Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογίας Πληροφορικής και Υπολογιστών

Heterogeneity-Aware Serverless Workflow Scheduling

Μελέτη και υλοποίηση

Διπλωματικη Εργασια

του

ΓΙΑΓΚΟΥ Ι. ΔΗΜΗΤΡΙΟΥ

Επιβλέπων: Δημήτριος Ι. Σούντρης
Καθηγητής ΕΜΠ

Αθήνα, 2022

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογίας Πληροφορικής και Υπολογιστών

Heterogeneity-Aware Serverless Workflow Scheduling

Μελέτη και υλοποίηση

Διπλωματικη Εργασια

του

ΓΙΑΓΚΟΥ Ι. ΔΗΜΗΤΡΙΟΥ

Επιβλέπων: Δημήτριος Ι. Σούντρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 14η Νοεμβρίου 2022.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Δημήτριος Ι. Σούντρης	Παναγιώτης Τσανάκας	Σωτήριος Ξυδής
Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ	Επίκ. Καθηγητής ΧΠ

Αθήνα, 2022

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογίας Πληροφορικής και Υπολογιστών

ΔΗΛΩΣΗ ΜΗ ΛΟΓΟΚΛΟΠΗΣ ΚΑΙ ΑΝΑΛΗΨΗΣ ΠΡΟΣΩΠΙΚΗΣ ΕΥΘΥΝΗΣ

Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ενυπογράφως ότι είμαι αποκλειστικός συγγραφέας της παρούσας Πτυχιακής Εργασίας, για την ολοκλήρωση της οποίας κάθε βοήθεια είναι πλήρως αναγνωρισμένη και αναφέρεται λεπτομερώς στην εργασία αυτή. Έχω αναφέρει πλήρως και με σαφείς αναφορές, όλες τις πηγές χρήσης δεδομένων, απόψεων, θέσεων και προτάσεων, ιδεών και λεκτικών αναφορών, είτε κατά κυριολεξία είτε βάσει επιστημονικής παράφρασης. Αναλαμβάνω την προσωπική και ατομική ευθύνη ότι σε περίπτωση αποτυχίας στην υλοποίηση των ανωτέρω δηλωθέντων στοιχείων, είμαι υπόλογος έναντι λογοκλοπής, γεγονός που σημαίνει αποτυχία στην Πτυχιακή μου Εργασία και κατά συνέπεια αποτυχία απόκτησης του Τίτλου Σπουδών, πέραν των λοιπών συνεπειών του νόμου περί πνευματικών δικαιωμάτων. Δηλώνω, συνεπώς, ότι αυτή η Πτυχιακή Εργασία προετοιμάστηκε και ολοκληρώθηκε από εμένα προσωπικά και αποκλειστικά και ότι, αναλαμβάνω πλήρως όλες τις συνέπειες του νόμου στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής άλλης πνευματικής ιδιοκτησίας.

(Υπογραφή)

.....
Δημήτριος Ι. Γιάγκος

14 Νοεμβρίου 2022

Περίληψη

Η δημοτικότητα της εκτέλεσης υπολογιστικών φορτίων στο νέφος έχει πλέον εκτοξευθεί. Προηγουμένως, την εκτέλεση αυτή αναλάμβαναν ιδιωτικοί τοπικοί εξυπηρετητές (servers) ενώ τα τελευταία χρόνια γι' αυτό το φορτίο είναι υπεύθυνα, κατά την μερίδα του λεόντος, σύγχρονα και δημόσιας πρόσβασης περιβάλλοντα στο νέφος. Η αλλαγή αυτή έδωσε την δυνατότητα στους προγραμματιστές να απαλλαχθούν από την επίπονη διαδικασία διαχείρισης και ενορχήστρωσης των εξυπηρετητών και των υποκείμενων μηχανημάτων, μία ευθύνη την οποία ανέλαβαν οι πάροχοι υπολογιστικών υπηρεσιών του νέφους (cloud providers). Οι πάροχοι, μάλιστα, έχουν προχωρήσει στην ανάπτυξη υπηρεσιών των οποίων η κεντρική ιδέα στρέφεται γύρω από την πλήρη απαγκίστρωση των προγραμματιστών από τους εξυπηρετητές. Αυτό είχε ως αποτέλεσμα να γεννηθεί ο όρος *Serverless computing*, ο οποίος ετυμολογικά υπονοεί την απουσία των εξυπηρετητών από το πεδίο δράσης των προγραμματιστών.

Το Function-as-a-Service (FaaS) είναι ένα serverless υπολογιστικό μοντέλο, το οποίο επιτρέπει στους προγραμματιστές να ανεβάσουν και να εκτελέσουν στο νέφος μικρά λειτουργικά κομμάτια κώδικα, τα οποία έχουν την μορφή συναρτήσεων. Παρ' όλα αυτά, η εγγύηση για γρήγορη και πλήρη κάλυψη των πιθανών απαιτήσεων του χρήστη απέχει λίγο από το παρόν διότι μερικές φορές η εκτέλεση εφαρμογών υπό αυτό το μοντέλο μπορεί να αποβεί προβληματική. Όμως, πρόσφατα έχει λάβει χώρα αρκετή έρευνα γύρω από το συγκεκριμένο θέμα γιατί φαίνεται ιδιαίτερα υποσχόμενο, σε πολλές περιπτώσεις χρήσης, να απαλλαχθεί ο χρήστης από την ευθύνη διαχείρισης και συντήρησης του υποκείμενου εξοπλισμού ενώ παράλληλα να δημιούργονται καταλληλότερες συνθήκες για αποδοτικότερη και οικονομικότερη χρησιμοποίηση των πόρων (resource utilization).

Η συγκεκριμένη διπλωματική εργασία ερευνά τους παράγοντες που επηρεάζουν την εκτέλεση ενός υπολογιστικού serverless φορτίου στο νέφος και προτείνει μία δυναμική λύση στην δρομολόγηση serverless εφαρμογής αναπτύσσοντας την με την βοήθεια Βαθιάς Ενισχυτικής Μάθησης. Στόχος είναι η βέλτιστη εξυπηρέτηση των αιτημάτων των χρήστων κάτω από δυναμικές συνθήκες. Η βελτιστοποίηση αυτή μπορεί να οδηγήσει σε σημαντική μείωση κόστους και βελτιωμένη εμπειρία χρήσης, κάτι το οποίο θα συνεισφέρει έντονα σε μία ευρύτερη υιοθέτηση του serverless στο φάσμα του edge-cloud.

Λέξεις Κλειδιά

Serverless, Function-as-a-Service, Kubernetes, OpenFaas, Βαθιά Ενισχυτική Μάθηση, Χρησιμοποίηση Πόρων, Παρεμβολές, Ετερογένεια, DAG

Abstract

The increasing popularity of deploying workflows on cloud premises has skyrocketed. Previously, this deployment was handled by private servers but lately the responsibility has shifted towards public cloud environments. Now, the developers are free from the infrastructure management responsibility and can focus solely on the development of their application code, since the cloud service providers take care of the rest mechanisms.

Function-as-a-Service (FaaS) is a serverless paradigm that enables developers to upload small, practical segments of code to the cloud, that are formatted as functions while being unaware of the underlying infrastructure. Nonetheless, it is quite often that the functions' performance may be undermined by a set of factors that have to be taken into consideration in order to provide the best possible Quality of Experience (QoE) to the end-user. In this direction, a lot of work has been done by both academia and industry, with the goal of making serverless more efficient.

This diploma thesis aims to examine the factors which influence the execution of serverless workload and proposes a dynamic solution in scheduling a serverless application with the aid of Deep Reinforcement Learning. The goal is to serve user requests efficiently under dynamic conditions. This optimization can lead to important cost cuts and improved user experience which contributes highly to a wider adaptation of *serverless* in the edge-cloud continuum.

Λέξεις Κλειδιά

Serverless, Function-as-a-Service, Kubernetes, OpenFaas, Deep Reinforcement Learning Resource Utilization, Interference, Heterogeneity, DAG

στην μητέρα μου Γεωργία

Ευχαριστίες

Με την ολοκλήρωση της παρούσας Διπλωματικής Εργασίας και του προπτυχιακού μου Τίτλου Σπουδών θα ήθελα να ευχαριστήσω εγκαρδίως μια σειρά από ανθρώπους που μου παρείχαν βοήθεια και συμπαράσταση καθόλη την διάρκεια αυτής της πορείας.

Αρχικά, θα ήθελα να ευχαριστήσω τον επιβλέποντα και Καθηγητή μου κ. Δημήτριο Σούντρη, ο οποίος αφενώς μου προσέφερε τη δυνατότητα και την ευκαιρία να αποτελέσω μέρος του Εργαστηρίου Μικροϋπολογιστων και Ψηφιακών Συστημάτων, εκπονώντας την διπλωματική μου εργασία σ' αυτό και αφετέρου όλο τον χρόνο που διέθεσε για πολυάριθμες μακροσκελείς συζητήσεις παντώς θεμάτων, που είχαμε από την πρώτη στιγμή, οι οποίες μου αφήνουν έναν εμπνευστικό αντίκτυπο.

Επιπλέον, θα ήθελα να ευχαριστήσω τους Σωτήριο Ξυδή, Δημοσθένη Μασούρο και Αχιλλέα Τζενετόπουλο για την επιστημονική καθοδήγηση καθ' όλη την διάρκεια της εκπόνησης της συγκεκριμένης Διπλωματικής Εργασίας, που μου έδωσε εφόδια παραπάνω απο πολύτιμα για σφαιρική και τεχνική κατάρτιση στις σύγχρονες τεχνολογίες. Οι ιδέες και προσεγγίσεις που συζητήθηκαν μεταξύ μας ήταν κομβικές για την αρτιότητα του συγκεκριμένου ερευνητικού αποτελέσματος στον χώρο του Cloud-, Serverless-Computing.

Επίσης, θα ήθελα να ευχαριστήσω τους στενούς μου φίλους για την υποστήριξη και όλες τις αμέτρητες, όμορφες στιγμές που έιχαμε μαζί όλα τα χρόνια, τόσο σε ακαδημαϊκό όσο σε και προσωπικό επίπεδο.

Τελευταίο και σημαντικότερο, θα ήθελα να ευχαριστήσω την οικογένεια μου που βρίσκονταν πάντα δίπλα σε οποιοδήποτε επιλογή μου και με ωθούσαν ο καθένας με τον τρόπο του να ακολουθώ τα όνειρα μου.

Αθήνα, Νοέμβριος 2022

Δημήτριος Ι. Γιάγκος

Περιεχόμενα

Κατάλογος Σχημάτων

Εκτεταμένη Περίληψη

0.1 Εισαγωγή

Η εμφάνιση των υπολογιστικών συστημάτων νέφους έχει οδηγήσει στην δραστική αλλαγή της εκτέλεσης υπολογιστικών φορτίων από ιδιωτικά δωμάτια εξυπηρετητών (servers) σε δημόσια περιβάλλοντα νέφους. Η αλλαγή αυτή επέτρεψε στους προγραμματιστές να μεταφέρουν την ευθύνη διαχείρισης εξυπηρετητών και των υποδομών τους στον πάροχο υπηρεσιών νέφους. Προ αυτής της εξέλιξης, οι προγραμματιστές ήταν αδήριτη ανάγκη να αγοράσουν ή νοικιάσουν ιδιωτικές υποδομές εξυπηρετητή για να λειτουργήσουν τα συστήματα που ανέπτυσσαν. Η συγκεκριμένη τακτική είχε ως προαπαιτούμενο υψηλή χρηματική επένδυση σε υλικό ενώ παράλληλα προκαλούνταν αύξηση των λειτουργικών εξόδων λόγω της ανάγκης πρόσληψης προσωπικού για την λειτουργία και συντήρηση των υλικών υποδομών, πάνω στις οποίες στηρίζονταν τα αναπτυσσόμενα εργαλεία λογισμικού. Επιπρόσθετα, αν δημιουργούνταν η ανάγκη για αύξηση της υπολογιστικής ισχύος, αυτό θα απαιτούσε αρκετό χρόνο μιας και οι διαδικασίες αγοράς, εγκατάστασης και ρύθμισης νέου υλικού θα έπρεπε να έχουν ολοκληρωθεί προτού μπορέσουν οι υποδομές αυτές να χρησιμοποιηθούν. Πλεόν, με τις νέες εξελίξεις, είναι δυνατό να ελέγξει κανείς νέες υποδομές εξυπηρετητών σε ελάχιστο χρόνο κάνοντας χρήση εμπορικών πλατφόρμων νέφους.

Με τις πρόσφατες λύσεις στο υπολογιστικό νέφος, η ανάπτυξη υπηρεσιών αποτελεί ένα προϊόν με χαμηλό χρόνο άφιξης στην αγορά ενώ παράλληλα οι προγραμματιστές δεν είναι ανάγκη να διαχειρίζονται τις ανάλογες υπολογιστικές υποδομές. Οι εμπορικές πλατφόρμες νέφους συνεχώς εξελίσσονται ώστε να παρέχουν νέα μοντέλα υπηρεσιών που θα δημιουργήσουν χώρο για περαιτέρω αποποίηση ευθήνης από τους προγραμματιστές και μεταφοράς της στους παρόχους νέφους. Η τελευταία προσθήκη σε αυτόν τον τομέα υπολογιστικών υπηρεσιών είναι το serverless computing το οποίο υπόσχεται να απαλλάξει πλήρως τον χρήστη από την διαχείριση υπολογιστικών υποδομών.

Η Συνάρτηση Σαν Υπηρεσία (FaaS) είναι ένα υπολογιστικό μοντέλο που επιτρέπει στον προγραμματιστή να εκτελέσει ατομικές συναρτήσεις στο νέφος. Η FaaS κυριάρχησε χάρη στην εικονικοποίηση σε επίπεδο λειτουργικού συστήματος. Το υπολογιστικό μοντέλο Serverless δημιουργεί ένα νέο τρόπο σχεδιασμού και κλιμάκωσης εφαρμογών και υπηρεσιών, επιτρέποντας στος προγραμματιστές να διαφύγουν από την συνηθισμένη τακτική ανάπτυξης μονολοθικών εφαρμογών και να υιοθετήσουν ένα πιο αποδομημένο ύφος που προσφέρεται από τις serverless συναρτήσεις, χωρίς ανάγκη για επιπλέον συντήρηση, κλιμάκωση και διαχείριση των πόρων τους.

0.2 Ιστορικό για την Εικονικοποίηση

Η εικονικοποίηση (virtualization) είναι ο ακρογωνιαίος λίθος της τεχνολογίας serverless και γενικότερα του υπολογιστικού νέφους. Η εικονικοποίηση δημιουργεί καλύτερες συνθήκες για μεταφορά των υπολογιστικών φορτίων και καθιστά εύκολο στους παρόχους υπηρεσιών στο νέφος να μοιράζουν στους πελάτες τους υπολογιστική ισχύ. Στην συνέχεια θα αναφερθούμε σε δύο είδη εικονικοποίησης: *εικονικοποίηση βασισμένη σε επιτηρητή* και *εικονικοποίηση βασισμένη σε πακέτο* (container).

