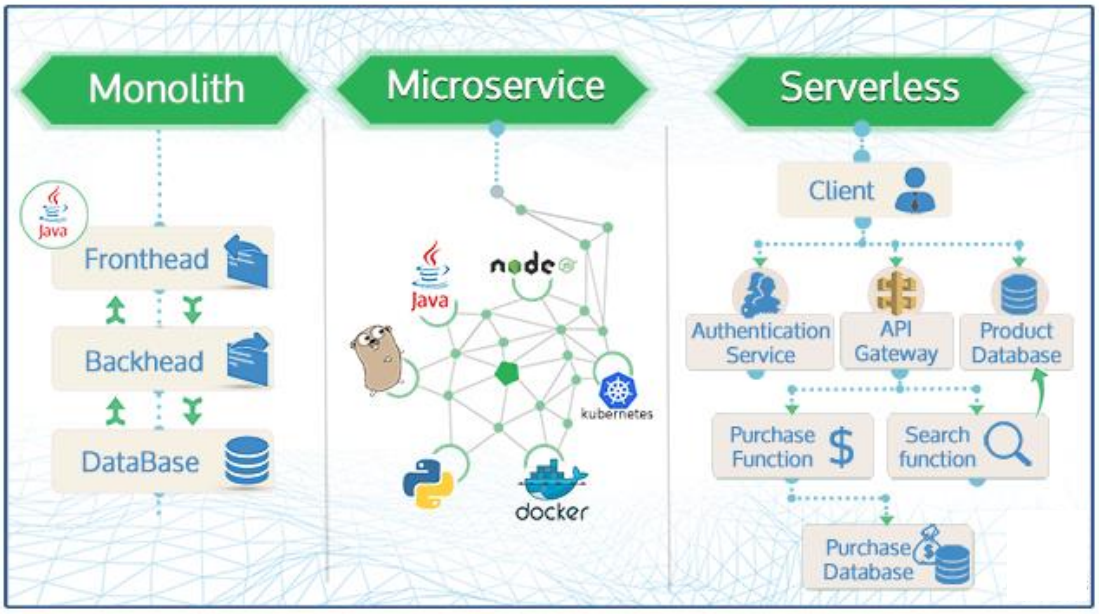


Fig 4.3.1: Evolution of architectural patterns [17].

- Monolith code: At the very start, we used to build monoliths, three-tier applications. They were very heavyweight, slow to deploy and had trouble testing them.

- Microservices: After monolith, we broke these down into microservices. They focused on being compostable and they are deployed with Docker containers.

- Serverless: Functions are small, discrete, reusable pieces of code that can easily be deployed. Those functions are stateless and can run in a very short time.

It is important to point out that serverless functions do not replace our monolith code or microservices, they work best alongside our existing systems building integrations and helping events flow between our ecosystem.

The main goal of OpenFaas is to simplify serverless functions with Docker containers, allowing us to run complex and flexible infrastructures. OpenFaaS is publicly available and a free open-source environment for creating and hosting serverless functions. In more detail this platform allows us to:

- Reduce idle resources.
- Quickly process data.
- Interconnect with other services.
- Balance load with intensive processing of a large number of requests.

However, despite the advantages of OpenFaaS and serverless computing in general, developers must assess the application's logic before starting an implementation. This means that you must first break the logic down into separate tasks, and only then can you write any code.

4.3.1 OpenFaaS Design & Architecture

OpenFaaS architecture is based on a cloud-native standard and includes the following components: API Gateway, Function Watchdog, and the container orchestrators Kubernetes, Docker Swarm, Prometheus, and Docker. According to the architecture shown below at Fig 4.3.2, when a developer works with OpenFaaS, the process begins with the installation of Docker and ends with the Gateway API.

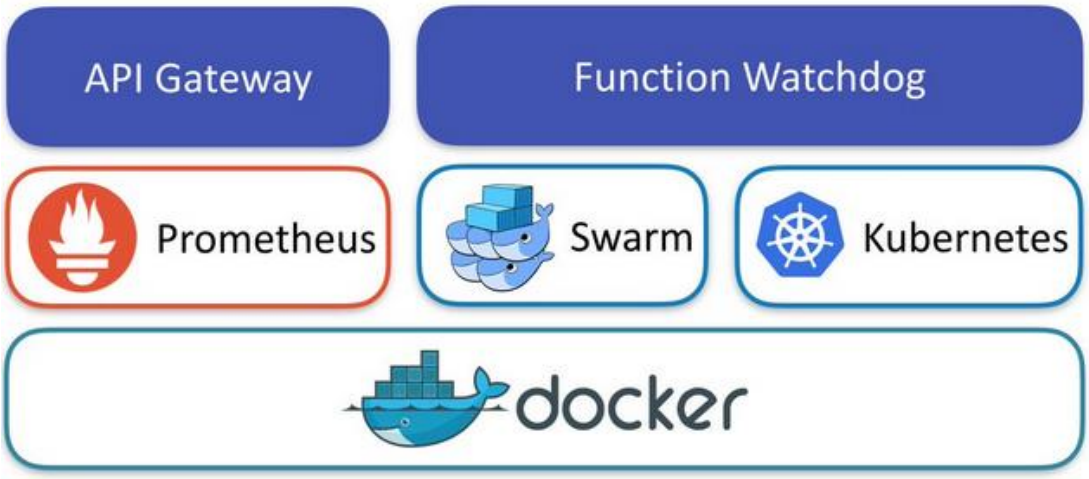


Fig 4.3.2: OpenFaaS architecture [18].

- API Gateway: Through the API Gateway, a route to the location of all functions is provided, and cloud-native metrics are collected through Prometheus. API Gateway will scale functions according to demand by altering the service replica count in the Docker Swarm or Kubernetes API. A UI is baked in allowing you to invoke functions in the browser and create new ones when it is needed.

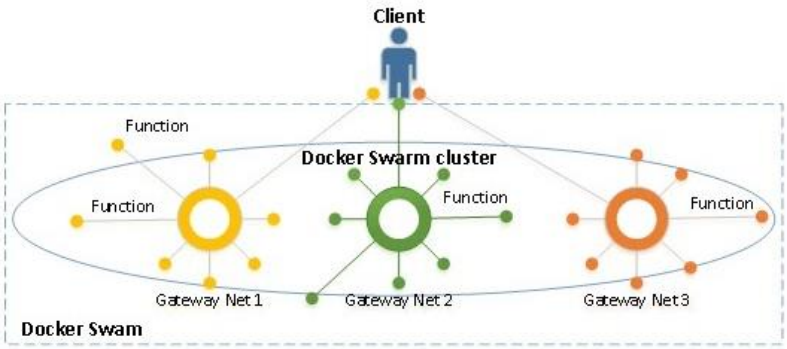


Fig 4.3.3: Client to Functions Routing [19].

- Function Watchdog: A Watchdog component is integrated into each container to support the serverless application functions and provides a common interface between the user and the function. Any Docker image can be turned into a serverless function by adding the Function Watchdog (a tiny Golang HTTP server). The Function Watchdog is the entrypoint allowing HTTP requests to be forwarded to the target process via STDIN. The response is sent back to the caller by writing to STDOUT from the application.

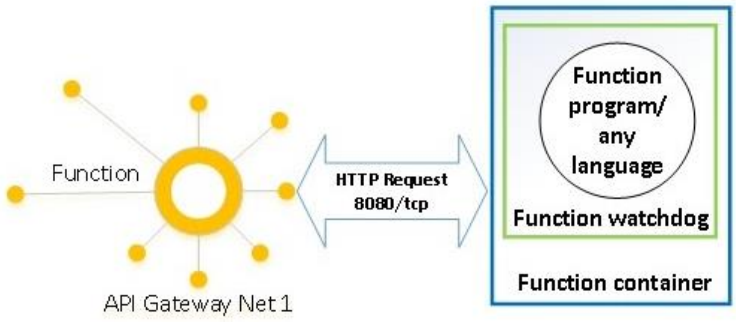


Fig 4.3.4: OpenFaaS Watchdog Interface [20].

- Prometheus: This component allows you to get the dynamics of metric changes at any time, compare them with others, convert them, and view them in text format or in the form of a graph without leaving the main page of the web interface. Prometheus stores the collected metrics in RAM and saves them to a disk upon reaching a given size limit or after a certain period of time.

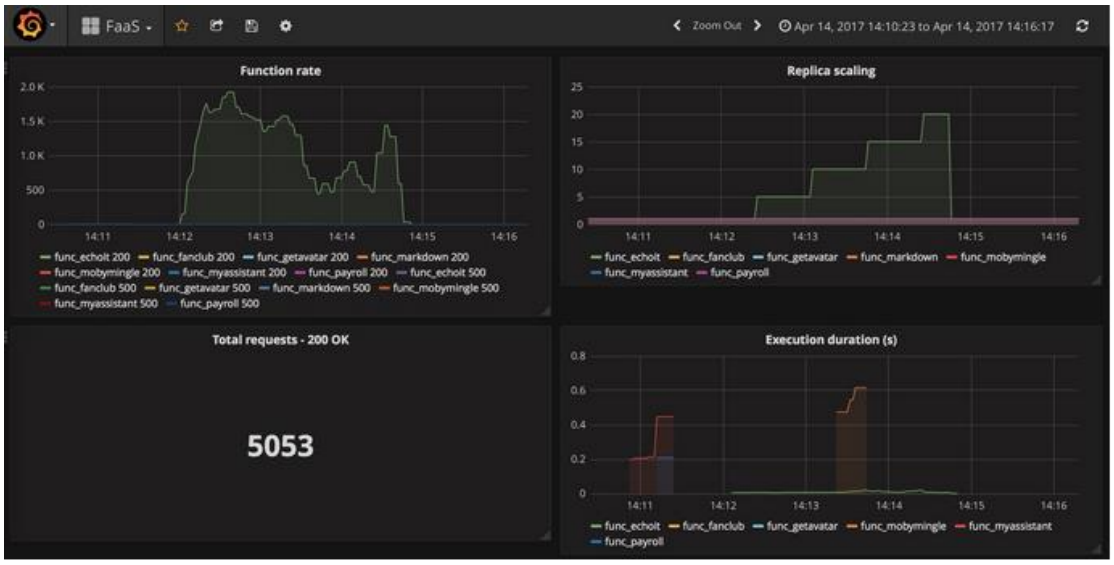


Fig 4.3.5: Example of a Prometheus/Grafana dashboard linked to OpenFaaS showing auto-scaling live in action.

- Docker Swarm and Kubernetes are the engines of orchestration. Components such as the API Gateway, the Function Watchdog, and an instance of Prometheus work on top of these orchestrators. Kubernetes is used to develop products, while Docker Swarm is used to create local functions. Moreover, all developed functions,

microservices, and products are stored in the Docker container, which serves as the main OpenFaaS platform for developers and sysadmins to develop, deploy, and run serverless applications with containers.

As we mentioned above any container or process in a Docker container can be a serverless function in FaaS. Using the FaaS CLI, OpenFaaS gives us the opportunity to deploy our functions or quickly create new functions from templates such as Node.js or Python as shown at Fig 4.3.6. So, new functions can be generated using the FaaS-CLI and built-in templates or through any binary for Windows or Linux in a Docker container.