0.2.1 Εικονικοποίηση Βασισμένη σε Επιτηρητή

Από την δεκαετία του ′60 και του ′70, η εικονικοποίηση άρχισε να κυριαρχεί σαν υπολογιστικό μοντέλο, μιας και η IBM [1] είχε χρησιμοποιήσει την συγκεκριμένη τεχνολογία στα συστήματα της. Στην εικονικοποίηση βασισμένη σε επιτηρητή [2], προσφέρεται ένα ολοκληρωμένο εικονικό μηχάνημα που εικονικά χρησιμοποιεί όλο το υλικό hardware του συστήματος, του οποίου το λειτουργικό σύστημα αγνοεί την εικονικοποίηση των υπολογιστικών πόρων. Οι επιτηρητές μπορούν να διακριθούν σε δύο κατηγορίες: Επιτηρητής γυμνού μετάλλου και φιλοξενούμενος επιτηρητής, όπου έκαστος έχει διαφορετικές περιπτώσεις χρήσης.

0.2.2 Εικονικοποίηση Βασισμένη σε Πακέτο

Αντιθέτως με την εικονικοποίηση βασισμένη σε επιτηρητή η οποία λειτουργεί με ένα λειτουργικό σύστημα να ενεργεί ¨πάνω¨ από εικονικοποιημένο υλικο (hardware), η εικονικοποίηση βασισμένη σε πακέτο λειτουργεί στο ίδιο επίπεδο με το λειτουργικό σύστημα. Το πακέτο container είναι μια αυτοτελής μονάδα λογισμικού που περιέχει τον κώδικα αλλά και όλες τις εξαρτήσεις του συγκεκριμένου κώδικα ώστε ο κώδικας να μπορεί να εκτελεστεί γρήγορα και αξιόπιστα σε ποικίλα υπολογιστικά περιβάλλοντα. Τα πλεονεκτήματα από την χρήση πακέτων συγκριτικά με την χρήση εικονικών μηχανημάτων είναι αρκετά. Το σημαντικότερο είναι η ευελιξία και ο γρηγορότερος κύκλος ανάπτυξης και διάθεσης του περιεχόμενου λογισμικού.

0.2.3 Ενορχήστρωση

Στα περισσότερα υπολογιστικά μηχανήματα υπάρχει η ανάγκη ενορχήστρωσης και διαχείρισης πολλών πακέτων containers. Την ανάγκη αυτή καλύπτουν συστήματα όπως ο Κυβερνήτης[3]. Ο Κυβερνήτης είναι ένα σύστημα ανοιχτού κώδικα που αυτοματοποιεί την εκτέλεση, την κλιμάκωση και την διαχείριση πακεταρισμένων εφαρμογών. Η Google αρχικά σχεδίασε το σύστημα του Κυβερνήτη, το οποίο πλέον συντηρείται απο την CNCF. Όταν εγκαθιστούμε τον Κυβερνήτη, αποκτάμε μία ομάδα στοιχείων λογισμικού. Η ομάδα αποτελείται από κόμβους τύπου Αφέντη (μαστερ) και κόμβους τύπου Εργάτη (ωορκερ). Ένας κόμβος τύπου Αφέντη περιλαμβάνει: kube-apiserver, etcd, kube-scheduler, kube-controller-manager. Ένας κόμβος τύπου Εργάτη περιλαμβάνει: kubelet, kube-proxy, Container runtime.

Ο Κυβερνήτης λειτουργεί με πολλαπλά επίπεδα αφαίρεσης, τα οποία σε φθίνουσα σειρά αφαίρεσης είναι τα εξής: Deployment, ReplicaSet, Pod, Cluster Node, Node Process, Docker container. Συγκεκριμένα, για να εκτελέσουμε μία εφαρμογή την τοποθετούμε σ' ένα πακέτο container το οποίο θα διαχειριστεί από ένα Pod, το οποίο με την σειρά του θα διαχειριστεί από λογισμικό μεγαλύτερου επιπέδου αφαίρεσης. Ο Κυβερνήτης υποστηρίζει μία μεγάλη ποικιλία από λειτουργίες που καλύπτουν ένα μέγαλο φάσμα των δυνατοτήτων που προσφέρει η τεχνολογία του υπολογιστικού νέφους. Βασική ευθύνη του Κυβερνήτη είναι να εκτελέσει ένα φορτίο μέσω ενός Pod στο καταλληλότερο διαθέσιμο μηχάνημα ώστε να επιτευχθεί η καλύτερη δυνατή διαχείριση των διαθέσιμων πόρων.

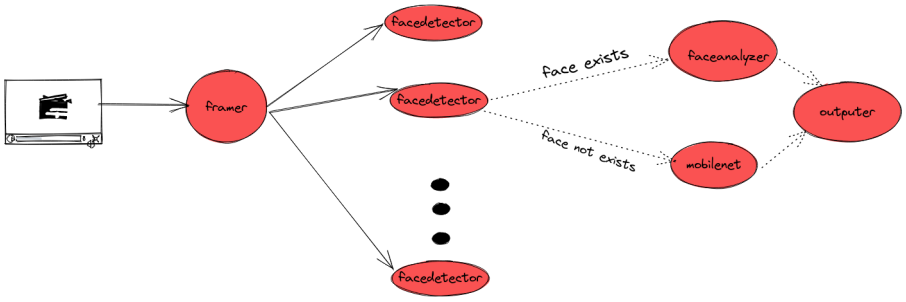
0.3 Ανάλυση Κινήτρων

Το υπολογιστικό νέφος έχει παρουσιασεί αλματώδη πρόοδο μετά την εικονικοποίηση του υλικού hardware και της ιδέας των εικονικών μηχανημάτων. Για χρόνια, πολλές πρότυπες πλατφόρμες δημιουργήθηκαν χάρη σ' αυτές τις τεχνολογίες που οδήγησαν σε μείωση κόστους σε όλα τα μεγέθη εταιριών. Το επόμενο βήμα ήταν η ανάπτυξη της εικονικοποίησης βασισμένης σε πακέτα η οποία με την σειρά της δημιούργησε χώρο για συνεχή πρόοδο. Όμως η ανάγκη για μεγαλύτερη ευελιξία και πιο δίκαιη κοστολόγιση έφερε στο προσκήνιο το serverless. Το serverless ενώ είχε εκκινήσει σαν τεχνολογία βασισμένη σε γεγονότα events, έφτασε να χρησιμοποιείται για εφαρμογές ανάλυσης δεδομένων, τεχνητής νοημοσύνης και υπολογισιμότητα υψηλής απόδοσης. Λόγω αυτού του διευρημένου φάσματος εφαρμογών καταλαβαίνουμε ότι ο αντίκτυπος που μπορεί να έχει μία βελτίωση στο serverless θα ωφελήσει μία σειρά άλλων πεδίων. *Ποιοι είναι όμως οι παράγοντες που επηρέαζουν την απόδοση μίας serverless εφαρμογής που εκτελείται σε μία ομάδα υπολογιστών (cluster) στο νέφος·* Αυτούς τους παράγοντες εξετάσαμε σε μία σειρά πειραμάτων που θα αναλύσουμε ακολούθως.

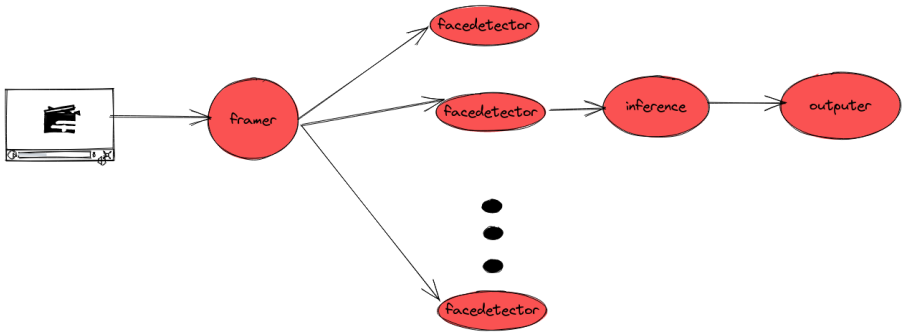
0.3.1 Περιγραφή της Serverless Εφαρμογής και των Εφαρμογών Πίεσης

Η serverless εφαρμογή που επιλέξαμε να χρησιμοποιήσουμε στην μελέτη της συγκεκριμένης διπλωματικής εργασίας αποτελείται από ένα σύνολο συναρτήσεων που επεξεργάζονται στιγμιότυπα ενός αρχείου βίντεο κάνοντας ανάλυση του περιεχόμενου κάθε στιγμιότυπου μέσω μοντέλων τεχνητής νοημοσύνης. Σε πιο αφαιρετικό επίπεδο, η εφαρμογή δέχεται αρχικά στην είσοδο ένα οποιοδήποτε αρχείο βίντεο τύπου mp4. Στην συνέχεια, η πρώτη συνάρτηση, framer, εξάγει, με συγκεκριμένο βήμα, στιγμιότυπα από το συγκεκριμένο βίντεο τα οποία προωθεί σ' ένα μοντέλο εύρεσης προσώπου, που ονομάζεται facedetector. Το μοντέλο αυτό, αφού αποφανθεί την παρουσία ή την απουσία ανθρώπινου προσώπου στέλνει το στιγμιότυπο είτε σ' ένα μοντέλο ανάλυσης συναισθημάτων, faceanalyzer, ή σ' ένα μοντέλο αναγνώρισης αντικειμένου, mobilenet, αντιστοίχως.

Προκειμένου να υλοποιήσουμε την εκτέλεση της συγκεκριμένης εφαρμογής, στραφήκαμε αρχικά στις επιλογές που προσφέρει το Apache OpenWhisk[4]. Το συγκεκριμένο εργαλείο όμως δεν επιτρέπει εύκολα την τοποθέτηση μίας συνάρτησης σε κόμβο της επιλογής του χρήστη αλλά δρομολογεί την συνάρτηση όπου του δώσει εντολή ο εσωτερικός δρομολογητής του με αποτέλεσμα να αποτελεί τροχοπέδη στο δικό μας εγχείρημα. Η τοποθέτηση



Σχήμα 1: *Αρχιτεκτονική της Version1*

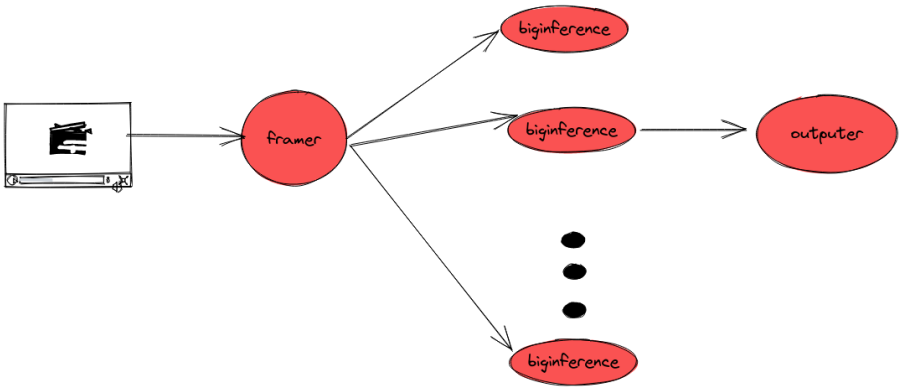


Σχήμα 2: *Αρχιτεκτονική της Version2*

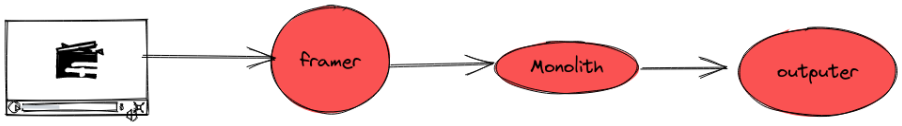
συνάρτησης σε κόμβο επιλογής είναι βασικό στοιχείο της συγκεκριμένης δουλείας μιας και στόχος είναι να δρομολογούμε κάποια συνάρτηση στον καταλληλότερο κόμβο βάση κάποιων παραμέτρων. Το επόμενο βήμα ήταν η αντικατάσταση του Apache OpenWhisk με το OpenFaas[5], το οποίο όντας πιο κοντά στην υλοποίηση του Κυβερνήτη προσφέρει στον χρήστη μία σειρα ιδιαίτερα χρήσιμων δυνατοτήτων, όπως αυτή της τοποθέτησης μίας συνάρτησης σε κόμβο της επιλογής μας.

Μέσω του OpenFaas δημιουργήσαμε serverless συναρτήσεις που υλοποιούν την εφαρμογή μας και τις εκτελέσαμε ενορχηστρωμένα με δύο εργαλεία. Το πρώτο εργαλείο που χρησιμοποιήσαμε ήταν το faas-flow[6], ένα λογισμικό ανοιχτού κώδικα που επιτρέπει την σύνθεση μίας πολύπλοκης serverless συνάρτησης που αποτελείται από τις συναρτήσεις που επιθυμούμε να συνδέσουμε και να εκτελέσουμε ενορχηστρωμένα. Κάθε φόρα που θα καλείται η πολύπλοκη συνάρτηση, θα εκτελείται μία φορά η συνολική εφαρμογή. Σαν δεύτερο εργαλείο χρησιμοποιήσαμε μία πλατφόρμα που υλοποιήθηκε από μας και είναι σχεδιασμένη για την συγκεκριμένη εφαρμογή που μελετάμε. Η πλατφόρμα αναπτύχθηκε στην γλώσσα προγραμματισμού Python και επιτρέπει και η ίδια με την σειρά της ενορχηστρωμένη εκτέλεση των συναρτήσεων που έχουμε δημιουργήσει σε προηγούμενο στάδιο. Περισσότερες πληροφορίες για τον custom μηχανισμό εκτέλεσης μπορούν να βρεθούυν εδώ.

Προκειμένου να μελετήσουμε την εφαρμογή από διαφορετικές οπτικές γωνίες, κατασκευάσαμε τέσσερις μορφές της ίδιας εφαρμογής με διαφορετικό αριθμό συναρτήσεων. Προκειμένου να μειωθεί ο αριθμός των συναρτήσεων σε κάθε επόμενη μορφή ενσωματώνουμε δύο συναρτήσεις σε μία. Οι τέσσερις μορφές παρουσιάζονται παρακάτω σε φθίνουσα σειρά αριθμού συναρτήσεων.



Σχήμα 3: *Αρχιτεκτονική της Version3*



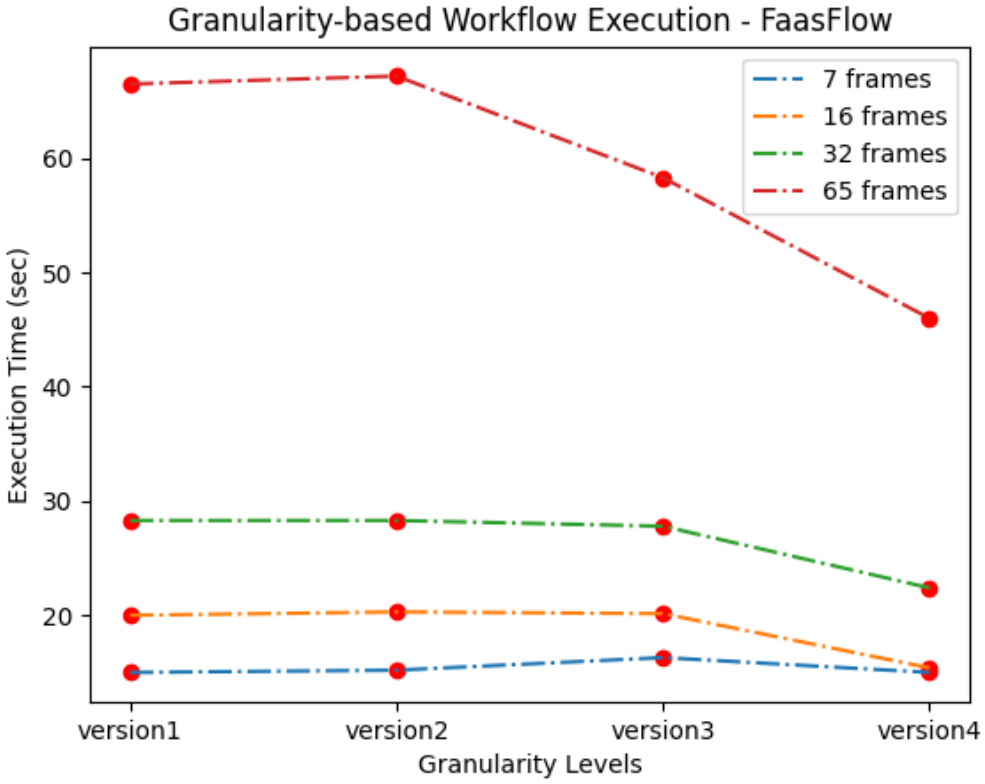
Σχήμα 4: *Αρχιτεκτονική της Version4*

0.3.2 Αντίκτυπος του Αριθμού Συναρτήσεων στην Απόδοση

Μία serverless εφαρμογή συνήθως αποτελείται από πολλαπλές συναρτήσεις που συνδέονται μεταξύ τους αλυσιδωτά σχηματίζοντας ένα DAG, το οποίο αναλαμβάνει την συνολική εκτέλεση της εφαρμογής. Ο αριθμός συναρτήσεων που αποτελείται το DAG επηρεάζει άμεσα την ανάγκη για επικοινωνία μεταξύ των συναρτήσεων και άρα παίζει βασικό ρόλο στην απόδοση της εφαρμογής. Η συγκεκριμένη επιρροή ενισχύεται ακόμη περισσότερο από το γεγονός ότι όλες οι πρόσφατες serverless πλατφόρμες εκτελούν κάθε συνάρτηση εκκινώντας ένα νέο πακέτο container. Αν για κάποιο λόγο δύο εξαρτώμενες συναρτήσεις ενοποιηθούν σε μία, η συνολική επικοινωνία θα μειωθεί αλλά αυτό θα μειώσει επίσης και την ικανότητα της εφαρμογής να παραλληλοποιηθεί. Η σχεδιαστική επιλόγη άρα ως προς την παραλληλοποιησιμότητα της εφαρμογής ή την μείωση του αριθμού των συναρτήσεων είναι καθοριστικής σημασίας και πρέπει να λαμβάνεται υπόψιν πριν την επαναλαμβανόμενη εκτέλεση της σε περιβάλλοντα νέφους. Η μελέτη του αντικτύπου του αριθμού των συναρτήσεων στην απόδοση της εφαρμογής έγινε και με τα δύο προαναφερθέντα εργαλεία.

Προσέγγιση με το εργαλείο faas-flow

Όπως αναφέρθηκε παραπάνω, τέσσερις μορφές της ίδιας εφαρμογής δημιουργήθηκαν με στόχο η καθεμία να εκπροσωπεί όλο και λιγότερη συμπαγή εφαρμογή. Η λιγότερη συμπαγή από όλες τις μορφές είναι η Version1 η οποία αποτελείται από πέντε διαφορετικές συναρτήσεις ενώ φτάνουμε στην περισσότερο συμπαγή μορφή εν τέλει, την Version4 που αποτελείται συνολικά από μόλις δύο συναρτήσεις. Στο σχήμα 5 παρουσιάζουμε την συγκριτική επίδοση των τεσσάρων μορφών όπου εκτελούνται με τέσσερα είδη εισόδου και καλούνται να επεξεργαστούν 7, 16, 32 και 65 στιγμιότυπα ανά περίπτωση εισόδου. Όπως ήταν αναμενόμενο, η Version4 στην μεγαλύτερη είσοδο (65 στιγμιότυπα) είναι κατά 30% γρηγορότερη απο την Version1. Επιπρόσθετα, στις περιπτώσεις εισόδου 7, 16 και 32 στιγμιότυπων οι δια-



Σχήμα 5: *Μέσος χρόνος καθυστέρησης εκτέλεσης για τις 4 εκδόσεις με είσοδο 65 στιγμιότυπα*

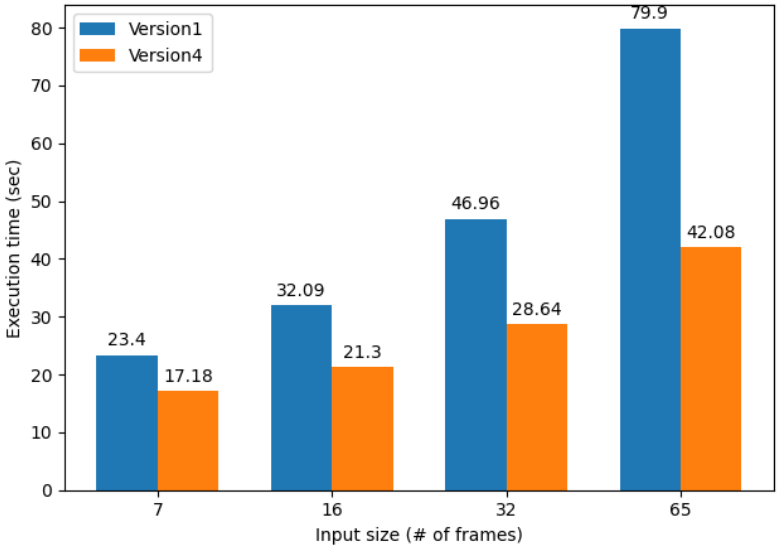
φορές μεταξύ Version1, Version2, Version3 είναι μικρές κάτι το οποίο οφείλεται στο ότι οι συγκεκριμένοι αριθμού στιγμιότυπων δεν είναι αρκετά μεγάλοι για να αποτελέσουν κώλυμα στις συγκεκριμένες μορφές της εφαρμογής.

Προσέγγιση με το custom εργαλείο

Σε δεύτερη σειρά πειραμάτων που εκτελέστηκαν υπό το custom εργαλείο που αναπτύξαμε για τις OpenFaas συναρτήσεις συγκρίνουμε την επίδοση της Version1 με την Version4 όντας η πιο αποδομημένη και η πιο συμπαγής μορφή της εφαρμογής αντιστοίχως. Τα αποτελέσματα των πειραμάτων φαίνονται στο σχήμα 6, όπου παρατηρούμε ειδικά στην μεγαλύτερη είσοδο (65 στιγμιότυπα) την Version4 να είναι 46.5% ταχύτερη από την Version1 ενώ στο αντίστοιχο πείραμα χρησιμοποιώντας το πρώτο εργαλείο η αντίστοιχη διαφορά ήταν ίση με 30%. Ένα ασφαλές συμπέρασμα που μπορεί να εξαχθεί απο αυτές τις σειρές πειραμάτων είναι ότι ανεξαρτήτως από τον μηχανισμό εκτέλεσης της εφαρμογής, ο βαθμός αποδόμησης μίας εφαρμογής επηρεάζει έντονα την χρονική επίδοση της, δεδομένου ότι η εφαρμογή δεν παραλληλοποιηθεί.

0.3.3 Αντίκτυπος των Παρεμβολών στην Απόδοση

Σ' αυτή την υπο-ενότητα αξιολογούμε την συμπεριφόρα διαφορετικών τοπολογιών της Version1 και της Version4 όταν στα μηχανήματα επενεργούμε πίεση από τρίτες παρεμβολές, υπό ρεαλιστικές δηλαδή συνθήκες. Με αυτό τον τρόπο μπορούμε να ξεχωρίσουμε τους



Σχήμα 6: *Συγκριτική εκτέλεση των Version1, 4 στον κόμβο PM*

παράγοντες που επηρέαζουν λιγότερο ή περισσότερο την επίδοση της εφαρμογής όταν εκτελείται σ' ένα υπολογιστικό περιβάλλον νέφους. Για παράδειγμα, μαθαίνοντας ότι η εφαρμογή μας απαιτεί παρατεταμένη χρήση πόρων της CPU, το να την δρομολογήσουμε σ' ένα κόμβο του οποίου η επεξεργαστική ισχύς καπιταλεύεται ήδη από άλλα υπολογιστικά φορτία είναι κακή επιλογή ως προς την εξυπηρέτηση του χρήστη. Η μελέτη γίνεται χρησιμοποιώντας και τους δύο προαναφερθέντες μηχανισμούς εκτέλεσης.

Προσέγγιση με το εργαλείο faas-flow

Προκειμένου να μελετήθει ο αντίκτυπος των παρεμβολών στο προφίλ της serverless εφαρμογής, διαφορετικές τοπολογίες της Version1 σε διάφορους κόμβους δημιουργόντας μια ποικιλία τοπολογιών οι οποίες παρουσίαζονται στον πίνακα 1. Συγκεκριμένα, μετρήσαμε τους χρόνους εκτέλεσης για τρία διαφορετικά μεγέθη εισόδου: 16, 32 και 65 στιγμιότυπα ώστε να εξάγουμε ασφαλέστερα συμπεράσματα για την συσχέτιση μεταξύ χρόνου εκτέλεσης και παρεμβολών. Τα συγκεντρωτικά αποτελέσματα για διάφορα επίπεδη πίεσης των πόρων παρουσίαζονται στα σχήματα 7 και 8.

Καταλήξαμε στις εξής παρατηρήσεις: Αρχικά το μέγεθος της εισόδου φαίνεται να μην επηρέαζει σημαντικά την ¨μάχη¨ μεταξύ των τοπολογιών για την ταχύτερη εκτέλεση, κάτι το οποίο πηγάζει από το γεγονός ότι η πιο αργή συνάρτηση, framer, ήταν τοποθετημένη σ' όλες τις τοπολογίες στον ίδιο κόμβο (Davinci). Το εικονικό μηχάνημα που ¨φιλοξενεί¨ ο server Davinci είναι υπολογιστικά το πιο δυνατό. Επιπλέον, οι διάφορες τοποθετήσεις των υπόλοιπων συναρτήσεων λόγω του μικρού χρόνου εκτέλεσης που παρουσιάζουν συγκριτικά με την συνάρτηση framer δεν επηρεάζουν σε μεγάλο βαθμό τις συγκριτικές καθυστερήσεις των διαφόρων τοπολογιών.

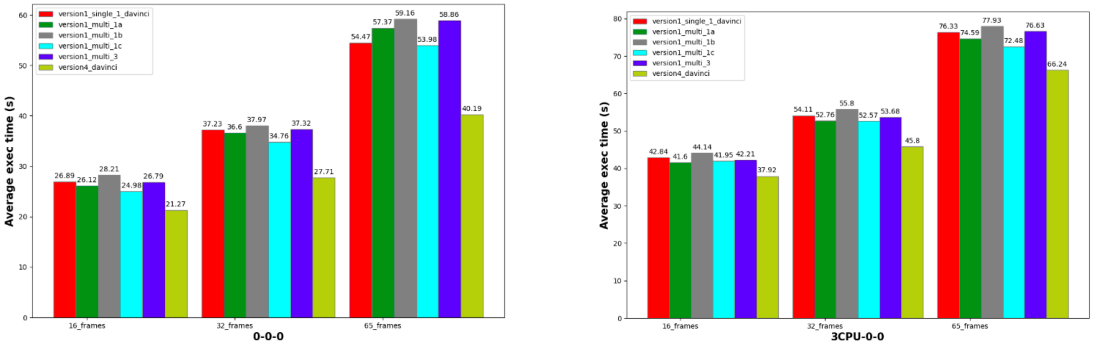
Επιπρόσθετα, η Version4 ήταν ταχύτερη σ' όλες τις περιπτώσεις το οποίο ήταν αναμενόμενο λόγω της χαμήλης ανάγκης της για επικοινωνία μεταξύ των συναρτήσεων που την απαρτίζουν. Είναι αξιοσημείωτο βέβαια πως όταν στους κόμβους ενήργησαν παρεμβολές τρίτων εφαρμογών οι διαφορές επίδοσης μεταξύ της Version4 και των εκδόσεων της Version1

μειώθηκαν σημαντικά διότι κάποιες από τις συναρτήσεις της Version1 τύχαινε να έχουν τοποθετηθεί σε κόμβο που δεν δέχεται παρεμβολές και άρα να διαθέτουν όλους τους πόρους του συγκεκριμένου μηχανήματος σε αποκλειστικότητα.

Τελικά, απ' όλα τα είδη παρεμβολών που δημιουργήσαμε οι παρεμβολές στην L3 cache ήταν αυτές που δημιούργησαν την μεγαλύτερη συμφόρηση στα μηχανήματα.