Name	Language	Version	Linux base	Watchdog	Link
dockerfile	Dockerfile	N/A	Alpine Linux	classic	Dockerfile template
go	Go	1.15	Alpine Linux	classic	Go template
node12	NodeJS	12	Alpine Linux	of-watchdog	NodeJS template
node14	NodeJS	14	Alpine Linux	of-watchdog	NodeJS template
node14	NodeJS	16	Alpine Linux	of-watchdog	NodeJS template
node14	NodeJS	17	Alpine Linux	of-watchdog	NodeJS template
node	NodeJS	12	Alpine Linux	classic	NodeJS template
python3	Python	3	Alpine Linux	classic	Python 3 template
python3-debian	Python	3	Debian Linux	classic	Python 3 Debian template
python	Python	2.7	Alpine Linux	classic	Python 2.7 template
java11-vert-x	Java and Vert.x	11	Debian GNU/Linux	of-watchdog	Java LTS template
java11	Java	11	Debian GNU/Linux	of-watchdog	Java LTS template
ruby	Ruby	2.7	Alpine Linux 3.11	classic	Ruby template
php7	PHP	7.4	Alpine Linux	classic	PHP 7 template
csharp	C#	N/A	Debian GNU/Linux 9	classic	C# template

Fig 4.3.6: Classic OpenFaaS templates.

4.4 Machine Learning – Predictive Modeling

Machine learning is an application of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from experience without being explicitly programmed. It focuses on the use of data and algorithms to imitate the way that humans learn, gradually improving its accuracy.

We can divide the learning system of a machine learning algorithm into three main parts:

1. **A Decision Process:** In general, machine learning algorithms are used to make a prediction or classification. Based on some input data, which can be labelled or unlabeled, the algorithm will produce an estimate about a pattern in the data.
2. **An Error Function:** An error function serves to evaluate the prediction of the model. If there are known examples, an error function can make a comparison to assess the accuracy of the model. This is why there is a test set and a training set at every machine learning algorithm implementation. The training set is used for the predictions model while the test set is there to evaluate the model by comparing the actual values with the predicted ones.
3. **A Model Optimization Process:** If the model can fit better to the data points in the training set, then weights are adjusted to reduce the discrepancy between the known example and the model estimate. The algorithm will repeat this evaluate and optimize process, updating weights autonomously until a threshold of accuracy has been met.

Machine Learning can be divided in three main categories in terms of learning environment as shown at Fig 4.4.1:

1. **Supervised Learning:** Supervised learning describes a class of problem that involves using a model to learn a mapping between input examples and the target variable. Models are fit on training data comprised of inputs and outputs and used to make predictions on test sets where only the inputs are provided and the outputs from the model are compared to the withheld target variables and used to estimate the skill of the model.
2. **Unsupervised Learning:** Unsupervised learning describes a class of problems that involves using a model to describe or extract relationships in data. Compared to

supervised learning, unsupervised learning operates upon only the input data without outputs or target variables.

3. **Reinforcement Learning:** Reinforcement learning describes a class of problems where an agent operates in an environment and must learn to operate using feedback. The use of an environment means that there is no fixed training dataset, rather a goal or set of goals that an agent is required to achieve, actions they may perform, and feedback about performance toward the goal.

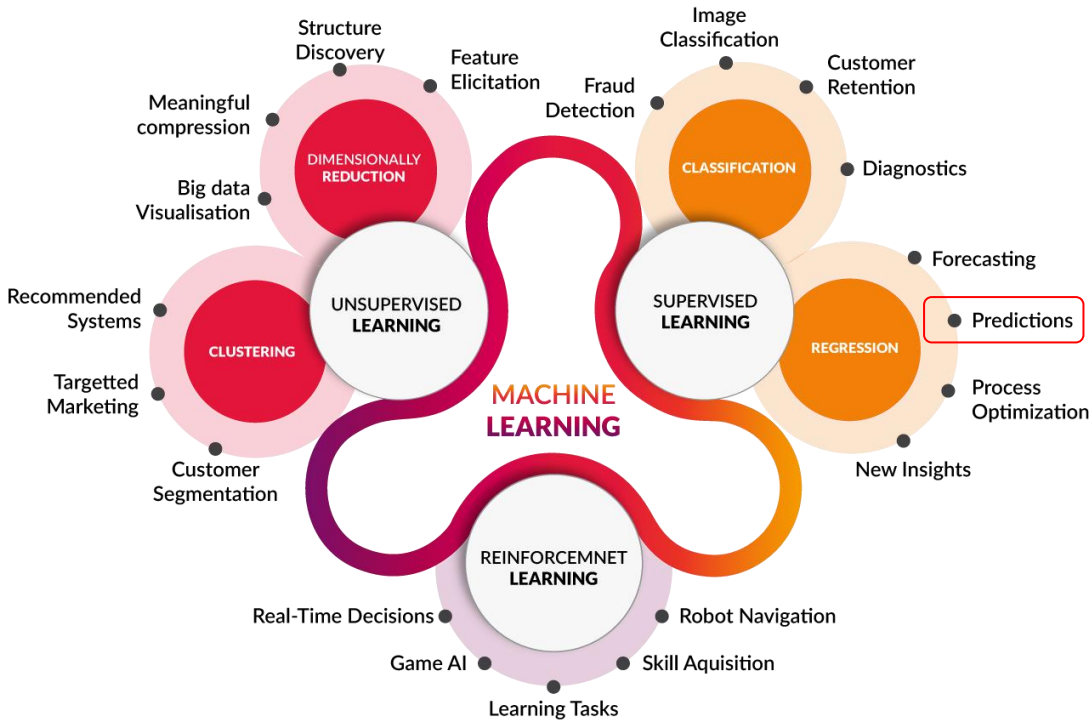


Fig 4.4.1 Description of Machine Learning.

Machine learning, more specifically the field of predictive modeling is primarily concerned with minimizing the error of a model or making the most accurate predictions possible, at the expense of explainability. In our proposal we used two models of predictive modeling, which will be described below, in order to predict the energy consumption and the runtime of our functions before assigning them to a node of the cluster. Predictions are the major factor with which the final decision of the algorithm is calculated. We will also describe the reason we chose these specific models as there is a number of predictive models available.

4.4.1 Linear Regression

Linear regression is a linear model, that assumes a linear relationship between the input variables (x) and the single output variable (y). More specifically, that y can be calculated from a linear combination of the input variables (x). Furthermore, linear regression has been around for more than 200 years and it has been studied from every possible angle.

When there is a single input variable (x), the method is referred to as *simple linear regression*. When there are multiple input variables, literature from statistics often refers to the method as *multiple linear regression*.

The representation is a linear equation that combines a specific set of input values (x) the solution to which is the predicted output for that set of input values (y). As such, both the input values (x) and the output value are numeric. For example, in a simple regression problem (a single x and a single y), the form of the model would be:

$$y = B_0 + B_1 * x$$

, where B_i represent coefficients (scale factor for each input value)

For example, let us use B0 = 0.1 and B1 = 0.5 and calculate the weight (in kilograms) for a person with the height of 182 centimeters.

$$weight = 0.1 + 0.5 * 182$$

$$weight = 91.1 \text{ (kg)}$$

Below at Fig 4.4.2 we can see the visual representation of the described equation:

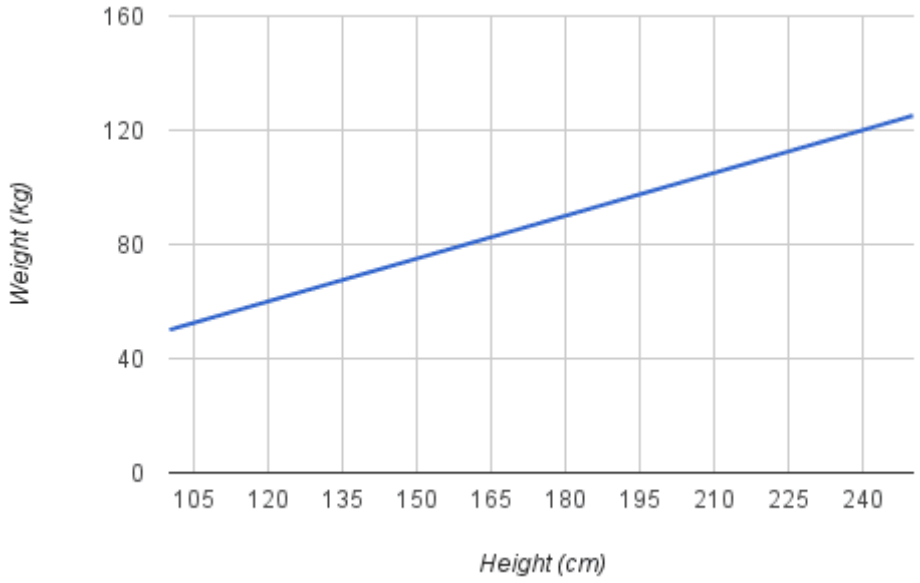


Fig 4.4.2 Example Height Vs Weight Linear Regression.

In our case we have two input values, memory and time as it will be described below, so our multiple linear regression form would be:

$$y = B_0 + B_1 * x_1 + B_2 * x_2$$

, where B_i represent coefficients

, x_i represents inputs (time and memory)

, y represents the predicted energy or the predicted run-time on the edge device

4.4.2 Decision Tree Regression

Decision Tree is a tree-structured classifier with three types of nodes. As shown at Fig 4.4.3 The Root Node is the initial node which represents the entire sample and may get split further into further nodes. The Interior Nodes represent the features of a data set and the branches represent the decision rules. Finally, the Leaf Nodes represent the outcome. This algorithm is very useful for solving decision-related problems.

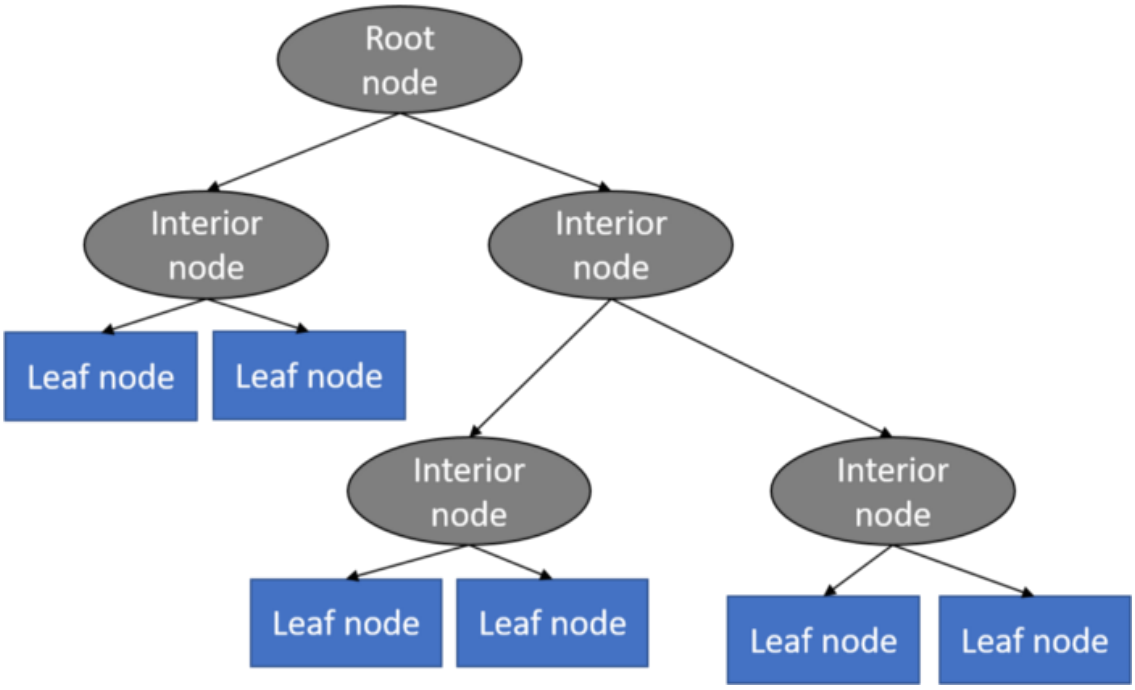


Fig 4.4.3 Decision Tree Regression model diagram.