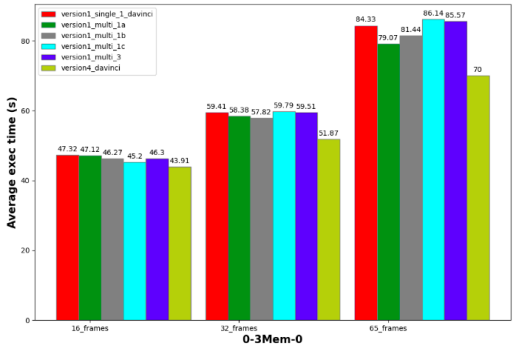
Functions	**Singlenode**	**Multinode 1a**	**Multinode 1b**	**Multinode 1c**	**Multinode 3**	**Version4**
Framer	w01	w01	w01	w01	w01	w01
Facedetector	w01	w02	w02	w01	w01	w01
Faceanalyzer	w01	w03	w02	w02	w01	w01
Mobilenet	w01	w01	w01	w02	w02	w01
Outputer	w01	w02	w02	w02	w02	w01
Wrapper	w01	w01	w01	w02	w02	w01

Πίνακας 1: *Τοπολογίες συναρτήσεων*

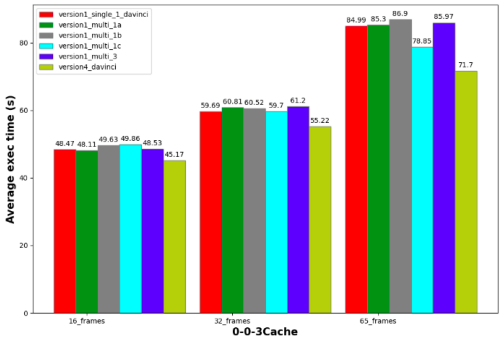


(α') *Zero Interference* (β') *CPU-pressure Intereference*

Σχήμα 7: *Version1 configurations and Version4(1)*



(α΄) *Memory-pressure Intereference*



(β΄) *L3-pressure Intereference*

Σχήμα 8: *Version1 configurations and Version4(2)*

Προσέγγιση με το custom εργαλείο

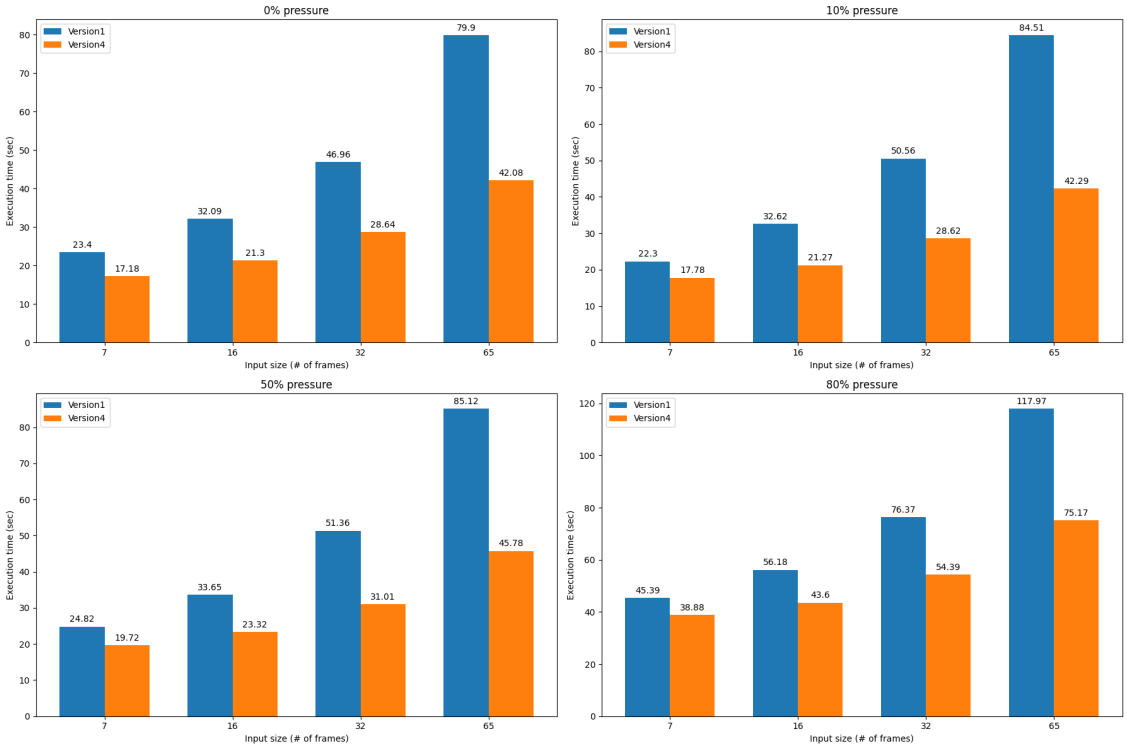
Σε δεύτερη σειρά αντίστοιχων πειραμάτων αλλά αυτή την φορά χρησιμοποιώντας τον custom μηχανισμό εκτέλεσης, τοποθετήσαμε και την Version1 (ολιστικά) και την Version4 σε ένα κόμβο κάθε φορά επί του συνόλου των διαθέσιμων κόμβων. Στον συγκεκριμένο κόμβο δημιουργήσαμε περιβάλλον παρεμβολών σε 4 επίπεδα πίεσης: 0%, 10%, 50%, 80%. Τα συγκεντρωτικά αποτελέσματα παρουσίαζονται στα σχήματα 9, 10, 11 και 12.

- Όσο μεγαλύτερη η πίεση παρεμβολών, τόσο μικρότερη η διαφόρα επίδοσης μεταξύ Version1, Version4

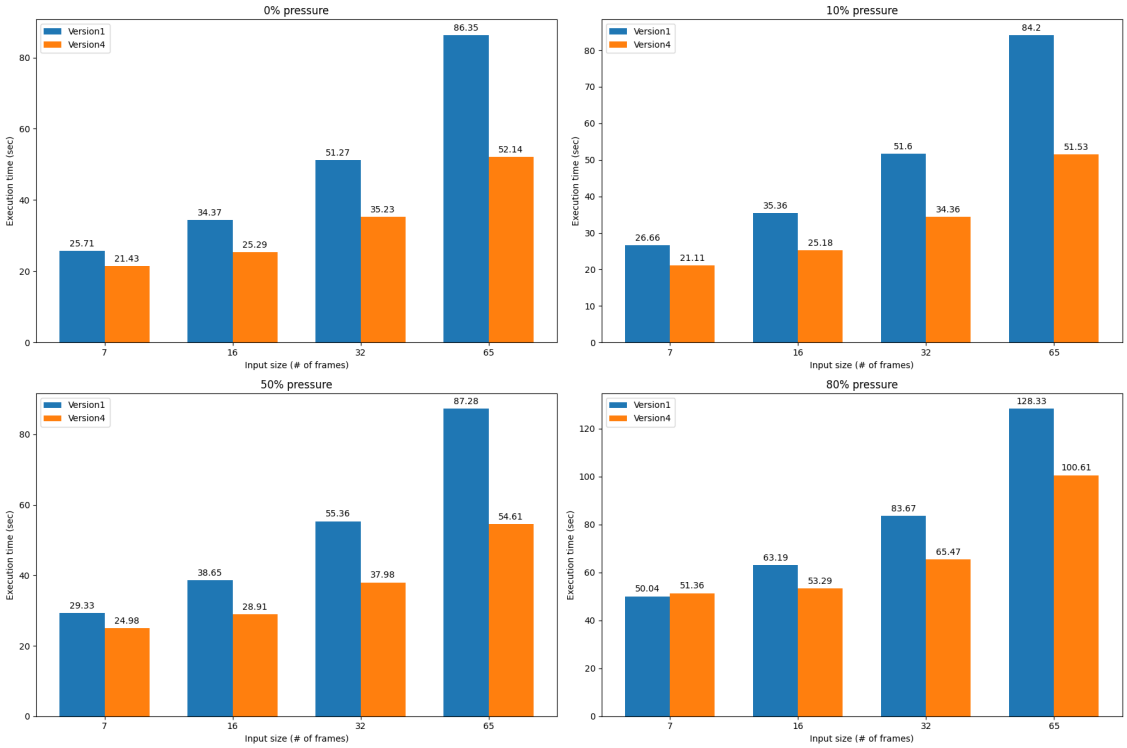
- Η διαφορά μεταξύ 0% και 80% πίεσης έχει ως αποτέλεσμα 33%, 33%, 30% και 47% καθυστέρηση στον συνολικό χρόνο μίας εκτέλεσης της εφαρμογής στους κόμβους w01, w02, w03, w04 αντιστοίχως

- Τα επίπεδα παρεμβολών 10% και 50% οριακά ¨απορροφόνται¨ από τους πόρους των κόμβων w01, w02, w03 επειδή μένουν ελεύθεροι, στην χειρότερη περίπτωση, οι μισοί πυρήνες που είναι αρκετοί για να εκτελέσουν αποδοτικά την συγκεκριμένη εφαρμογή με πολλαπλά νήματα. Αντιθέτως, το επίπεδο παρεμβολών 50% στον κόμβο w04 δεσμεύει δύο από τους συνολικούς τέσσερις πυρήνες του μηχανήματος και έτσι αφαιρεί από τον κόμβο την ικανότητα να εκμεταλλευτεί τα πολλαπλά νήματα.

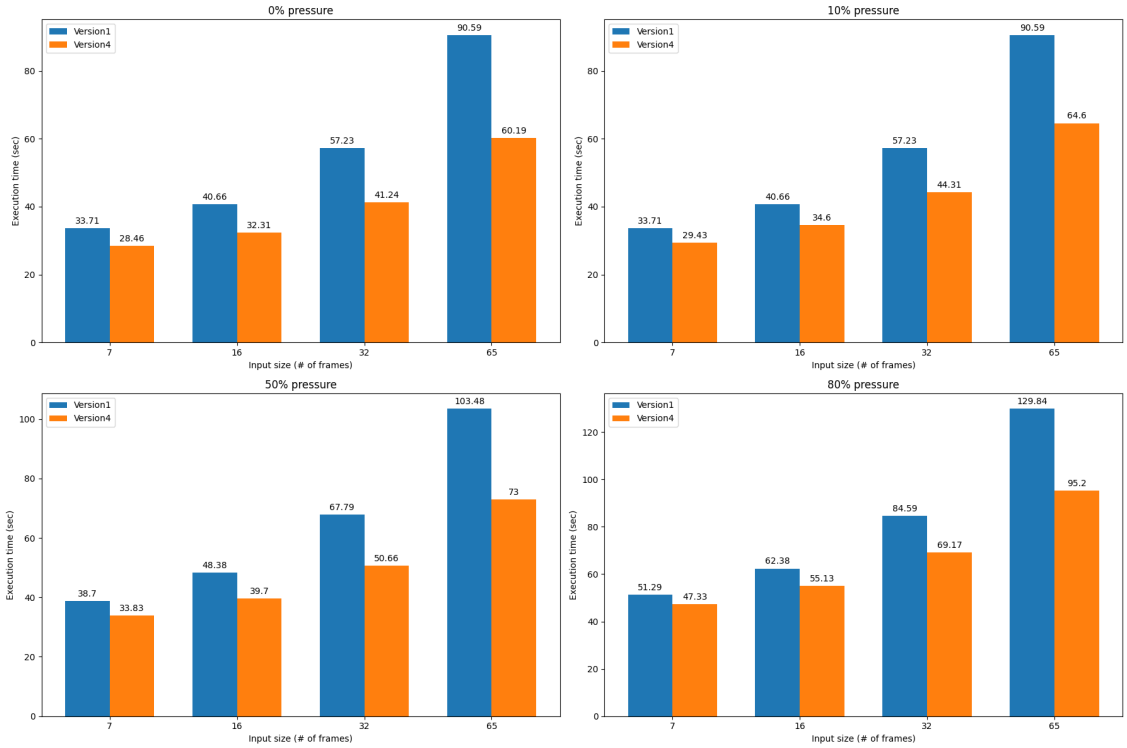
Προκειμένου να χαρακτηρίσουμε την ευαισθησία των συναρτήσεων σε συνθήκες παρεμβολών από τρίτες εφαρμογές, εκτελούμε τρίτες εφαρμογές με την βοήθεια της σουίτας iBench ώστε να αυξήσουμε το υπολογιστικό φορτίο στους κόμβους του συστήματος. Όπως φαίνεται στην εικόνα 13, η εφαρμογή μας παρουσίαζει σημαντικές μεταβολές στην επίδοση της κάτω από συνθήκες CPU παρεμβολών που φτάνουν 57.6% απόκλιση στην περίπτωση της συνάρτησης Framer και 47.2% χειρότερη επίδοση στην περίπτωση των συναρτήσεων των μοντέλων.



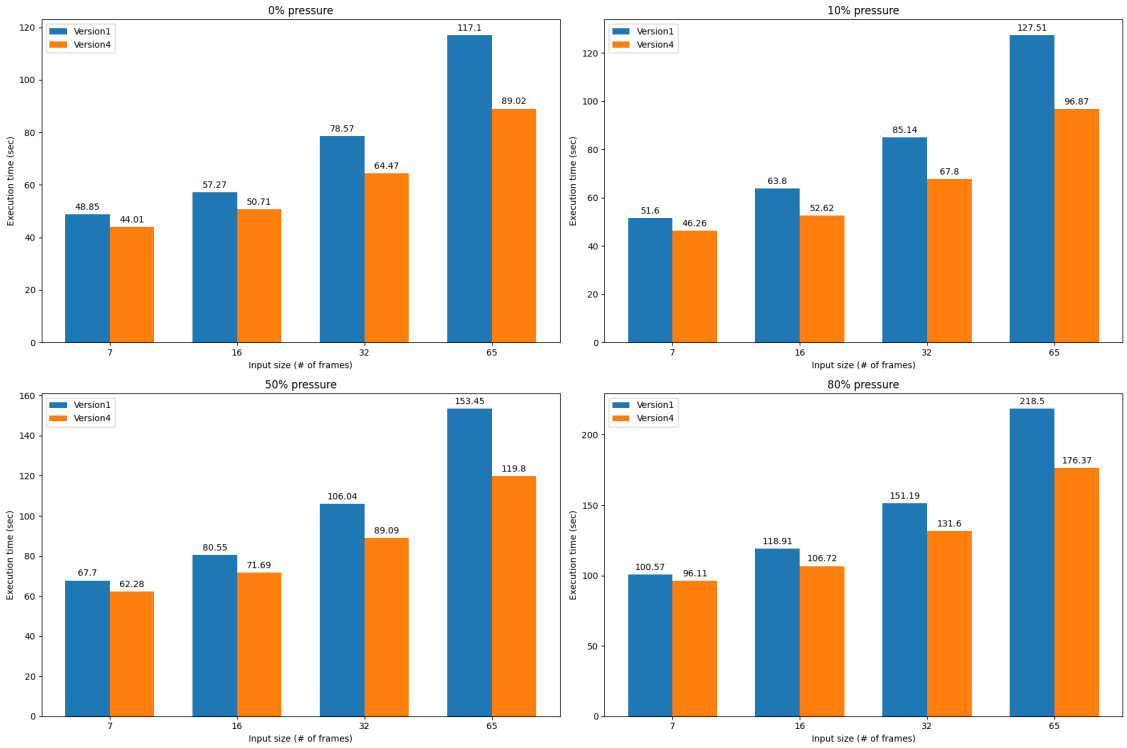
Σχήμα 9: *Συγκριτικές εκτελέσεις μεταξύ Version1 και Version4 στον κόμβο w01*



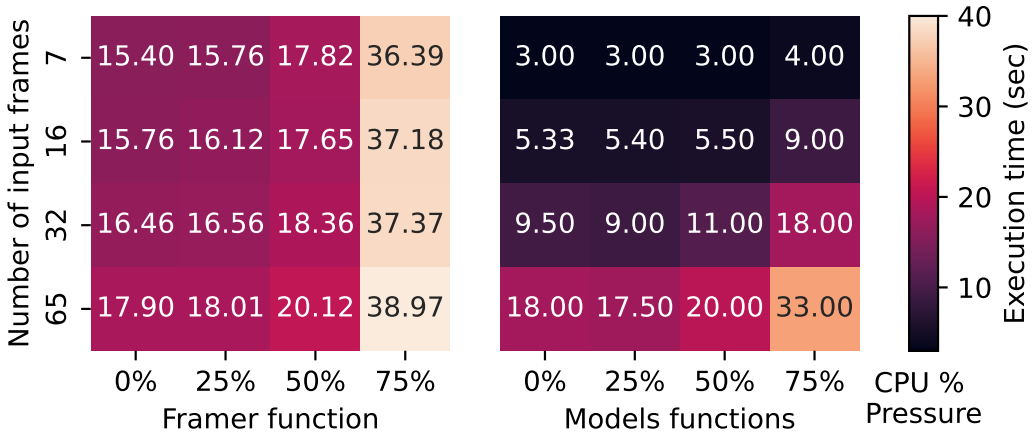
Σχήμα 10: *Συγκριτικές εκτελέσεις μεταξύ Version1 και Version4 στον κόμβο w02*