In a decision tree, for predicting the class of the given dataset, the algorithm starts from the root node of the tree. This algorithm compares the values of root attribute with the record (real dataset) attribute and, based on the comparison, follows the branch and jumps to the next node.

For the next node, the algorithm again compares the attribute value with the other sub-nodes and move further. It continues the process until it reaches the leaf node of the tree. The complete process can be better understood using the below algorithm:

1. **Step-1:** Begin the tree with the root node, says S, which contains the complete dataset.
2. **Step-2:** Find the best attribute in the dataset using Attribute Selection Measure (ASM).
3. **Step-3:** Divide the S into subsets that contains possible values for the best attributes.
4. **Step-4:** Generate the decision tree node, which contains the best attribute.
5. **Step-5:** Recursively make new decision trees using the subsets of the dataset created in step-3. Continue this process until a stage is reached where you cannot further classify the nodes and called the final node as a leaf node.

As an example (Fig 4.4.4) let us suppose there is a candidate who has a job offer and wants to decide whether he should accept the offer or Not. So, to solve this problem, the decision

tree starts with the root node (Salary attribute by ASM). The root node splits further into the next decision node (distance from the office) and one leaf node based on the corresponding labels. The next decision node further gets split into one decision node (Cab facility) and one leaf node. Finally, the decision node splits into two leaf nodes (Accepted and Declined offer).

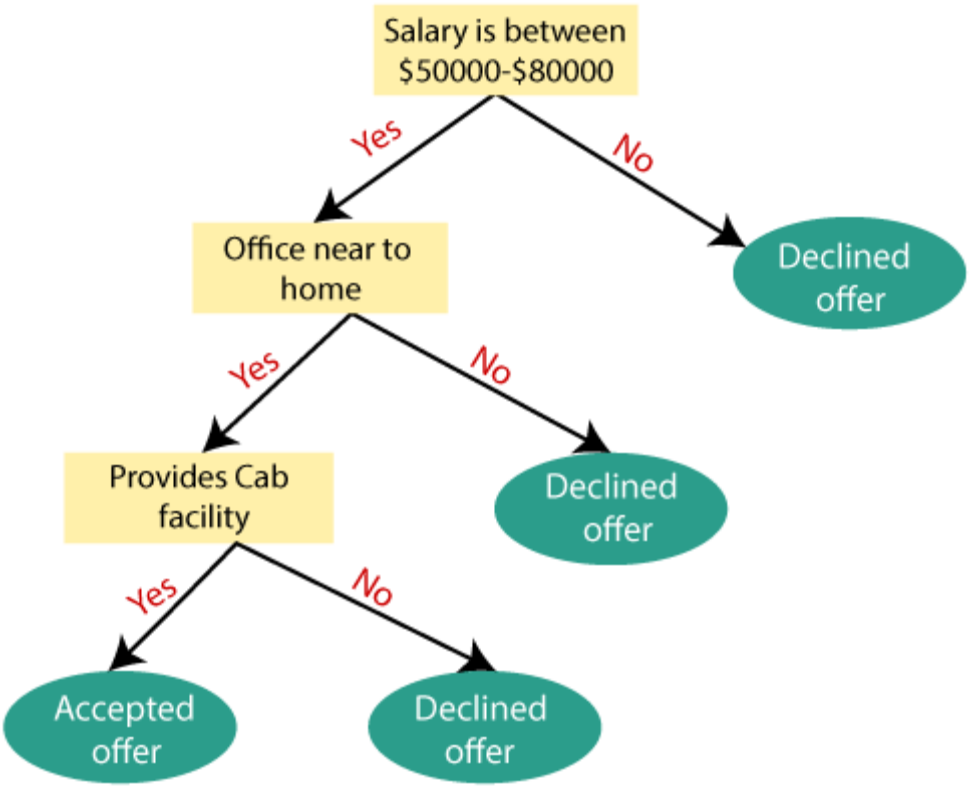


Fig 4.4.4 Decision Tree Regression example diagram.

5

Proposed approach

As it was mentioned above the problem we are trying to solve in this thesis is the energy consumption in a serverless environment in which a monolithic application needs to run. As the application needs to be divided into smaller pieces of code (functions) and each one will be assigned to a node in our kubernetes cluster, we try to distribute those functions in the most energy efficient way.

So, in this chapter, we present the proposed algorithm and the mechanisms we used for the function distribution throughout our cluster.

5.1 Profiling

Our main goal, by predicting the energy consumption of each function of our monolithic code and the time each function will need to run in every of our edge devices, is to be able to decide the proper assignment to our nodes in order to achieve the minimum energy consumption under a desired total time threshold.

The two inputs to our predictive model (as it is shown at Fig 5.1.1) will be:

1. Total amount of memory needed be every function (<*Memory*>).

2. Total time every function needs to run in our host-computer device which is a machine different than our edge devices (<*Time*>).

app_info.py		
Filename: <file_name>		
<function_name>	<Memory>	<Time>

Fig 5.1.1 Application Info.

5.1.1 1st Step – Memory profiling

For the memory profiling of our code we used Python Memory Profiler. This is a python module for monitoring memory consumption of a process as well as line-by-line analysis of memory consumption (in MB) for python programs. We also developed a tool which summarizes the total memory of a function because python memory profiler analyses the code line-by-line.

For example, as shown at Fig 5.1.1.1 our tool would output that the total memory of function <*my_func()*> is 338.867 (MB).

```
Line #    Mem usage    Increment  Occurences   Line Contents
=============================================================
     3   38.816 MiB   38.816 MiB           1   @profile
     4                                         def my_func():
     5   46.492 MiB    7.676 MiB           1       a = [1] * (10 ** 6)
     6  199.117 MiB  152.625 MiB           1       b = [2] * (2 * 10 ** 7)
     7   46.629 MiB -152.488 MiB           1       del b
     8   46.629 MiB    0.000 MiB           1       return a
```

Fig 5.1.1.1 Python Memory Profiler output example.

5.1.2 2nd Step – Time profiling

For the time profiling we used cProfile, a built-in python module that can perform time profiling and it is the most commonly used profiler currently. As it is shown at Fig 5.1.2.1 it gave us the total run time taken by the entire code, the time taken by each individual step and each individual function of our code and the number of times certain functions are being called.

```
      1961411 function calls (1931290 primitive calls) in 53.379 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    9.678    9.678 Test_Case_2.py:104(plot_grid_search_digits)
     1    0.000    0.000   14.420   14.420 Test_Case_2.py:158(plot_cv_digits)
     7    0.000    0.000    0.000    0.000 Test_Case_2.py:179(<lambda>)
     1    0.000    0.000    0.040    0.040 Test_Case_2.py:187(plot_feature_union)
     1    0.000    0.000    0.617    0.617 Test_Case_2.py:222(plot_pca_vs_lda)
     1    0.017    0.017    1.409    1.409 Test_Case_2.py:260(tree_regression_depth_200)
```

Fig 5.1.2.1 Python Memory Profiler output example.

In more detail:

- **ncalls:** Shows the number of calls made
- **tottime:** Total time taken by the given function. Note that the time made in calls to sub-functions are excluded.
- **percall:** Total time / No of calls. (remainder is left out)
- **cumtime:** Unlike *tottime*, this includes time spent in this and all subfunctions that the higher-level function calls. It is most useful and is accurate for recursive functions.
- The **percall** following **cumtime** is calculated as the quotient of cumtime divided by primitive calls. The primitive calls include all the calls that were not included through recursion.

At this point we point out that the time measurement is being collected on a machine different than the edge devices which we use in our cluster and it is briefly described in chapter 6.

5.2 Energy and Time predictions

One of the most crucial procedure in our solution is predicting the energy and time values which we will work with in order to deliver the best possible solution.

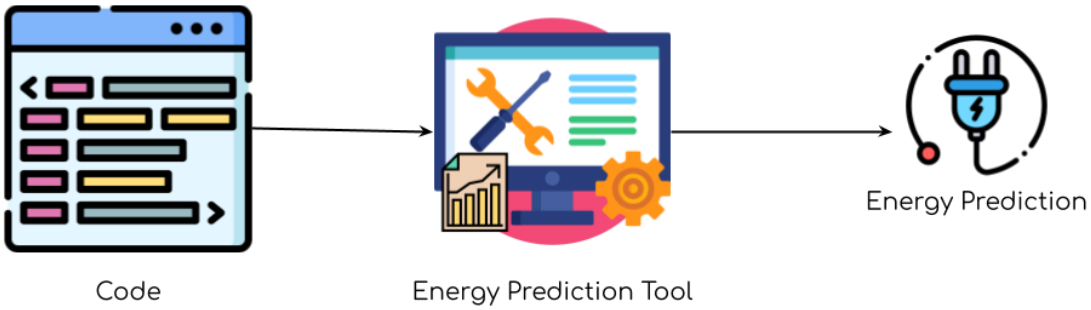


Fig 5.2 Prediction diagram.

So, after completing our profiling, we end up having a file with the details shown at Fig 5.2.1.

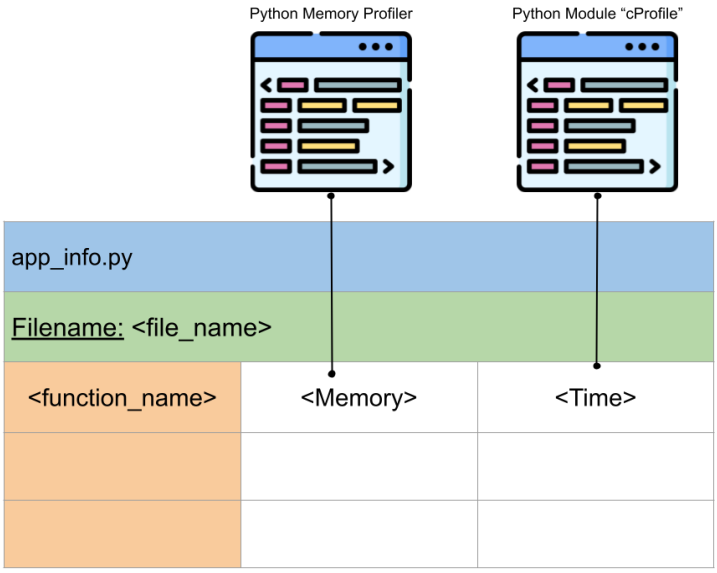


Fig 5.2.1 Final profiling representation.

The next step is to make the datasets which will be used in our prediction models. One dataset for each prediction (energy and time) and for each of our edge devices. As we will describe in the next sections we used two types of edge devices so we will need 4 datasets in total.

As machine learning techniques are a very useful and widely used technology we decided to take as samples for our measurements, examples and benchmarks from a widely used machine learning python library called “scikit-learn”. Scikit-learn is an open source machine learning library that supports supervised and unsupervised learning. It also provides various tools for model fitting, data preprocessing, model selection and evaluation, various examples and benchmarks and many other utilities.

5.2.1 Building the datasets

Our final dataset for each of the edge device follows the template below:

PyScript name *(from scikit-learn)*	Memory (MB)	Time on local machine (s)	Time on edge device (s) / Energy on edge device (Joule)
…	…	…	…
…	…	…	…

To make it more clear we describe the final status of the datasets as follows:

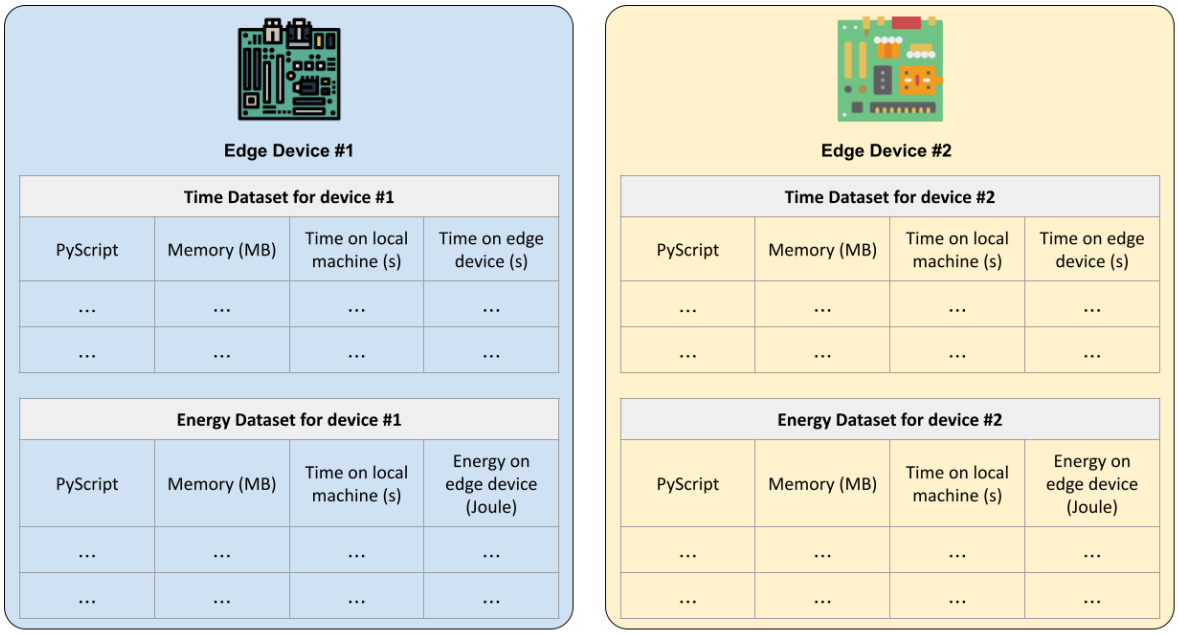


Fig 5.2.1.1 Datasets final form.

The process we followed in order to build the datasets is simple:

- First, we profiled every example and benchmark was included in scikit-learns' files (approx. 500 python scripts) just in the same way we did for profiling our sample monolithic code.
- Then, we run every profiled python script on each device and we measured the time it took to finish and its' energy consumption on that specific device. For the measurements we used the basic time library of python and for the energy we used bash files from the devices.

5.2.2 Prediction Model Decision

After building our datasets we had to decide which prediction model we will use. The best way to make the right decision is to test various models and find the one with the minimum Mean Square Error. In Statistics, Mean Square Error (MSE) is defined as Mean or Average of the square of the difference between actual and estimated values and the formula for MSE is:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \widehat{Y}_i\right)^2$$

MSE = mean squared error,

n = number of data points,

Y_i = observed values,

$\widehat{Y}_i$ = predicted values,

MSE tells us how close a regression line is to a set of points. It does this by taking the distances from the points to the regression line (these distances are the "errors") and squaring them. The squaring is necessary to remove any negative signs. It also gives more weight to larger differences. It's called the mean squared error as it finds the average of a set of errors. The lower the MSE, the better the forecast.

With that being said we are now ready to proceed to the steps we followed to find the best prediction modeling for us (*we have to point out that the same process has been followed for each edge device and each prediction, time and energy*):

Step 1: Correlation Matrix

We measure correlation of two numerical variables to find an insight about their relationships. On a dataset with many attributes, the set of correlation values between pairs of its attributes form a matrix which is called a correlation matrix. The correlation matrix is a symmetrical matrix with all diagonal elements equal to +1 where +1 describes a perfect positive correlation and 0 means no correlation.

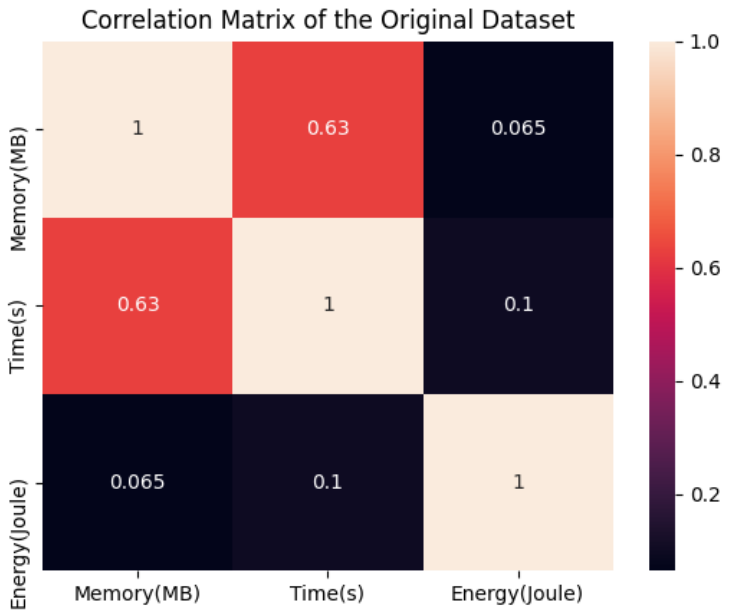
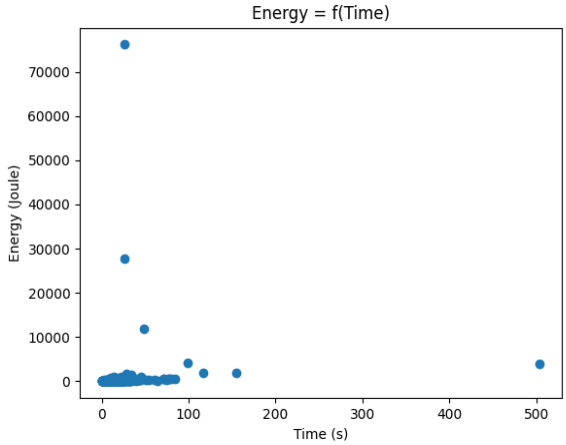
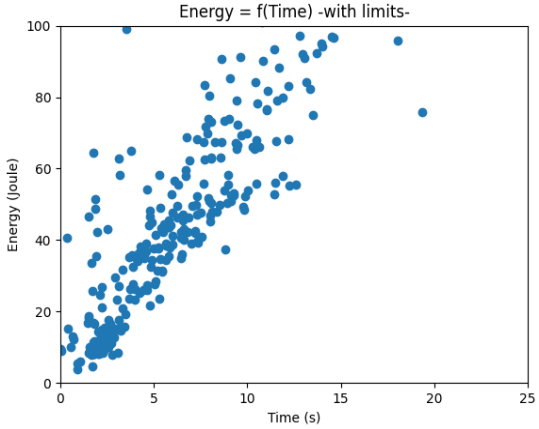
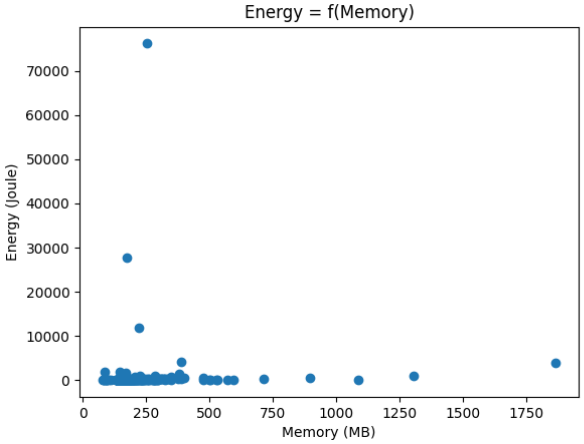


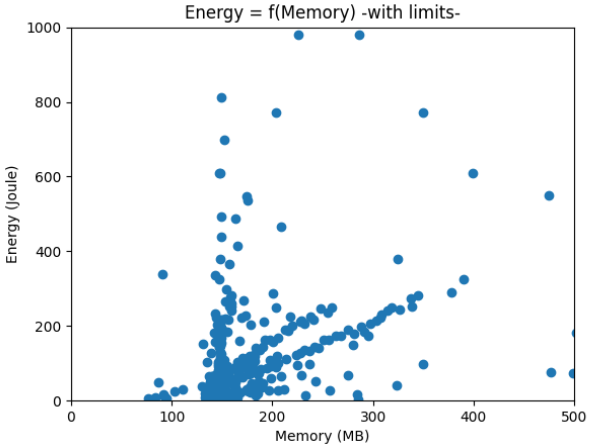
Fig 5.2.2.1 Correlation Matrix of the Original Dataset.

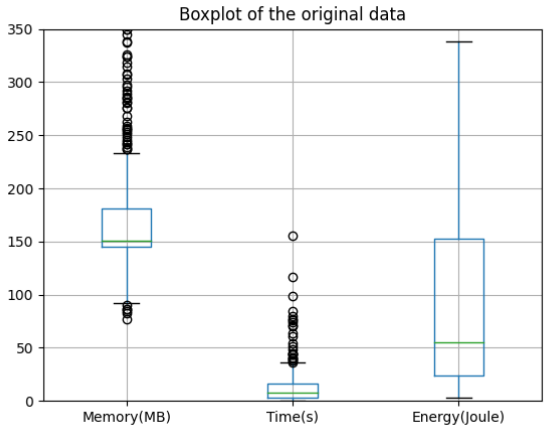
Step 2: Visualizing our dataset











Step 3: Removing the outliers

By visualizing our dataset, we can see that we have some measurements that are far greater than the majority (outliers). By definition outliers are observations that lie an abnormal distance from other values in a random sample from a population. Outliers can damage our predictions and increase the error of our model. After modifying our dataset and removing these outliers we have the new correlation matrix and the new boxplot:

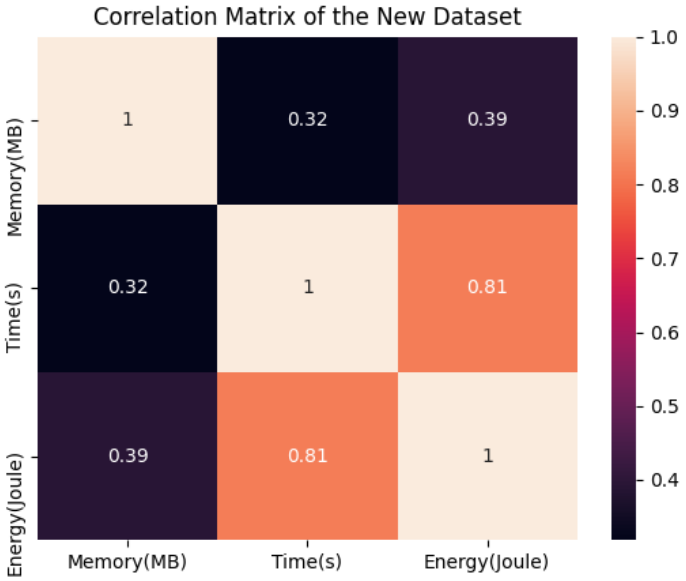


Fig 5.2.2.2 Correlation Matrix of the New Dataset.