Σχήμα 11: *Συγκριτικές εκτελέσεις μεταξύ Version1 και Version4 στον κόμβο w03*



Σχήμα 12: *Συγκριτικές εκτελέσεις μεταξύ Version1 και Version4 στον κόμβο w04*



Σχήμα 13: *Αντίκτυπος των παρεμβολών στις συναρτήσεις της εφαρμογής*

0.3.4 Αντίκτυπος της Ετερογένειας στην Απόδοση

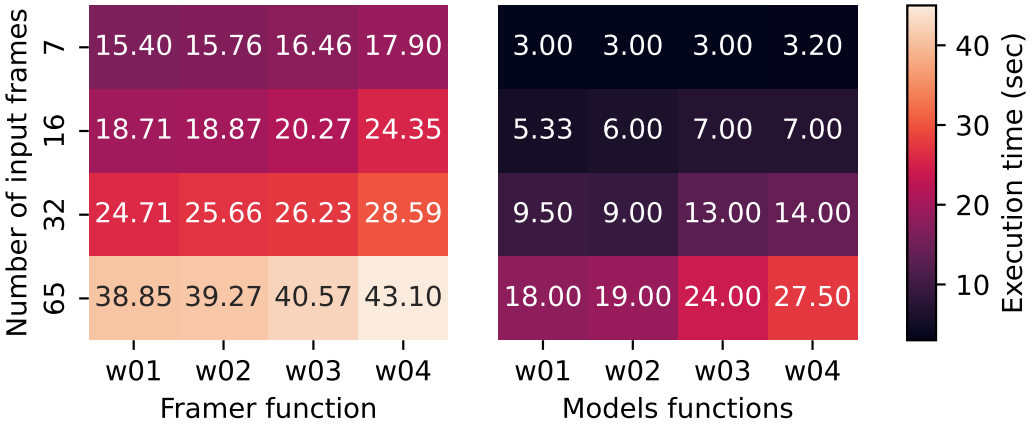
Προσέγγιση με το εργαλείο faas-flow

Προκειμένου να μελετήθει και ο αντίκτυπος της ετερογένειας στο προφίλ της serverless εφαρμογής, τοποθετήσαμε τις διαφορετικές τοπολογίες της Version1 σε διαφορετικούς κόμβους δημιουργόντας μια ποικιλία τοπολογιών οι οποίες παρουσίαζονται στον πίνακα 1. Συγκεκριμένα, μετρήσαμε τους χρόνους εκτέλεσης για τρία διαφορετικά μεγέθη εισόδου: 16, 32 και 65 στιγμιότυπα ώστε να εξάγουμε ασφαλέστερα συμπεράσματα για την συσχέτιση μεταξύ χρόνου εκτέλεσης και ετερογένειας. Τα συγκεντρωτικά αποτελέσματα παρουσίαζονται και πάλι στα σχήματα 7 και 8.

- Είναι προφανές ότι ο κόμβος w01 παρουσιάζει την πιο κυρίαρχη επίδοση ανεξαρτήτως μεγέθους εισόδου. Ο κόμβος w02 έρχεται δεύτερος σε επίδοση, ενώ οι κόμβοι w03, w04 έρχονται τρίτος και τελευταίος αντίστοιχα.
- Αν επικεντρωθούμε στην μεγαλύτερη είσοδο (65 στιγμιότυπα), παρατηρούμε ότι οι διαφορές μεταξύ Version1 και Version4 μειώνεται γραμμικά με την ισχύ των μηχανημάτων. Στον κόμβο w01, η διαφορά αυτή έχει την μέγιστη τιμή της και είναι ίση με 45% ενώ στους κόμβους w02, w03, w04 είναι ίση με 39%, 33% και 23% αντιστοίχως.
- Το συγκεκριμένο φαινόμενο επικρατεί σ' όλα τα μεγέθη εισόδου, αλλά μεγιστοποιείται στην μεγαλύτερη είσοδο.
- Όλες οι serverless συναρτήσεις εκτελούνται με πολλαπλά νήματα (threads) από τους διαθέσιμους πυρήνες κάθε μηχανήματος. Αυτή είναι η βασική αιτία πίσω από την χαμήλη επίδοση του κόμβου w04 ο οποίος διαθέτει συνολικά 4 πυρήνες μόλις και έτσι δεν είναι ικανός να επωφεληθεί της πολυνηματικής εκτέλεσης.

Στην εικόνα 14 παρουσιάζονται οι μεταβολές στην επίδοση των συναρτήσεων που απαρτίζουν την εφαρμογή μας, οι οποίες προκαλούνται από την ετερογένεια των μηχανήματων τα οποία τις εκτελούν. Για την συνάρτηση *Framer*, βρίσκουμε αποκλίσεις με μέγιστη τιμή 23% και ελάχιστη τιμή 10% στις περιπτώσεις της εισόδου 16 και 65 στιγμιοτύπων αντίστοιχα. Επιπλέον, στις συναρτήσεις των μοντέλων, οι μετρημένες αποκλίσεις έχουν μέγιστη τιμή 34%

και ελάχιστη 5%. Γενικώς, παρατηρούμε ότι ο αντίκτυπος της ετερογένειας στις συναρτήσεις αυξάνεται με την αύξηση του μεγέθους της εισόδου. Επιπλέον, τα μηχανήματα w01, w02 παρουσιάζουν την καλύτερη επίδοση.

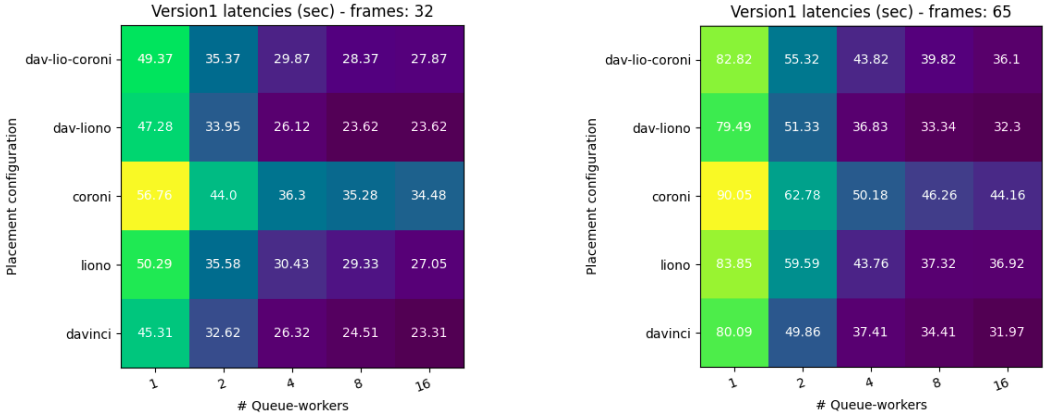


Σχήμα 14: *Αντίκτυπος της ετερογένειας στις συναρτήσεις*

Ένα χαρακτηριστικό της serverless πλατφόρμας OpenFaas [5] είναι ότι μπορεί να διαχειριστεί ασύγχρονα αιτήματα στις διαθέσιμες συναρτήσεις και άρα να επωφεληθεί από την παραλληλοποίηση της εκάστοτε εφαρμογής. Η παραλληλοποίηση μίας εφαρμογής απαιτεί την εξυπηρέτηση ασύγχρονων αιτημάτων ώστε να επικαλύπτονται χρόνοι μεταξύ των διαφόρων στιγμιοτύπων. Το OpenFaas εξυπηρετεί τα ασύγχρονα αιτήματα μέσω αύξησης των αντιγράφων του εργαλείου του που ονομάζεται Queue-Worker. Δημιουργώντας διαφορετικές τοπολογίες των Version1, Version4 εξετάσαμε με διαφορετικό αριθμό διάθεσιμων Queue-Workers την χρονική επίδοση όλων των συνδυασμών. Τα συγκεντρωτικά αποτελέσματα παρουσίαζονται στα σχήματα 15 και 16.

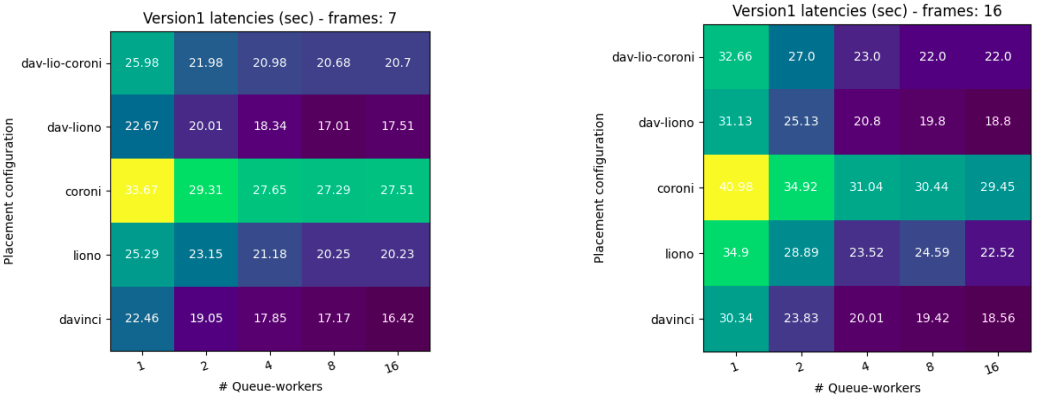
Οι τοπολογίες που σχηματίστηκαν ήταν οι εξής:

- Davinci: Όλες οι συναρτήσεις τοποθετημένες στον κόμβο w01
- Liono: Όλες οι συναρτήσεις τοποθετημένες στον κόμβο w02
- Coroni: Όλες οι συναρτήσεις τοποθετημένες στον κόμβο w03
- Davinci-Liono: Η συνάρτηση Framer τοποθετημένη στον κόμβο w01, οι υπόλοιπες στον κόμβο w02
- Davinci-Liono-Coroni: Η συνάρτηση Framer τοποθετημένη στον κόμβο w01, η Face-detector στον κόμβο w02 και οι υπόλοιπες στον κόμβο w03.



(α') *32 frames* (β') *65 frames*

Σχήμα 16: *Version1: placement configurations and number of replicas*



(α') *7 frames* (β') *16 frames*

Σχήμα 15: *Version1: placement configurations and number of replicas*

Συμπεράσματα αυτών των πειραμάτων είναι τα ακόλουθα:

- Η συγκριτική επιτάχυνση από την χρήση ασύγχρονων αιτήματων είναι μικρότερη στα μικρότερα μεγέθη εισόδου. Η μέγιστη τιμής επιτάχυνσης παρατηρήθηκε στον κόμβο w01 με την μεγαλύτερη είσοδο (65 στιγμιότυπα) όπου ήταν ίση με 60% επιτάχυνσης, ενώ η ελάχιστη ήταν ίση με 20% στην τοπολογία Davinci-Liono-Coroni με την μικρότερη είσοδο (7 στιγμιότυπα).

- Αυξάνοντας τον αριθμό από Queue-Workers, επιταχύνουμε την εκτέλεση της εφαρμογής. Όμως αυτό μπορεί να συμβεί έως ένα άνω όριο το οποίο εξαρτάται κάθε φορά από την εφαρμογή και τον βαθμό παραλληλοποιησιμότητας της. Στην δική μας περίπτωση, με 1 Queue-Worker πετυχαίνουμε καλύτερο χρόνο εκτέλεσης ίσο με 58 δευτερόλεπτα ενώ με 16 Queue-Workers ο καλύτερος χρόνος εκτέλεσης είναι μόλις 15 δευτερόλεπτα.

- Ένας Queue-Worker επιβαρύνει ελάχιστα το μηχάνημα στο οποίο βρίσκεται. Η επιβάρυνση σε μνήμη RAM δεν είναι περισσότερη από 10MB και άρα δεν επηρεάζει τους πόρους του εκάστοτε μηχανήματος.

0.4 Σχεδιασμός και Υλοποίηση Δυναμικού Δρομολογήτη

Η ενότητα αυτή αναλύει την σχεδίαση και την υλοποίηση ενός δυναμικού δρομολογητή βασισμένου σε βαθιά ενισχυτική μάθηση, με στόχο την διαχείριση μίας εφαρμογής ανάλυσης βίντεο η οποία εκτελείται σε serverless περιβάλλον νέφους. Η προτεινόμενη λύση εκμεταλλεύεται συστημικές μετρικές προκειμένου να αντιληφθεί την κατάσταση του εκάστοτε ετερογενούς μηχανήματος όσο αφορά τις ενεργές παρεμβολές, ώστε να εξυπηρετήσει ένα αίτημα χρήστη με στόχο την κανονικοποίηση του χρόνου εκτέλεσης του μέσω μετατοπίσης των συναρτήσεων σε κατάλληλους κόμβους αλλά και μείωσης/αύξησης των αντιγράφων τους. Η προσέγγιση μας καταφέρνει να ενορχηστρώσει τις συγκεκριμένες συναρτήσεις κάτω από δυναμικές συνθήκες πίεσης πόρων και αιτήματων του χρήστη.

Αρχικά θα παρουσιάσουμε συνοπτικά την τεχνολογία της βαθιάς ενισχυτικής μάθησης, ενώ στην συνέχεια θα ακολουθήσει ανάλυση για την βασική αρχιτεκτονική και τεχνική υλοποίηση του δυναμικού δρομολογητή.

0.4.1 Βαθιά Ενισχυτική Μάθηση

Η βαθιά ενισχυτική μάθηση είναι ένα είδος μηχανικής μάθησης που χρησιμοποιείται όλο και περισσότερο τα τελευταία χρόνια λόγω των λύσεων που προσφέρει σε μία ομάδα πολύπλοκων προβλημάτων όπως η όραση υπολογιστή, η επεξεργασία φυσικής γλώσσας ή η αναγνώριση μοτίβου. Η ικανότητα της βαθιάς μάθησης να καταναλώνει, μαθαίνοντας μοτίβα, μεγάλους όγκους δεδομένων καθώς και η πρόοδος σε υλικό υπολογιστή hardware η οποία πλέον μπορεί να καλύψει μεγάλες απαιτήσεις σε συπολογιστική ισχύ, σε συνδυασμό με την ενισχυτική μάθηση έχει ανοίξει νέα μονοπάτια επίλυσης σε προβλήματα που μέχρι πρότινος απαιτούσαν πολύ σύνθετες προσεγγίσεις.