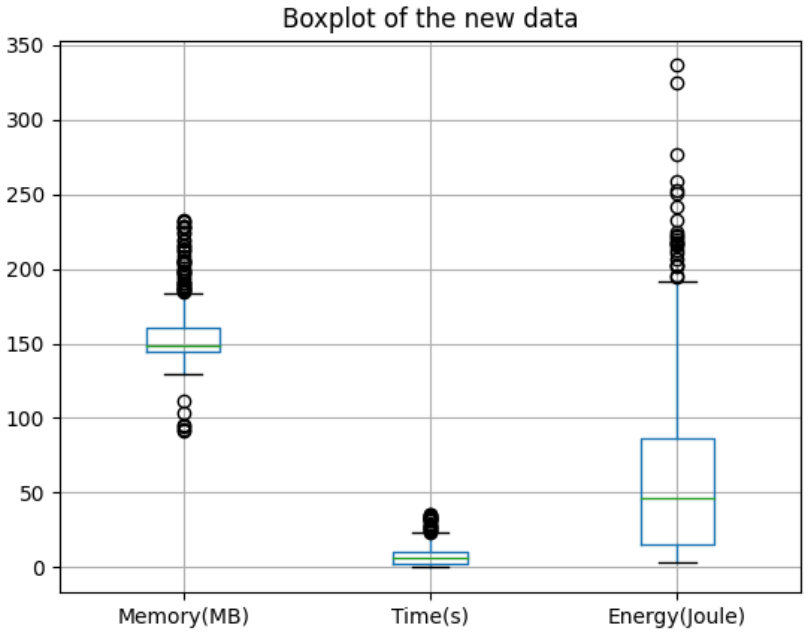


Fig 5.2.2.3 Boxplot of the New Data.

Step 4: Choose the right model

It is now clear that our dataset is in a better state, the correlation matrix seems more helpful and we can start making predictions with various prediction models and find the best one. After dividing our new dataset to test and training sets we chose to try the most widely used models and below we present the MSE of each one:

Linear Regression

(MSE = 464.0815)

Random Forest Regression

(MSE = 1101.8150)

Decision Tree Regression

(MSE = 2260.8714)

Ridge Regression

(MSE = 464.0870)

Lasso Regression

(MSE = 464.2736)

Polynomial Regression

(MSE = 509.5671)

Bayesian Ridge Regression

(MSE = 464.4987)

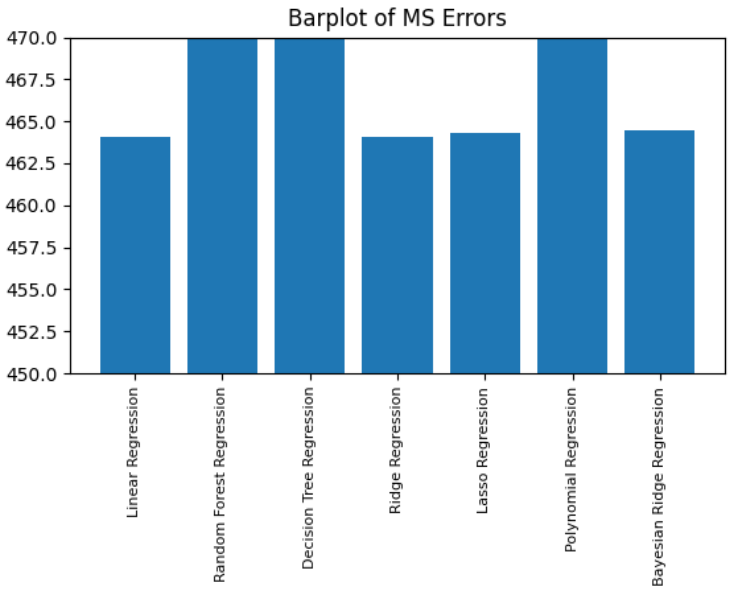


Fig 5.2.2.4 Boxplot of MSE of each model.

So for this particular prediction model we chose Linear Regression which was the model with the minimum MSE. Below we present more details regarding the accuracy of our model:

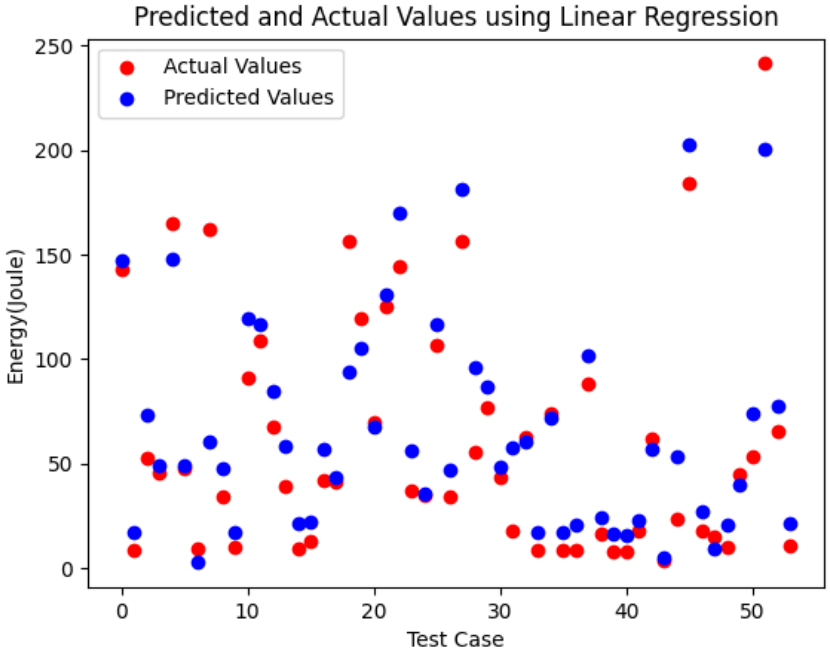


Fig 5.2.2.5 Predicted and actual values using Linear Regression.

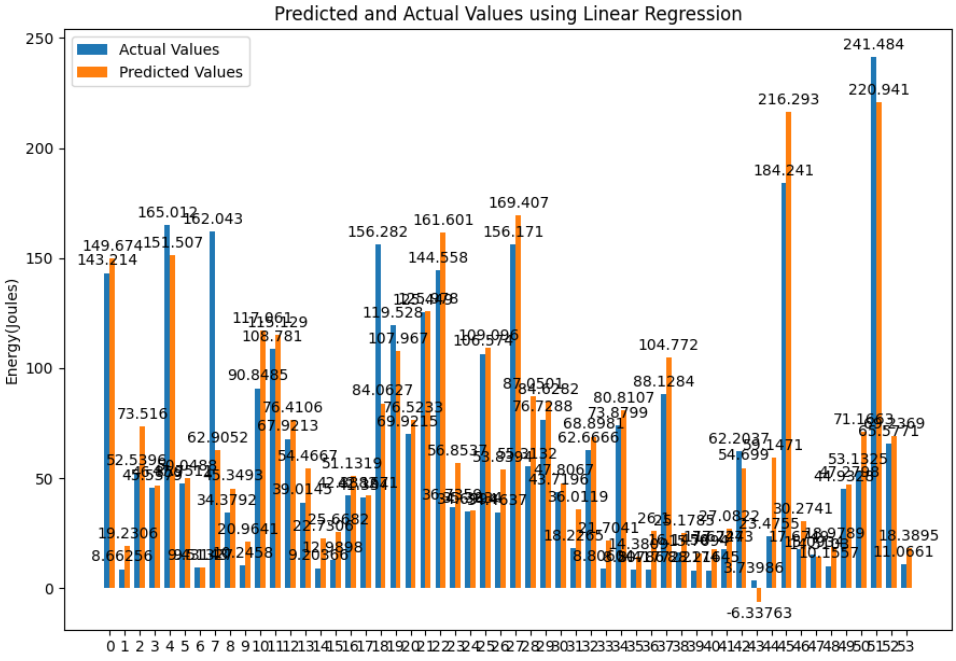


Fig 5.2.2.6 Predicted and actual values using Linear Regression.

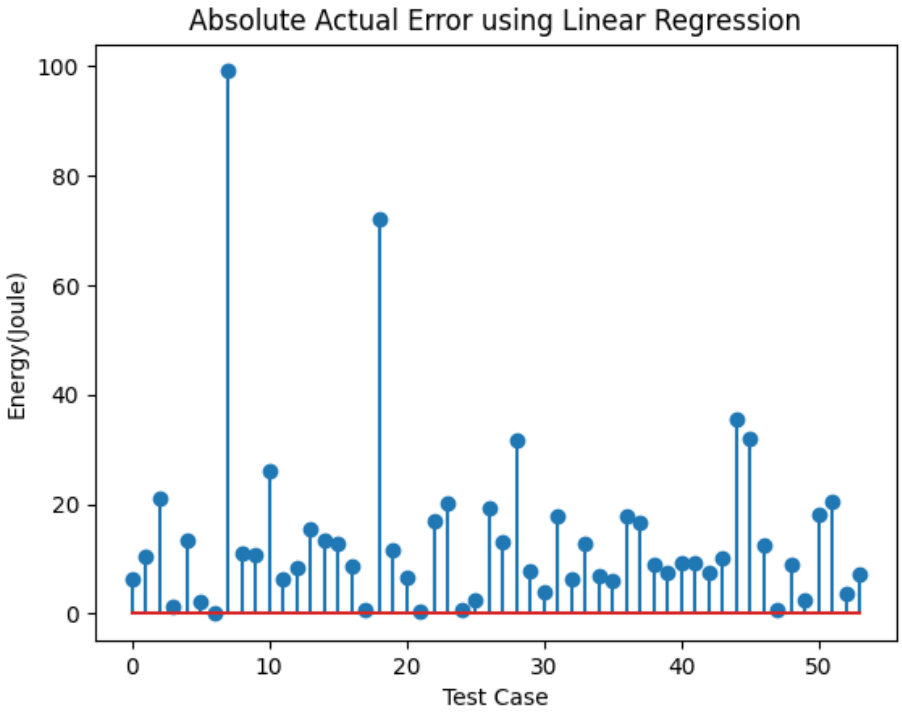


Fig 5.2.2.7 Absolut actual error using Linear Regression.