Η βαθιά μάθηση είναι ένας τύπος μηχανικής μάθησης που δεν απαιτεί ακριβή ανθρώπινη προεπεξεργασία των δεδομένων που θα καταναλώσει ο υπολογιστής. Αντιθέτως, ο υπολογιστής είναι σε θέση να δεχθεί δεδομένα πολλών διαστάσεων, μη απλουστευμένα, και να εξάγει μοτίβα που χαρακτηρίζουν μοναδικά αυτές τις ομάδες δεδομένων. Έτσι εξυπηρετείται ικανοποιητικά ο στόχος της τεχνητής νοημοσύνης που είναι η κατανόηση του κόσμου με την ελάχιστη δυνατή ανθρώπινη συμμετοχή. Η αρχιτεκτονική των δικτύων βαθιάς μάθησης αποτελείται από πολλαπλά επίπεδα αυτόματων κωδικοποιητών (RBM) και στρώσεις συνέλιξης. Τα δίκτυα δέχονται στην είσοδο τους μεγάλες ποσότητες δεδομένων, τα οποία επεξεργάζονται σειριακά σε κάθε επίπεδο. Η έξοδος ενός επιπέδου αποτελεί την είσοδο του επόμενου και είναι συνήθως μη γραμμικοί συνδυασμοί της εισόδου τους. Προκειμένου η τελική έξοδος του δικτύου να λύνει επιτυχώς το πρόβλημα που ζητείται είναι αναγκαίο το δίκτυο να έχεις τις κατάλληλες τιμές στο σύνολο των υπερπαραμέτρων του. Η διαδικασια εύρεσης κατάλληλων τιμών για τις υπερπαραμέτρους του δικτύου είναι συχνά μία αργή διαδικασία που απαιτεί εξοικίωση με τους διάφορους τύπους αρχικτεκτονικών και προβλημάτων που λύνει η καθεμία.

Η ενισχυτική μάθηση είναι ένας τύπος μηχανικής μάθησης όπου ο υπολογιστής μαθαίνει να λύνει ένα πρόβλημα αλληλεπιδρώντας με το περιβάλλον του προβλήματος που καλείται να επιλύσει. Η βασική μέθοδος απόκτησης γνώσης στην ενισχυτική μάθηση είναι η διαδικασία

δοκιμής και σφάλματος. Συγκεκριμένα, ο υπολογιστής δοκιμάζει λύσεις στο πρόβλημα και αναλόγως από την έκβαση της κατάστασης, το περιβάλλον του προσφέρει μία θετική ή αρνήτικη επιβράβευση. Στόχος του πράκτορα (εναλλακτική ονομασία του υπολογιστή στα προβλήματα ενισχυτικής μάθησης) είναι σε βάθος χρόνου να συγκεντρώσει συσσωρευτικά όσο γίνεται μεγαλύτερη θετική επιβράβευση.

0.4.2 Αρχιτεκτονική και Υλοποίηση Δυναμικού Δρομολογήτη

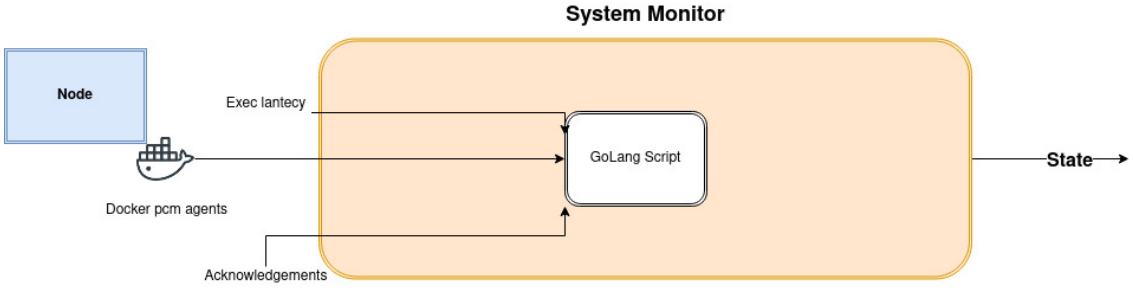
Στα πλαίσια έρευνας της συγκεκριμένης διπλωματικής εργασίας αναπτύξαμε ένα μοντέλο δυναμικού δρομολογητή βασισμένο σε βαθιά ενισχυτική μάθηση. Σχετικά με την στρατηγική που ακολουθήσαμε, ξεχωρίζουν τα εξής σημεία:

- Εκ των προτέρων γνώση του υπολογιστικού προφίλ της εφαρμογής δεν απαιτείται. Με άλλα λόγια, δεν χρειάζεται έξτρα χρόνος για μελέτης της εφαρμογής που πρόκεται να αναλάβει ο δρομολογητής για δυναμική δρομολόγηση, αλλά ο πράκτορας θα μάθει την συμπεριφορά της εφαρμογής κατά την διάρκεια της εκπαίδευσης του. Αυτή είναι μία ρεαλιστική προσέγγιση προβλήματος για serverless εφαρμογές που εκτελούνται στο νέφος.
- Οι πιθανές δρομολογήσεις που μπορεί να εφαρμόσει ο δρομολογητής είναι διακριτές, πεπερασμένες και εξαρτώνται από τις υποδομές του συστήματος πάνω στο οποίο εκτελείται η δρομολόγηση.
- Όταν οι συνθήκες του συστήματος αλλάζουν, ο δρομολογητής είναι σε θέση να προσαρμοστεί σ' αυτές αντιστοίχως ώστε να εξυπηρετήσει όσο το δυνατόν το όποιο αίτημα του χρήστη.

Αρχιτεκτονική Δρομολογήτη

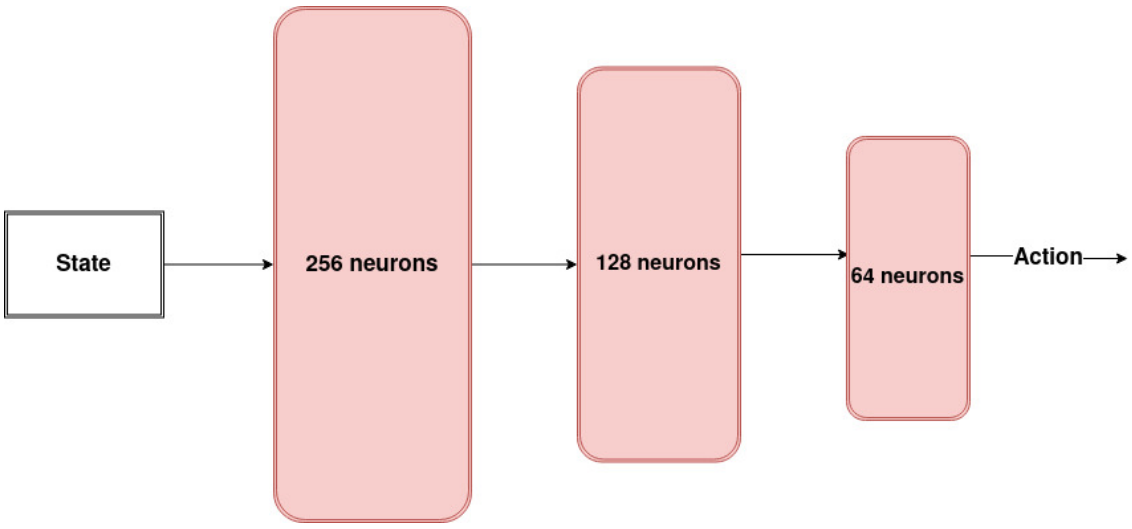
Η δομή του δυναμικού δρομολογητή αποτελείται από:

- **System Monitor**, το οποίο συλλέγει μετρικές που συνθέτουν την κατάσταση του συστήματος. Η συλλογή των μετρικών περιλαμβάνει PCM metrics, τον συνολικό χρόνο εκτέλεσης της εφαρμογής, την τοπολογία των συναρτήσεων στο σύστημα και άλλα μηνύματα επιβεβαίωσης για διάφορες διεργασίες. Τα PCM metrics συγκεντρώνονται από κάθε κόμβο με χρήση Docker πακέτων, τα οποία με την σειρά τους προωθούνται στο system monitor. Σχηματική απεικόνιση προσφέρεται στο σχήμα 17.



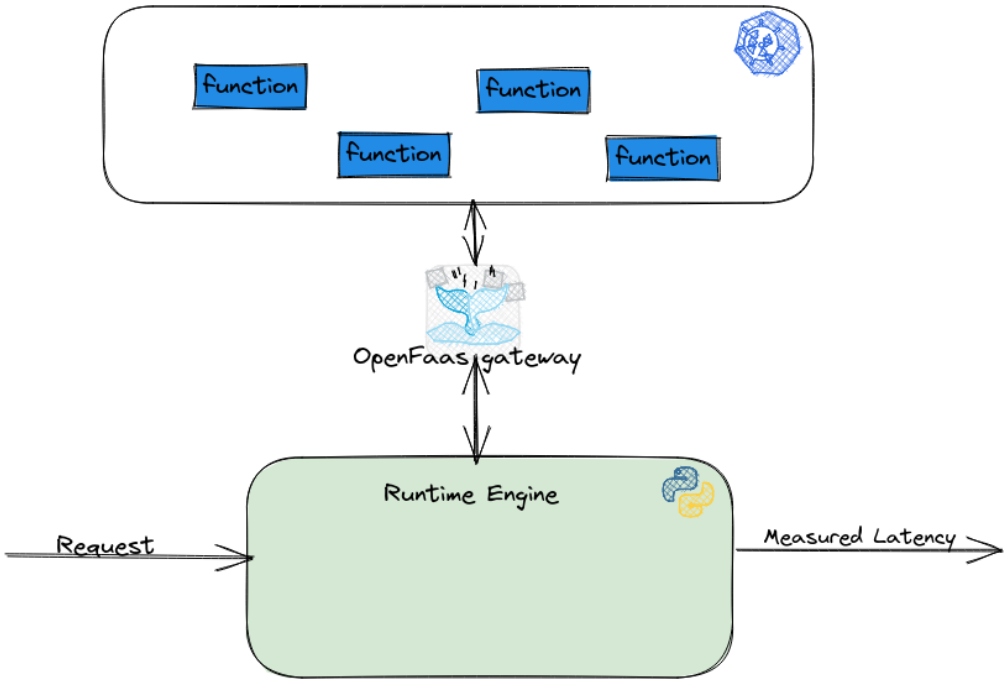
Σχήμα 17: *Αρχιτεκτονική του System Monitor*

- **DRL-based agent**, ο οποίος είναι ένας πράκτορας βαθιάς ενισχυτικής μάθησης που προσπαθεί να μάθει τις βέλτιστες αποφάσεις ώστε να αυξήσει την επιβράβευση που λαμβάνει αλληλεπιδρώντας με το περιβάλλον του. Ο πράκτορας χρησιμοποιεί το δίλημμα εξερεύνησης-εκμετάλλευσης το οποίο του επιτρέπει όχι μόνο να εκμεταλλευτεί την έως τότε βέλτιστη απόφαση αλλά να εξερευνήσει κι άλλες αποφάσεις που μπορεί να αποβούν καλύτερες μακροπρόθεσμα. Ο παράγοντας με τον οποίο ο πράκτορας επιλέγει να εκμεταλλευτεί μία απόφαση ή να εξερευνήσει μία άλλη, μειώνεται κατά την διάρκεια της εκπαίδευσης με προκαθορισμένο ρυθμό. Αναλόγως του αποτέλεσματος της εκάστοτε απόφασης του πράκτορα, το DRL-based agent λαμβάνει μία επιβράβευση η οποία μπορεί να είναι θετική, αν εξυπηρετήθηκε επιτυχώς το αίτημα του χρήστη, ή αρνητική αν δεν εξυπηρετήθηκε το αίτημα. Η ακριβή τιμή της επιβράβευσης καθορίζεται από την σύναρτηση επιβράβευσης που παρουσιάζεται αναλυτικά εδώ. Σχηματική απεικόνιση προσφέρεται στο σχήμα 18.



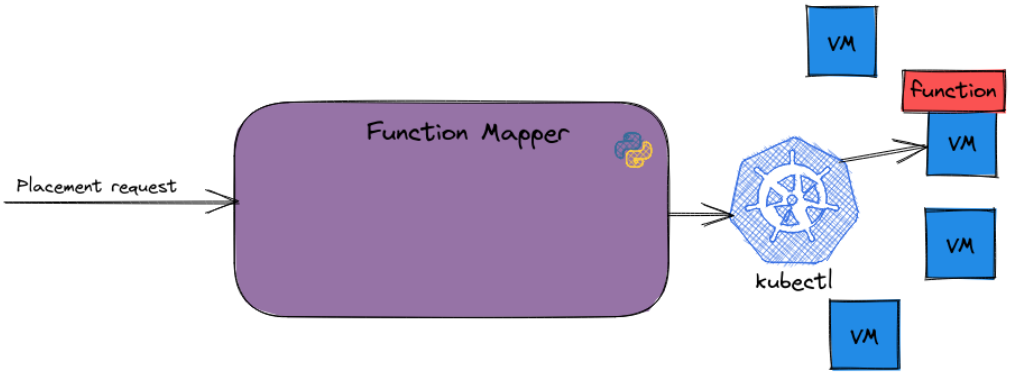
Σχήμα 18: *Αρχιτεκτονική του DRL-based agent*

- **Runtime Engine**, όπου δεδομένης της τοπολογίας των συναρτήσεων την εκάστοτε στιγμή αναλαμβάνει να εκτελέσει μία επανάληψη της εφαρμογής. Κατά την φάση της εκπαίδευσης του DRL-based agent, ο συνολικός χρόνος εκτέλεσης της εφαρμογής προσφέρεται στο νευρωνικό δίκτυο ώστε να υπολογιστεί η ανάλογη επιβράβευση. Περισσότερες πληροφορίες για την υλοποίηση της runtime engine βρίσκονται εδώ. Σχηματική απεικόνιση προσφέρεται στο σχήμα 19.



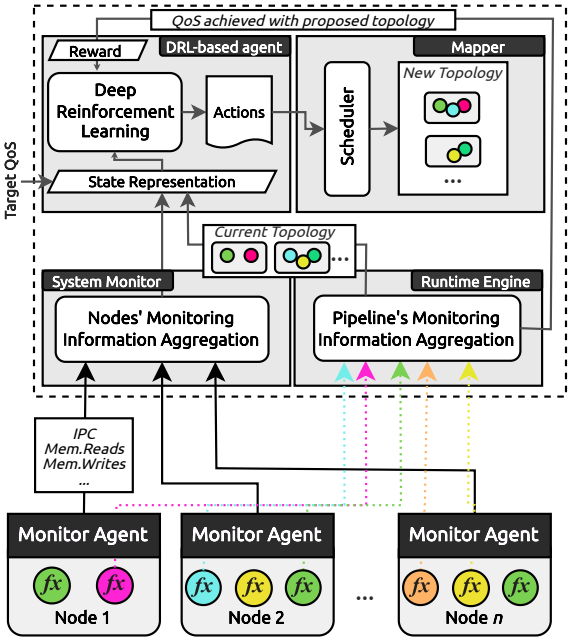
Σχήμα 19: *Overview of the Runtime Engine*

- **Function Mapper**, όπου βάση της απόφασης του νευρωνικού δικτύου DRL-based agent τοποθετεί την ζητούμενη συνάρτηση στον επιλεγμένο κόμβο του συστήματος. Σχηματική απεικόνιση προσφέρεται στο σχήμα 20.