So we end up having 4 files that include our time and energy predictions for each node as the Fig 5.2.2.8 below describes:

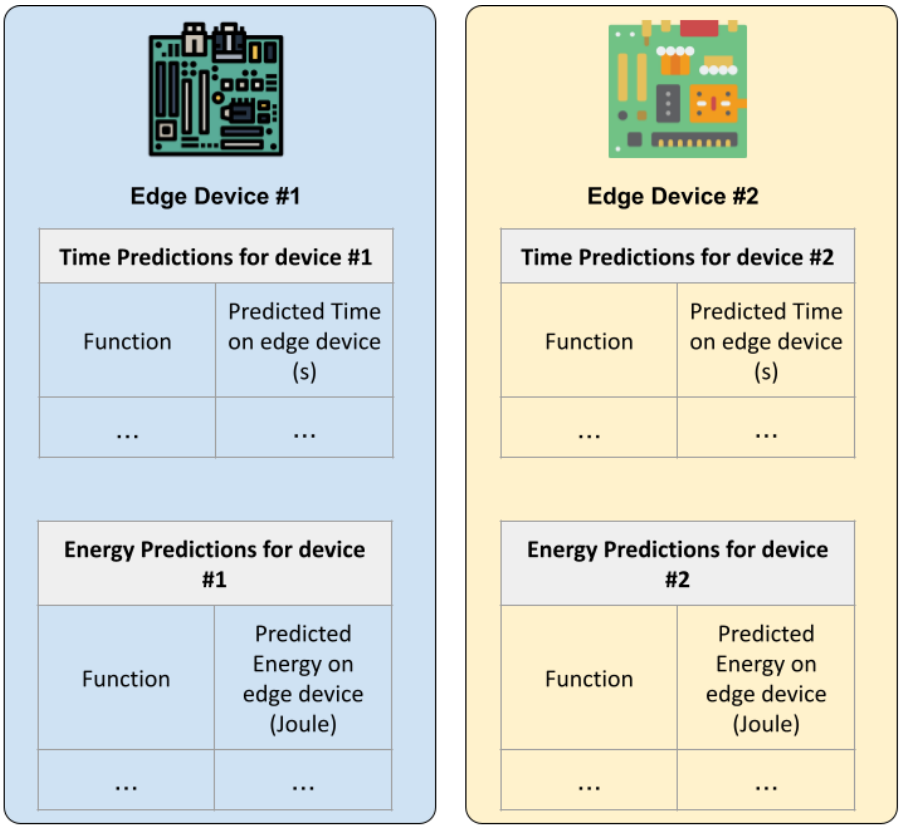


Fig 5.2.2.8 Prediction files description.

5.3 Placing algorithm

In this chapter we will describe the algorithm responsible for placing and distributing the functions through our cluster while minimizing the predicted total energy consumption under a specific time threshold.

5.3.1 Knapsack and Multiple knapsack problem

At first we recognize our problem as a multiple knapsack problem. The knapsack problem is an optimization problem used to illustrate both problem and solution. It derives its name from a scenario where one is constrained in the number of items that can be placed inside a fixed-size knapsack. Given a set of items with specific weights and values, the aim is to get as much value into the knapsack as possible given the weight constraint of the knapsack. A

simple and clear example could be the problem of placing different kind of objects in a knapsack as it is shown at Fig 5.3.1.1 while we try maximize the total value of the knapsack which has limited carrying storage.

Fig 5.3.1.1 Example of knapsack problem.

The knapsack problem (KP) is an example of a combinational optimization problem, a topic in mathematics and computer science about finding the optimal object among a set of objects. It is a commonly used example problem in combinatorial optimization, where there is a need for an optimal object or finite solution where an exhaustive search is not possible. In the knapsack problem, the given items have two attributes at minimum – an item's value, which affects its importance, and an item's weight or volume, which is its limitation aspect. Since an exhaustive search is not possible, one can break the problems into smaller sub-problems and run it recursively. This is called an optimal sub-structure. This deals with only one item at a time and the current weight still available in the knapsack. The problem solver only needs to decide whether to take the item or not (binary KP) based on the weight that can still be accepted. However, if it is a program, re-computation is not independent and would cause problems. This is where dynamic programming techniques can be applied. Solutions to each sub-problem are stored so that the computation would only need to happen once.

The multiple knapsack problem (or MKP) is an NP-hard extension to the standard binary knapsack selection problem. The goal is the same; to find a subset of items that maximizes the total profit/gain (objective function), however, the difference is that instead of having a single knapsack or resource, there are multiple knapsacks/resources (each is a separate constraint) and the subset of items should not violate the capacity of any of these knapsacks.

5.3.2 Problem Definition

Now that we have made clear how our problem can be approached we can try to define it:

- **m** = Knapsacks (no. of Nodes)
- **n** = Items to be inserted in the knapsacks (no. of functions)
- $\mathbf{w_{ij}}$ = Weight (Time prediction on edge device (i) for function (j))
- $\mathbf{d_i}$ = Cost / Value (Energy on a node (i) by the allocation of the functions)
- $\mathbf{x_{ij}}$ = 1 if the function j has been allocated to node i
- $\mathbf{x_{ij}}$ = 0 if the function j has not been allocated to node i
- Each function can be allocated by only one node
- **T** = Time threshold (Input by the user)

Equation:

$$\text{Minimize } z = \sum_{i=1}^{m} d_i$$

Subject to:

- $x_{ij} = \begin{cases} 1 \\ 0 \end{cases}$,i∈M = {1, …, m}, j∈N = {1, …, n}
  - $\sum_{i=1}^{m} x_{ij} = 1$, j∈N = {1, …, n}
  - $\sum_{i=1}^{m} w_{ij} \leq T$, j∈N = {1, …, n}

5.3.3 Algorithm

The approach we took for the minimization problem we are facing is basically by trying a big number of possible solutions and give an approximate solution. The algorithm creates a pool

of possible solutions and choses the solution which satisfies the constraints given by the user while minimizing the total energy consumption.

5.3.4 Algorithm Implementation

Now that we have stated the problem we are trying to solve and the theoretical background which will help us achieve it lets dive into our algorithm. Our algorithm is developed in python programming language and makes use of all of the tools we described above (profiling and predictions). The process follows the steps below:

Step 1: The user inputs the time threshold of his/her choice.

Step 2: Get the cluster details. This includes number and type of edge devices in our environment which are also imported from the user.

Step 3: The user inputs if the algorithm should process the data according to parallel or serial run in the serverless environment. We decided to give the option to run the code in serial or in parallel (as long as there are no dependencies between the functions) in order to see the difference in our results.

Step 4: This is the step where data about our monolithic code are imported. The data from profiling (e.g., number of functions) and also the predictions we described above.

Step 5: At this point the algorithm generates map tables which describe the distribution of the serverless applications (functions of the monolithic code) and runs a set of tests to find the binary map table which gives us the distribution with the minimum predicted energy consumption under the user's time threshold. A binary map table can be visualized as it is shown at Fig 5.3.4.2.

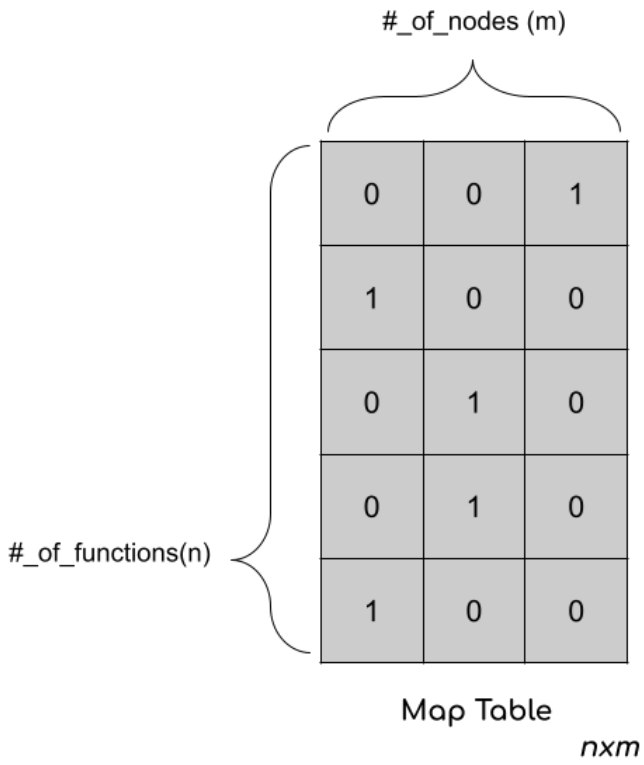


Fig 5.3.4.2 Example of a binary map table with 3 nodes and 5 functions to be placed.

The explanation is simple:

- function_0 will be executed at node_2
- function_1 will be executed at node_0
- function_2 will be executed at node_1
- function_3 will be executed at node_1
- function_4 will be executed at node_0

After creating the map tables, the process follows the simple process of table multiplication for every device and every variable (energy and time), as it is described in Fig 5.3.4.3, in order to calculate the total energy consumption and the total run-time:

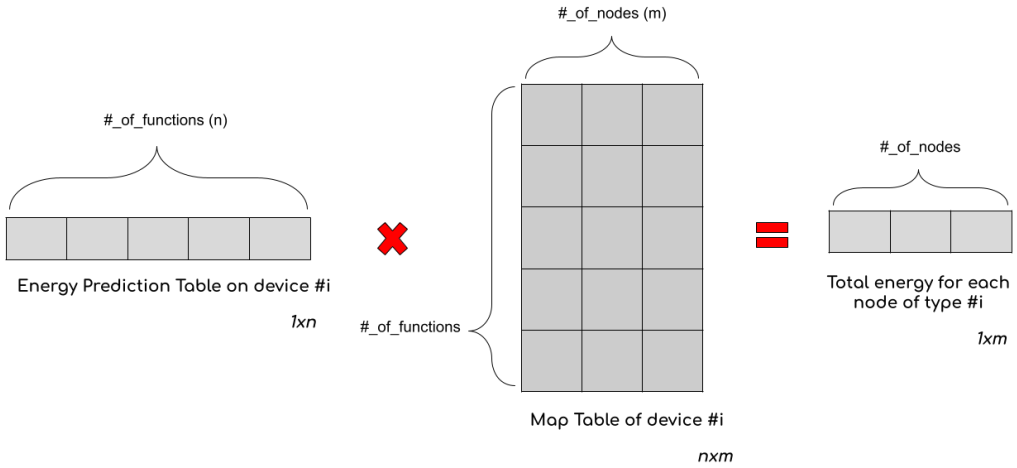


Fig 5.3.4.3 Process of producing the final map table with 3 nodes and 5 functions to be placed.

Every prediction (energy or time) is stored in a row table. The first element is the prediction of function_0, the second element of the row table is the prediction for the function_1 etc. Then the predictions row table is multiplied with the generated map table and the result is a row table which gives us the total energy consumption or run-time on each node. The first element of the last table is the total energy consumption of node_0, the second element is the total energy consumption of node_1 etc.

Step 6: This is the last stage where the algorithm calculates the total energy consumption in our cluster (by adding every element of the produced table above) and makes sure that the total run-time is under the users' desired time threshold. If it is under the time limit then it compares this distribution with the previous one and saves the one with the minimum total energy consumption. What we get as a result is the best map table accompanied by the predicted total energy consumption and the predicted total run-time.