Σχήμα 20: *Αρχιτεκτονική του Function Mapper*

Τα προαναφερθέντα στοιχεία απεικονίζονται συνολικά στο σχήμα 21.



Σχήμα 21: *Ολιστική Αρχιτεκτονική του Δυναμικού Δρομολογήτη*

Τεχνική Υλοποίηση Δρομολογήτη

Αρχικά για την τεχνική υλοποίηση των δρομολογητών απαιτούνταν **κρίσιμες μετρήσεις**. Συγκεκριμένα, η μέτρηση του συνολικού χρόνου εκτέλεσης μίας επανάληψης της εφαρμογής γίνεται με την runtime-engine. Το σφάλμα στην μέτρηση αυτή είναι το πολύ 400 milliseconds. Για τις μετρικές που αναπαριστούν την κατάσταση του συστήματος κάθε δεδομένη χρονική στιγμή (PCM metrics) χρησιμοποιείται το system monitor.

Στην συνέχεια, προκειμένου ο πράκτορας του νευρωνικού δικτύου να συγκλίνει σε σωστές αποφάσεις που εξυπηρετούν βέλτιστα τα αιτήματα του χρήστη απαιτείται **ρύθμιση των διαφόρων υπερπαραμέτρων** του δικτύου. Ψστερα από πειραματισμούς σε διαφόρα επίπεδα καταλήξαμε στις εξής τιμές:

- Μέγεθος minibatch = 32
- Βελτιστοποιητής Αδαμ με ρυθμό εκμάθησης 0.0025
- Discount factor = 0.99
- Παράγοντας εξερεύνησης, αρχική τιμή = 1
- Παράγοντας εξερεύνησης, τελική τιμή = 0.01
- Τρία επίπεδα κόμβων στο νευρωνικό δίκτυο με μεγέθη 256, 128, 64 αντιστοίχως
- Συνάρτηση ενεργοποίησης ReLU
- Μέγεθος καταχωρητή = 10^6

0.5 Αποτελέσματα και Αξιολόγηση

Σε αυτή την ενότητα παρουσιάζουμε αρχικά τις πειραματικές συνθήκες κάτω από τις οποίες μελετήσαμε τον προτεινόμενο δυναμικό δρομολογητή καθώς και τα κριτήρια αξιολόγησης του, στην συνέχεια αναλύουμε τους τέσσερις δρομολογητές που ανάπτυξαμε και κατόπιν ακολουθεί η συγκριτική τους αξιολόγηση.

0.5.1 Πειραματικές Συνθήκες

Κάθε δυναμικός δρομολογητής που αναπτύχθηκε για την συγκεκριμένη διπλωματική εργασία τοποθετήθηκε προς λειτουργία στον κόμβο Αφέντη (Master) ώστε να μην προσθέτει θόρυβο στους κόμβους Εργάτες, οι οποίοι αναλαμβάνουν να εκτελέσουν όλο το υπολογιστικό φορτίο. Οι δρομολογητές μοιράζονται τον ίδιο στόχο: να τοποθετούν τις serverless συναρτήσεις στους κατάλληλους κόμβους για να εξυπηρετήσουν αναλόγως το αίτημα του χρήστη. Ως επίπεδα εξυπηρέτησης επιλέξαμε το 35 και το 26, τα οποία είναι ουσιαστικά ο ανώτερος ανεκτός χρόνος εξυπηρέτησης της εφαρμογής. Η συγκεκριμένη επιλογή προέκυψε από ανάλυσης της εφαρμογής σε πρώτο χρόνο, όπου βρέθηκε ότι τα συγκεκριμένα επίπεδα εξυπηρέτησης μπορούν να επιτευχθούν με συγκεκριμένες τοπολογίες των συναρτήσεων στους κόμβους. Στόχος είναι οι δυναμικοί δρομολογητές να ανακαλύψουν γρήγορα τις συγκεκριμένες τοπολογίες και να συγκλίνουν σ' αυτές.

Προκειμένου να δημιουργήσουμε πολυπλοκότερες συνθήκες για το πρόβλημα, απαιτούμε κάθε επίπεδο εξυπηρέτησης να επιτευχθεί με και χωρίς την παρουσία παρεμβολών στους κόμβους του συστήματος. Η ανάγκη προσομείωσης των παρεμβολών χρησιμοποιήσαμε μία εφαρμογή iBench η οποία δεσμεύει το επιθυμητό κάθε φορά ποσοστό των πυρήνων (CPU) του εκάστοτε μηχανήματος. Η προσομείωση έγινε με συνολικά τέσσερα επίπεδα παρεμβολών: 0%, 25%, 50%, 75%. Σημαντικό να σημειωθεί ότι λόγω της ετερογένειας των κόμβων του συστήματος, οι κόμβοι έχουν διαφορετικό αριθμό συνολικών πυρήνων. Για παράδειγμα, το 50% επίπεδο παρεμβολών στον κόμβο Davinci δεσμεύει 4 από τους 8 πυρήνες, ενώ στον κόμβο Coroni δεσμεύει 8 από τους 16 πυρήνες του.

Κατά την εκπαίδεση των δρομολογητών είχαμε τις εξής συνθήκες:

$$QoS = \left\{ \begin{array}{ll} 35seconds, & 0 \leq trainingsteps \leq 300 \\ 26seconds, & \text{if } 300 \leq trainingsteps \leq 500 \end{array} \right\} \tag{1}$$

$$CPUpressure = \left\{ \begin{array}{ll} 25\%PM1, 50\%PM4, & \text{if } 0 \leq trainingsteps \leq 100 \\ 50\%PM1, 25\%PM4, & \text{if } 100 \leq trainingsteps \leq 200 \\ 50\%PM2, 25\%PM3, & \text{if } 200 \leq trainingsteps \leq 300 \\ 0\%, & \text{if } 300 \leq trainingsteps \leq 400 \\ 50\%PM1, 25\%PM2, 75\%PM3 & \text{if } 400 \leq trainingsteps \leq 500 \end{array} \right\} \tag{2}$$

Μετά την εκπαίδεση των δρομολογητών για περαιτέρω αξιολόγηση δημιουργήσαμε τις εξής συνθήκες:

$$QoS = \left\{ \begin{array}{ll} 35seconds, & \text{if } 0 \leq trainingsteps \leq 150 \\ 26seconds, & \text{if } 150 \leq trainingsteps \leq 250 \end{array} \right\} \quad (3)$$

$$CPUpressure = \left\{ \begin{array}{ll} 25\%PM1, 50\%PM4, & \text{if } 0 \leq trainingsteps \leq 50 \\ 50\%PM1, 25\%PM4, & \text{if } 50 \leq trainingsteps \leq 100 \\ 50\%PM2, 25\%PM3, & \text{if } 100 \leq trainingsteps \leq 150 \\ 0\%, & \text{if } 150 \leq trainingsteps \leq 200 \\ 50\%PM1, 25\%PM2, 75\%PM3 & \text{if } 200 \leq trainingsteps \leq 250 \end{array} \right\} \quad (4)$$

0.5.2 Κριτήρια Αξιολόγησης

Αξιολογούμε την προτεινόμενη λύση μας, δηλαδή την επίδοση του δυναμικού δρομολογητή να στο να ενορχηστρώνει την εκτέλεση, μεταφορά και κλιμάκωση των συναρτήσεων ώστε να εξυπηρέτησει ένα αίτημα χρήστη (μη ξεπερνώντας το χρονικό όριο που έχει θέσει ο χρήστης). Η προς έρευνα μετρική που μας ενδιαφέρει είναι το κλάσμα πραγματικού χρόνου εκτέλεσης προς το άνω χρονικό όριο που έχει θέσει ο χρήστης. Ένα κλάσμα μικρότερο ή ίσο της μονάδας, υποδικνύει την επιτυχή εξυπηρέτηση του αιτήματος του χρήστη ενώ μία τιμή κλάσματος μεγαλύτερη της μονάδας υποδικνύει παραβίαση του χρονικού ορίου του χρήστη και άρα αποτυχία στην εξυπηρέτηση του αιτήματος του χρήστη. Επιπλέον, όσο πιο κοντά στην μονάδα βρίσκεται η τιμή του κλάσματος μιας επιτυχημένης εξυπηρέτησης τόσο καλύτερη χρήση των πόρων θεωρούμε ότι πετύχαμε και έτσι αξιοποιήσαμε τους ελάχιστους δυνατούς από τους διαθέσιμους υπολογιστικούς πόρους.

0.5.3 Παρουσίαση των Δρομολογητών

Fullmap Δυναμικός Δρομολογητής

Ο Fullmap δυναμικός δρομολογητής είναι μία λύση ειδική για την υποδομή του συστήματος πάνω στην οποία καλείται να δουλέψει. Συγκεκριμένα, ο πράκτορας έχει ένα χώρο δράσεων διάστασης 15, ο οποίος περιλαμβάνει δώδεκα δράσεις που τοποθετούν κάθε συνάρτηση σε όλους τους διαθέσιμους κόμβους του συστήματος, δύο δράσεις που αυξάνουν και μειώνουν τα αντίγραφα των συναρτήσεων faceanalyzer, mobilenet και μία δράση για διατήρηση της υπάρχουσας τοπολογίας.

Custom Δυναμικός Δρομολογητής

Ο Custom δυναμικός δρομολογητής είναι μία ειδικά σχεδιασμένη λύση για την επιλεγμένη εφαρμογή και προέκυψε αποκτώντας καλή διαίσθηση γύρω από την συμπεριφορά της εφαρμογής κάτω από διάφορες συνθήκες. Συγκεκριμένα, ο πράκτορας διαθέτει ένα χώρο δράσεων διάστασης 12, ο οποίος περιλαμβάνει 8 δράσεις για τοποθέτηση των συναρτήσεων framer, facedetector σ' όλους τους διαθέσιμους κόμβους, μία δράση που μετακινεί τις συναρτήσεις faceanalyzer, mobilenet στον κόμβο με την μικρότερη τιμή C0, δύο δράσεις που αυξάνουν ή μειώνουν τα αντίγραφα των συναρτήσεων faceanalyzer, mobilenet και μία δράση για διατήρηση της υπάρχουσας τοπολογίας.

Kubernetes Δυναμικός Δρομολογητής

Ο Kubernetes δυναμικός δρομολογητής είναι μία προσέγγιση όπου τον κόμβο στόχο μίας μετακίνησης μίας serverless συνάρτησης τον επιλέγει ο δρομολογήτης του Κυβερνήτη του συστήματος. Πιο συγκεκριμένα, αυτός ο δρομολογητής έχει συνολικά 4 διαθέσιμες δράσεις: μία για μετακίνηση της συνάρτησης framer, μία για μετακίνηση της συνάρτησης facedetector, μία για μετακίνηση των faceanalyzer, mobilenet και μία για διατήρηση της υπάρχουσας τοπολογίας. Ο δυναμικός δρομολογητής ουσιαστικά διαλέγει ποια συνάρτηση θα μετακινηθεί αλλά ο δρομολογητής του Κυβερνήτη επιλέγει τον κόμβο στόχο της συγκεκριμένης μετακίνησης.

Oracle Δυναμικός Δρομολογητής

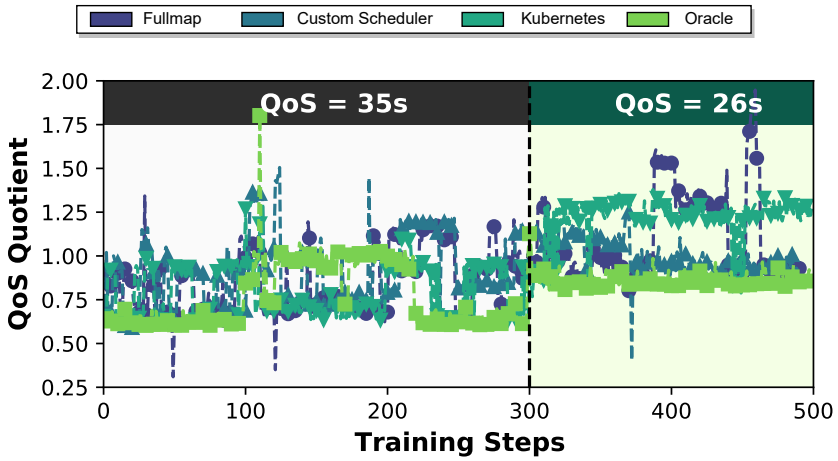
Ο τελευταίος δρομολογητής που αναπτύχθηκε βασίζεται σε oracle παρατηρήσεις. Μία oracle παρατήρηση παρέχει πληροφορία για μία τοπολογία που έχει ληφθεί σε πρώτο χρόνο, πριν την εκπαίδευση των δρομολογητών. Στα πρώτα στάδια της συγκεκριμένης διπλωματικής εργασίας μελετήσαμε την συμπεριφορά της εφαρμογής κάτω από διάφορες συνθήκες όπως έχει προηγουμένως αναφερθέι και έτσι συλλέχθηκε ένα σύνολο από oracle παρατηρήσεις. Επομένως, ο δρομολογητής θα έχει την ευθύνη να υποδικνύει την συνάρτηση που πρέπει να μετακινηθεί αλλά τον κόμβο στόχος της μετακίνησης θα τον επιλέγει μία oracle παρατήρηση. Ο πράκτορας του τελευταίου δρομολογητή διαθέτει τέσσερις δράσεις όπου μία για μετακίνηση της συνάρτησης framer, μία για μετακίνηση της συνάρτησης facedetector, μία για μετακίνηση των faceanalyzer, mobilenet και μία για διατήρηση της υπάρχουσας τοπολογίας.

0.5.4 Συγκριτική Αξιολόγηση των Δρομολογητών

Εδώ παρουσιάζουμε την συγκριτική αξιολόγηση των τεσσάρων δυναμικών δρομολογητών που αναπτύξαμε. Η σύγκριση πραγματοποιείται βάση των εξής παραγόντων: 1) Κλάσμα χρόνου εκτέλεσης προς χρονικό όριο χρήστη 2) Συσσωρευμένη επιβράβευση των πρακτόρων 3) Παραβιάσεις στο χρονικό όριο χρήστη 4) Απαιτούμενος χρόνος για σύγκλιση και 5) Κλιμάκωση

Κλάσμα χρόνου εκτέλεσης προς χρονικό όριο χρήστη

Όπως φαίνεται στην εικόνα 22, εκτός του Oracle δρομολογητή, ο Custom δρομολογητής παρουσίαζει αξιόλογη σταθερότητα υπό τις μεταβαλλόμενες συνθήκες στα επίπεδη πίεσης των πόρων και την αλλαγή του άνω χρονικού ορίου του χρήστη σε μία πιο ¨αυστήρη¨ τιμή, αφού καταφέρνει να προσαρμοστεί σχετικά άμεσα πετυχαίνοντας μία τιμή κλάσματος κοντά στην μονάδα. Μια παρόμοια, αλλά σχετικά λιγότερο σταθερή συμπεριφορά είχε και ο Full-map δρομολογητής, κάτι το οποίο μπορεί να οφείλεται στο μεγαλύτερο πλήθος διαθέσιμων ενεργείων που κατέχει ο πράκτορας του συγκεκριμένου δρομολογήτη. Τέλος, ο Kubernetes δρομολογητής αν και στην αρχή φαίνεται να έχει μία αξιόλογη επίδοση, στην συνέχεια λόγω της άγνοιας του για την ετερογένεια και τις ενεργές παρεμβολές του συστήματος αποτυγχάνει να πετύχει τον στόχο του.