6

Experimental Evaluation

In this chapter, we are going to describe the Kubernetes cluster created for our experiments and we will also present the tools we used to test and evaluate our algorithm.

6.1 Experimental Setup

Our cluster (or serverless environment) consists of four nodes, three Edge-Worker nodes and one Master node as shown in Fig 6.1.

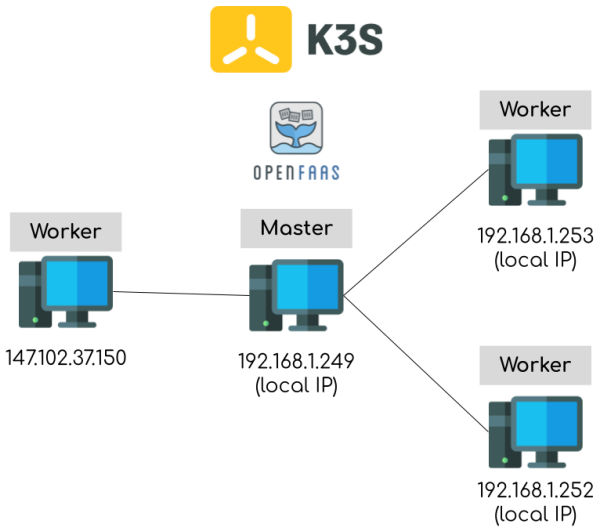


Fig 6.1 Experimental Setup

For the 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. For our Edge-Worker nodes we used one NVIDIA Jetson Nano and two NVIDIA Jetson Xavier NX and they both use Aarch64, 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 Docker and OpenFaaS on top of it. Most of the components, as well as the devices, will be described below.

So, we created a heterogeneous and scalable cluster which can be found in real life situations. The main idea is that a developer would like to run his cloud-native application, written in the form of a monolithic code, in a serverless environment while keeping the total energy consumption of the application in minimum levels. This can easily scale up or down depending on the application/s and it could consider to be a solid scenario in an industrial environment situation.

6.1.1 Edge Devices

ARM is the acronym for Advanced RISC Machines where RISC is the acronym for Reduced Instruction Set Computing. It is the most pervasive processor architecture in the world, with billions of Arm-based devices like sensors, wearables and smartphones, supercomputers etc. being used in our everyday lives. 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.

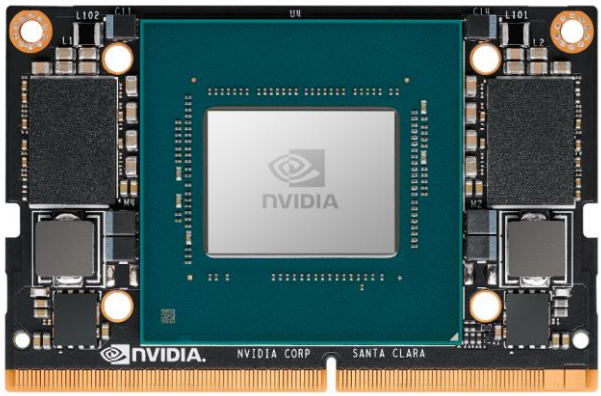
6.1.1.1 NVIDIA Jetson Nano

Jetson Nano is an embedded Linux module featuring the GPU accelerated processor NVIDIA Tegra targeted at edge AI applications. It is a full blown single-board-computer in the form of a module. 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 specifications are shown in Fig 6.1.1.1:

GPU	NVIDIA Maxwell architecture with 128 NVIDIA CUDA® cores
CPU	Quad-core ARM Cortex-A57 MPCore processor
Memory	4 GB 64-bit LPDDR4, 1600MHz 25.6 GB/s
Storage	16 GB eMMC 5.1
Video Encode	250MP/sec 1x 4K @ 30 (HEVC) 2x 1080p @ 60 (HEVC) 4x 1080p @ 30 (HEVC) 4x 720p @ 60 (HEVC) 9x 720p @ 30 (HEVC)
Video Decode	500MP/sec 1x 4K @ 60 (HEVC) 2x 4K @ 30 (HEVC) 4x 1080p @ 60 (HEVC) 8x 1080p @ 30 (HEVC) 9x 720p @ 60 (HEVC)
Camera	12 lanes (3x4 or 4x2) MIPI CSI-2 D-PHY 1.1 (1.5 Gb/s per pair)
Connectivity	Gigabit Ethernet, M.2 Key E
Display	HDMI 2.0 and eDP 1.4
USB	4x USB 3.0, USB 2.0 Micro-B
Others	GPIO, I^2C, I^2S, SPI, UART
Mechanical	69.6 mm x 45 mm 260-pin edge connector

Fig 6.1.1.1 NVIDIA Jetson Nano Specifications

6.1.1.2 NVIDIA Jetson Xavier NX



Jetson Xavier NX is ideal for use in high-performance AI systems like commercial robots, medical instruments, smart cameras, high-resolution sensors, automated optical inspection,

smart factories, and other AIoT embedded systems. It supports multiple power modes, including low-power modes for battery-operated systems and it comes with cloud-native support. Furthermore, Jetson Xavier NX accelerates the NVIDIA software stack with more than 10X the performance of its widely adopted predecessor, Jetson TX2. The specifications are shown in Fig 6.1.1.2:

AI Performance	21 TOPS (INT8)
GPU	384-core NVIDIA Volta™ GPU with 48 Tensor Cores
GPU Max Freq	1100 MHz
CPU	6-core NVIDIA Carmel ARM®v8.2 64-bit CPU 6MB L2 + 4MB L3
CPU Max Freq	2-core @ 1900MHz 4/6-core @ 1400Mhz
Memory	8 GB 128-bit LPDDR4x @ 1866MHz 59.7GB/s
Storage	16 GB eMMC 5.1
Power	10W|15W|20W
PCIe	1 x1 + 1x4 (PCIe Gen3, Root Port & Endpoint)
CSI Camera	Up to 6 cameras (36 via virtual channels) 12 lanes MIPI CSI-2 D-PHY 1.2 (up to 30 Gbps)
Video Encode	2x 4K60 | 4x 4K30 | 10x 1080p60 | 22x 1080p30 (H.265) 2x 4K60 | 4x 4K30 | 10x 1080p60 | 20x 108p30 (H.264)
Video Decode	2x 8K30 | 6x 4K60 | 12x 4K30 | 22x 1080p60 | 44x 1080p30 (H.265) 2x 4K60 | 6x 4K30 | 10x 1080p60 | 22x 1080p30 (H.264)
Display	2 multi-mode DP 1.4/eDP 1.4/HDMI 2.0
DL Accelerator	2x NVDLA Engines
Vision Accelerator	7-Way VLIW Vision Processor
Networking	10/100/1000 BASE-T Ethernet
Mechanical	45 mm x 69.6 mm 260-pin SO-DIMM connector

Fig 6.1.1.2 NVIDIA Jetson Xavier NX Specifications

6.1.2 Evaluated Application

At this stage we present the application with which we will evaluate our algorithm and its proposed solution. We chose to evaluate a monolithic application which consists of multiple Machine Learning functions as our tool is developed and our cluster has been built in such a way that proposes solutions on these heavily used technologies. More specific our test applications consist of functions that create random datasets and try to predict the next

values using different prediction models. The structure of the application we chose to evaluate our algorithm is shown at Fig 6.1.2.1.

Fig 6.1.2.1 Evaluated Application structure

In order for our application to be cloud-native we had to build docker images which we will run as functions (or serverless applications) in our OpenFaaS environment. We had to build two different images as the tools to measure energy consumption and run-time in every edge device are different.

With that being said, as you can see at Fig 6.1.2.2, we have two individual images each one consisting of all of the functions of our test-application (monolithic code) shown above, all the tools needed to capture the measurements (energy consumption and time) and the Dockerfile which has all the information about building each image.

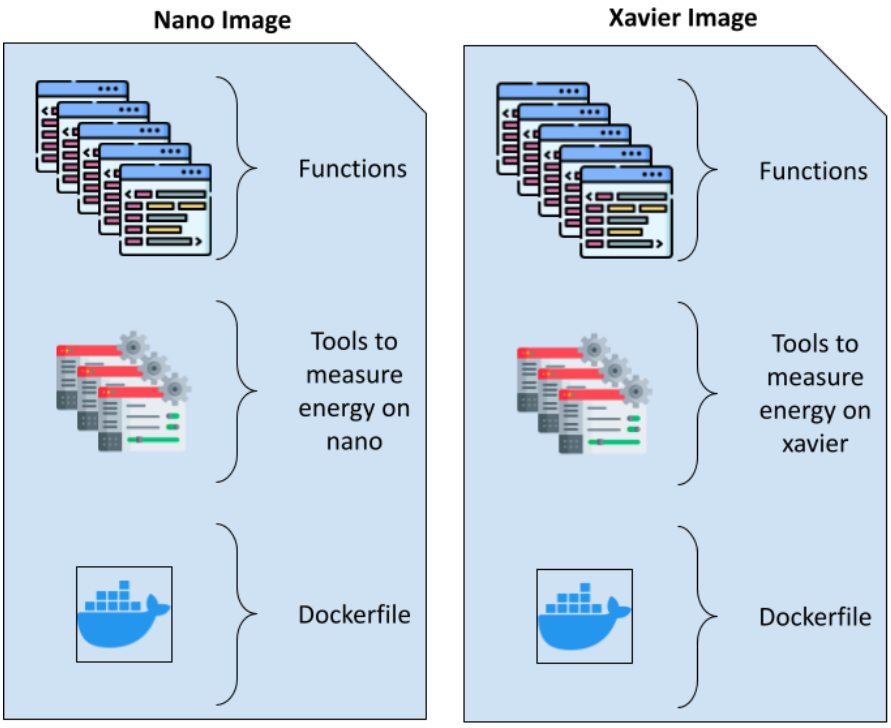


Fig 6.1.2.2 Images used for evaluation

6.2 Evaluation

Now we use our experimental infrastructure described above to evaluate our custom placement decision algorithm. We used a machine learning application which consists of 25 individual functions. The algorithm proposes a placement accordingly to our cluster as it is described above while predicting the energy consumption of the proposal as well as the total run-time of the whole application.

6.2.1 Experimental Procedure

In order to evaluate our algorithm, we used an application which consists of 25 machine learning functions which are inspired by the scikit-learn python library. This monolithic application with its individual functions is to be splitted into 25 independent serverless functions and run in our OpenFaaS environment in the most energy efficient way.

Also our algorithm makes a proposal depending on the accepted by-the-user total run-time so it is interesting to test the behavior of our proposals when the users tries to decrease the total run-time and at the end, find the minimum run-time that the application can run in this specific environment.

With that being said, the procedure we followed is made simple. First, we run the algorithm and we got the first proposal. Then we kept decreasing the total run-time until we reached the proposal with the minimum run-time. The algorithm gave us 9 proposals, each one with less time than the previous one. The results we have been given from the algorithm are what we have been expecting.

As it is shown in Fig 6.2.1 our algorithm first proposed a solution where all of the placement was taking place at the most energy efficient and “fast” devices (devices 2 and 3 - the xavier ones). But when we needed to decrease the total run-time, the algorithm, in order to achieve parallelization, as a second proposal made used of the “slower” and not that energy efficient device (device 1 - nano) to run one of the functions. The results were to decrease the time as needed but increase the total energy consumption. The same behavior has been observed for the next proposals until we reach a time limit and the minimum run-time that could be achieved.

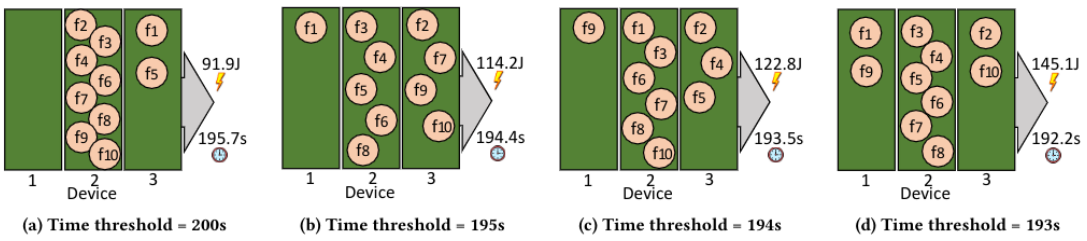


Fig 6.2.1 Placement behavior while decreasing time (example)

At this time, we were ready to test each one of the proposals in our real-life environment. We used our custom docker images described above to run every function in the specific node given by the map table of the proposal and capture its actual energy consumption and run-time. At the end we could easily sum up the total energy consumption and run-time of each proposal and compare those numbers with the predicted numbers given by the algorithm.

It was interesting to compare our solution with the original kubernetes placement without selecting nodes for each of the function to run on. So using our custom docker images we run all of the serverless functions without telling kubernetes what to do and where to place each one of them. We simply run every function and captured its energy consumption and run-time.

6.2.2 Results and Kubernetes Comparison

In this section, we present the two charts of captured total energy consumption and total run-time of each proposal while comparing those results with the prediction given by the algorithm and the placement of kubernetes.

While running our experiments we noticed that the theoretical-predicted values of both energy and time had some standard difference from the actual values which was measured with our tools while running the functions. Using some simple methods, we have been able to calculate that standard overhead which is consider to be an overhead caused by the kubernetes deployment and the OpenFaaS. More specific, we noticed that there is a type of noise on the measurements coming from the device itself and it seems to be increased due to the kubernetes deployment and the way OpenFaaS communicates with the master node.

With that being said, in order to be more precise on the evaluation charts, instead of the predicted values given by the algorithm we used the overhead model which is nothing more than the predicted values with that overhead added on them.

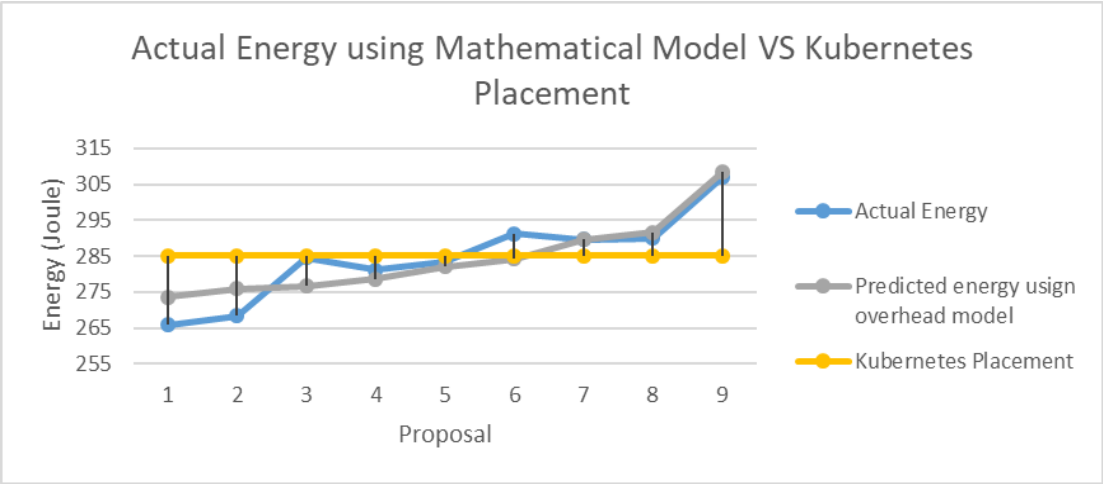


Fig 6.2.2.1 Energy Evaluation chart

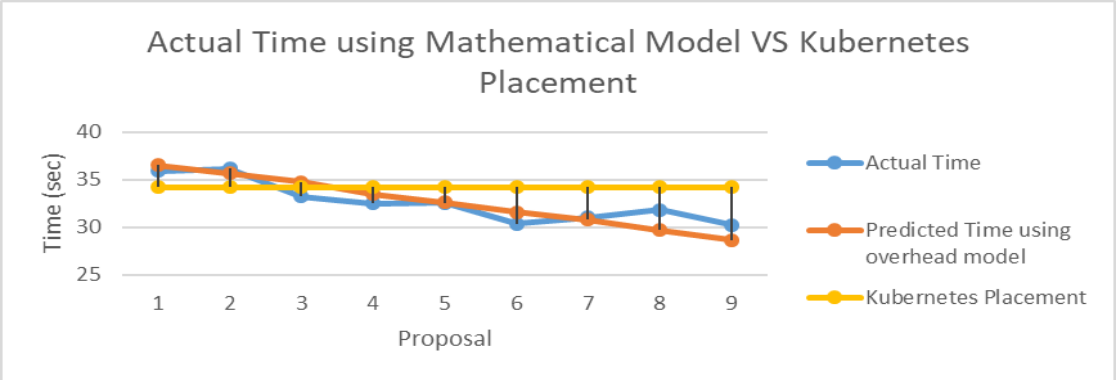


Fig 6.2.2.2 Time Evaluation chart

At Fig 6.2.2.1 and Fig 6.2.2.1 we first notice that there is a small error between the predicted and actual values, which is quite expected as there is always a room for improvement when it comes to predictive models, but both lines in both charts follow the same trend and behavior. Also we can see that as we keep decreasing the desired run-time the energy consumption is increasing, as it was also expected.

When it comes to evaluate our algorithm with the kubernetes placement it is important to point out that if we set a time threshold that is bigger than the kubernetes-placement-time we accomplish less energy consumption. Also using our tool, we can set a time threshold lower than the kubernetes placement, which will give our user more control on the workload, workflow and on his overall experience when using OpenFaaS and serverless functions.

In a deeper statistical analysis, we achieved approx. up to 6% less energy consumption comparing our solution with the default kubernetes placement (if the desired time threshold is above the kubernetes placement total run-time, as it is in proposal 1) and approx. up to 11% speedup when our user wants to run the application in a faster way, other than the kubernetes default placement. All in all, combining the energy and time efficiency and results, we managed to achieve an approx. 2.6% better placement than the default kubernetes placement.

7

Conclusion and Future Work

7.1 Summary

In this thesis, we design and evaluate a decision placement algorithm for monolithic applications consisting of multiple functions that need to run on a serverless environment while ensuring the minimum total energy consumption and a desired run-time threshold. We used state of the art tools such as Kubernetes and OpenFaaS while profiling the application using python memory profiler and python time profiler. It is very important to mention that our tool is scalable and can support heterogeneous machines. Furthermore, we evaluated the tool, against the Kubernetes default placement. As shown, in our experiments, the tool improves the quality of service and achieves this while giving the user more control on his serverless deployment.

7.2 Future Work

This particular subject is very promising due to its versatility and its necessity. The fact that, with an optimal placement decision 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 algorithm can be evaluated in more heterogeneous devices and applications, so that we can explore the scalability of our approach. Another improvement might be taking into consideration the devices available resources before deciding if a function with standard needs can be deployed on the specific device.

Furthermore, there can be an even more research on predictive models and techniques in order to improve the energy and time predictions of the algorithm. We could investigate several ML models or use additional, more sophisticated features for our models. This way, the model should be more effective and train specifically for each different case.

Another idea is to integrate into the decision algorithm, connection related aspects. Thus, making the algorithm more appropriate for clusters consisted of devices in different locations. Moreover, it is also interesting to investigate applications that consists of connected functions, functions that communicate with one another and see the cost of that communications as an overhead on our total predictions and actual values on the measurements.

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.

8

Βιβλιογραφία / References

[1] Rob van der Meulen. What Edge Computing Means for Infrastructure and Operations Leaders (October 03, 2018) Available: https://www.gartner.com/smarterwithgartner/what-edge-computing-means-for-infrastructure-and-operations-leaders

[2] https://www.onlogic.com/company/io-hub/edge-computing-vs-fog-computing/#:~:text=In%20a%20nutshell%2C%20edge%20computing,purposes%2C%20such%20as%20data%20filtering.

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

[4] https://openwhisk.apache.org/documentation.html

[5] Cheol-Ho Hong and Blesson Varghese. 2019. Resource Management in Fog/Edge Computing: A Survey on Architectures, Infrastructure, and Algorithms. ACMComput. Surv. 52, 5, Article 97 (September 2019)

[6] Charalampos Marantos, Konstantinos Salapas, Lazaros Papadopoulos, Dimitrios Soudris. A Flexible Tool for Estimating Applications Performance and Energy Consumption Through Static Analysis

[7] Josef Spillner and Serhii Dorodko. Java Code Analysis and Transformation into AWS Lambda Functions (February 21, 2017)

[8] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, Philippe Suter. Serverless Computing: Current Trends and Open Problems (10 Jun 2017)

[9] Babak Bashari Rad, Harrison John Bhatti, Mohammad Ahmadi. An Introduction to Docker and Analysis of its Performance (March 2017)

[10] Ricardo Stegh Camati, Alcides Calsavara, Luiz Lima Jr. Solving the Virtual Machine Placement Problem as a Multiple Multidimensional Knapsack Problem (2014).

[11] Silvio Roberto Martins Amarante, André Ribeiro Cardoso, Filipe Maciel Roberto, Joaquim Celestino Jr. Using the multiple knapsack problem to model the problem of virtual machine allocation in cloud computing (2013).

[12] Achilleas Tzenetopoulos, Evangelos Apostolakis, Aphrodite Tzomaka, Christos Papakostopoulos, Konstantinos Stavrakakis, Manolis Katsaragakis, Ioannis Oroutzoglou, Dimosthenis Masouros, Sotirios Xydis, Dimitrios Soudris. FaaS and Curious: Performance Implications of Serverless Functions on Edge Computing Platforms (13 November 2021)

[13] Achilleas Tzenetopoulos, Charalampos Marantos, Giannos Gavrielides, Sotirios Xydis, Dimitrios Soudris. FADE: FaaS-inspired application decomposition and Energy-aware function placement on the Edge (13 November 2021)

[14] https://morioh.com/p/47719a08c1e8

[15] https://kubernetes.io/docs/concepts/overview/components/

[16] https://www.aquasec.com/cloud-native-academy/kubernetes-101/kubernetes-architecture/

[17] https://www.quora.com/What-is-more-secure-Kubernetes-serverless-or-microservices

[18] https://ericstoekl.github.io/faas/architecture/

[19] https://www.cncf.io/blog/2020/04/13/serverless-open-source-frameworks-openfaas-knative-more/

[20] https://www.cncf.io/blog/2020/04/13/serverless-open-source-frameworks-openfaas-knative-more/