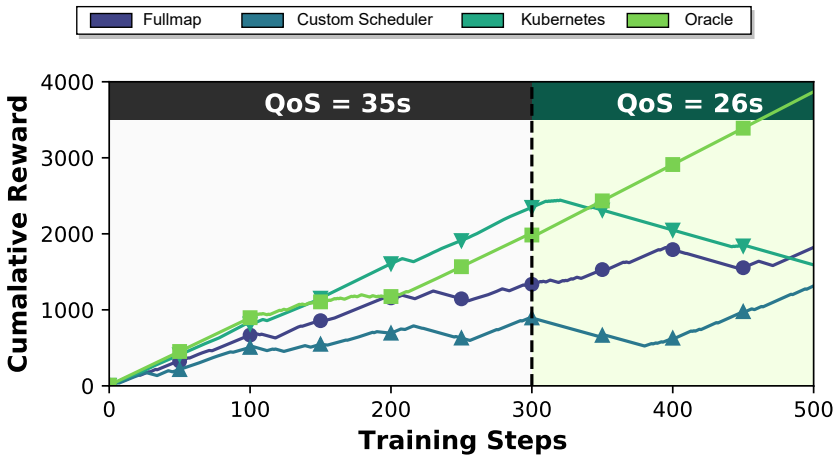


(α')

Σχήμα 22: *Κλάσμα χρόνου εκτέλεσης προς χρονικό όριο χρήστη κατά την εκπαίδεση*

Συσσωρευμένη επιβράβευση των πρακτόρων

Η επιβράβευση που λαμβάνει ο πράκτορας κάθε δρομολογητή αποκαλύπτει την ικανότητα του να ανακαλύπτει μία πιο αποδοτική τοπολογία των συναρτήσεων προκειμένου εξυπηρέτησει το αίτημα του χρήστη. Η εικόνα 23 δείχνει τις επιβραβεύσεις που λαμβάνουν οι πράκτορες κατά την εκπαίδεση τους. Και πάλι, ο Oracle δρομολογητής είναι ο πιο κυρίαρχος. Ο Fullmap δρομολογητής, όπως φαίνεται, λαμβάνει με το περασμά του χρόνου μεγαλύτερη επιβράβευση από τον Custom αλλά ο Custom έχει μία πιο ¨απότομη¨ θετική κλίση στα τελευταία στάδια που υποδικνύει μία πιο επαρκή γνώση του προβλήματος που καλείται να λύσει, κάτι που στο μέλλον μπορεί να μεταφραστέι σε καλύτερη επίδοση. Τέλος, ο Kubernetes δρομολογητής λόγω της ανικανότητας του να προσαρμοστεί στην αλλαγή του άνω χρονικού ορίου του χρήστη όπως αναφέρθηκε παραπάνω λαμβάνει αρνητικές επιβραβεύσεις στην δεύτερη φάση της εκπαίδευσης.

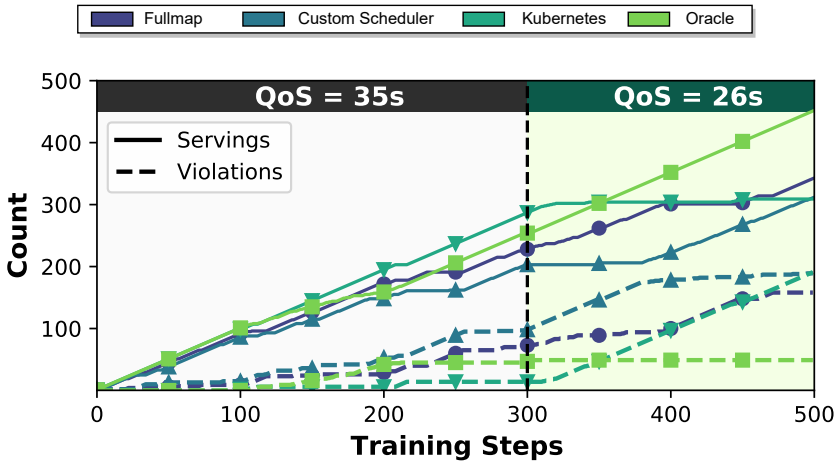


(α')

Σχήμα 23: *Συσσωρευμένη επιβράβευση των πρακτόρων κατά την εκπαίδεση*

Παραβιάσεις στο χρονικό όριο χρήστη

Στην εικόνα 24 παρουσίαζονται οι παραβιάσεις των δρομολογητών στα αιτήματα του χρήστη. Οι Fullmap, Custom δρομολογητές αποδεικνύουν την ικανότητα της βαθιάς ενισχυτικής μάθησης να λύνει το πρόβλημα της δυναμικής δρομολόγησης, αφού οι επιτυχείς εξυπηρετήσεις είναι περισσότερες από τις ανεπιτυχείς με μεγάλη διαφορά. Ο Oracle δρομολογητής με μεγάλη άνεση εξυπηρετεί τα αιτήματα αποδεικνύοντας την ικανότητα της βαθιάς ενισχυτικής μάθησης να συνεργαστεί με παρατηρήσεις Oracle. Τέλος, ο Kubernetes δρομολογητής έχει την χειρότερη επίδοση από όλους.



(α')

Σχήμα 24: *Παραβιάσεις στο χρονικό όριο χρήστη*

Απαιτούμενος χρόνος για σύγκλιση

Όσο αφορά τον απαιτούμενο χρόνο για σύγκλιση, ο Oracle δρομολογητής είναι ο ταχύτερος. Στην συνέχεια έρχονται οι Fullmap, Custom δρομολογητές με σχετικά παρόμοια ταχύτητα σύγκλισης ενώ τελευταίος με φανερά σημάδια αστάθειας είναι ο Kubernetes δρομολογητής όπως φαίνεται και στην εικόνα 22.

Κλιμάκωση

Το θέμα της κλιμάκωσης είναι ιδιαίτερα σημαντικό για όλες τις προσεγγίσεις και πρέπει να λαμβάνεται υπόψιν πριν να εφαρμοστεί η προτεινόμενη προσέγγιση της συγκεκριμένης διπλωματικής σε διαφορετικές τοπολογίες συστημάτων ή σε διαφορετική εφαρμογή. Οι Fullmap, Custom δρομολογητές απαιτούν προσαρμογή των διαθέσιμων ενεργειών των πρακτόρων τους αν αλλάξουν οι υποδομές του συστήματος ή η εφαρμογή προς ενορχήστρωση. Αντιθέτως, ο Kubernetes δρομολογητής δεν χρειάζεται κάποια αλλαγή λόγω της αγνωστικιστικής του φύσης, ενώ ο Oracle απαιτεί μία μικρή αλλαγή αν πρόκειται να ενορχηστρωθεί μια διαφορετική εφαρμογή από αυτή που εξετάστηκε στην συγκεκριμένη διπλωματική εργασία.

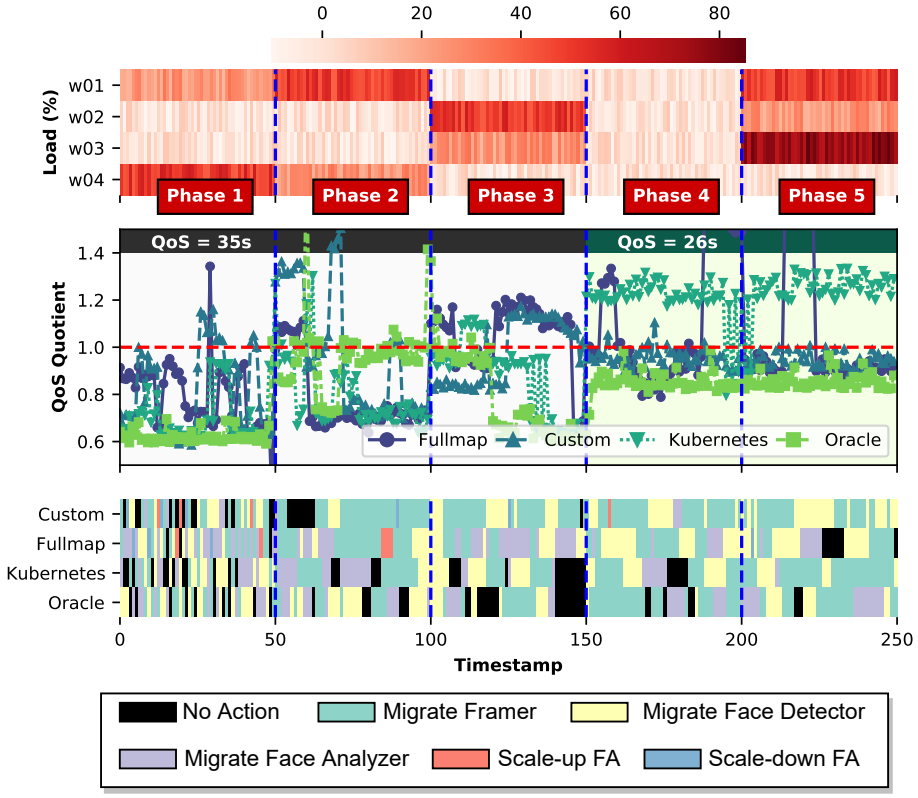
Δυναμικός δρομολογητής έναντι δρομολογήτη του Κυβερνήτη

Τέλος, συγκρίνουμε την προσέγγιση δυναμικού δρομολογητή βασισμένου σε βαθιά ενισχυτική μάθηση με μια αφελή προσέγγιση ενορχήστρωσης αποκλειστικά από τον δρομολογητή του Κυβερνήτη χωρίς καμία αλληλεπίδραση με δρομολογητή-πράκτορα. Εκτελώντας την ίδια εφαρμογή ανάλυσης βίντεο και κάτω από τις ίδιες μεταβαλλόμενες συνθήκες, η αφελής προσέγγιση κατάφερε να εξυπηρέτησει μόνο το 34% των αιτήματων με ¨χαλαρό¨ άνω όριο χρήστη (35 δευτερόλεπτα) ενώ δεν μπόρεσε να εξυπηρέτησει κανένα αίτημα με το ¨αυστηρότερο¨ άνω επιτρεπόμενο όριο χρήστη (26 δευτερόλεπτα). Αντιθέτως, η δική μας προσέγγιση πετυχαίνει την εξυπηρέτηση 95% των περιπτώσεων για το ¨χαλαρό¨ όριο και 75% των περιπτώσεων για το ¨αυστήρο¨ όριο κατά μέσο όρο αντιστοίχως.

0.5.5 Σύνοψη Αξιολόγησης

Όπως δείχθηκε προηγουμένως, η τελική επιβράβευση που λαμβάνει ο DRL-based agent εξαρτάται έντονα από την εκάστοτε πολιτική δρομολόγησης. Οπότε προκύπτει η εξής ερώτηση: *Αλλάζει ο DRL-based agent τις αποφάσεις αν αλλάζει η εσωτερική πολιτική δρομολόγησης·* Για να αξιολογήσουμε την ¨ευφυία¨ της βαθιάς ενισχυτικής μάθησης, ¨παγώνουμε¨ τις παραμέτρους του νευρωνικού δικτύου DQN και μελετάμε τον χώρο ενεργειών για κάθε διαφορετικό δρομολογητή. Εξερευνούμε διαφορετικές φάσεις, όπου η κάθεμια χαρακτηρίζεται από διαφορετικά επίπεδα παρεμβολών και άνω χρονικών ορίων χρήστη (QoS). Η εικόνα 25 δείχνει τα αποτελέσματα που λάβαμε. Συγκεκριμένα, το πρώτο γράφημα αναπαριστά τα επίπεδα παρεμβολών ανά κόμβο, ενώ το δεύτερο και το τρίτο αποτυπώνουν το κλάσμα άνω χρονικού ορίου χρήστη και τις επιλεγμένες ενέργειες των πρακτόρων αντιστοίχως. Έτσι προκύπτουν τα εξής αποτέλεσματα: i) Η πρώτη φάση χαρακτηρίζεται από μεγάλη ποικιλία στον χώρο των ενεργειών, μιας και κανένας από τους δρομολογητές δεν είναι σε θέση να εξυπηρετήσει επιτυχώς το ζητούμενο QoS ii) Στην δεύτερη φάση, ο Oracle δρομολογητής είναι ο μόνος ικανός να πετύχει το ζητούμενο QoS, συγκριτικά με τους υπόλοιπους δρομολογητές που επιλέγουν μόνο να μετατοπίσουν την συνάρτηση Framer, ο Oracle δρομολογητής πετυχαίνει το ζητούμενο μετατοπίζοντας την συνάρτηση Facedetector παρ' ότι η Framer είναι η συνάρτηση με την μεγαλύτερη καθυστέρηση. iii) Τέλος, ακόμα και σε περιπτώσεις χαμηλών παρεμβολών όπως στην φάση 4 ή σε περιπτώσεις όπου οι δρομολογητές επιλέγουν τις ίδιες ενέργειες όπως την φάση 5, ο Kubernetes δρομολογητής αδυνατεί να πετύχει το ζητούμενο QoS λόγω της άγνοιας του και για τα επίπεδα παρεμβολών στους κόμβους του συστήματος αλλά και στις διαφόρες επίδοσης μεταξύ των συναρτήσεων λόγω ετερογένειας.

Προτείνουμε ένα δυναμικό δρομολογητή βασισμένο σε βαθιά ενισχυτική μάθηση για δρομολόγηση συναρτήσεων που αποτελούν μία serverless εφαρμογή ανάλυσης βίντεο. Η λύση μας εκμεταλλεύεται μετρικές των μηχανήματων του συστήματος για να αναγνωρίσει την εκάστοστε κατάσταση που βρίσκεται το σύστημα και μπορεί να υλοποιηθεί χρησιμοποιώντας μία ποικιλία προσεγγίσεων ως προς την λήψη αποφάσεων. Η λύση μας προσαρμόζεται σε μεταβαλλόμενες συνθήκες πίεσης και καταφέρνει να εξυπηρετεί το 91.6% των αιτημάτων χρήστη.



Σχήμα 25: *Επίπεδα παρεμβολών, QoS κλάσμα και λήψη αποφάσεων από τους DRL-agents υπό διαφορετικές πολιτικές δρομολόγησης*

0.6 Σύνοψη και Μελλοντική Δουλειά

Μετρήσεις, αναλύσεις και προσεγγίσεις που περιγράφηκαν σε αυτή την διπλωματική εργασία αποτελούν ένα πρώτο βήμα στην αναζήτηση των τρόπων με τους οποίους η Βαθιά Ενισχυτική Μάθηση μπορεί να επέμβει στην λύση του προβλήματος δρομολόγησης serverless συναρτήσεων σ' ένα υπολογιστικό περιβάλλον νέφους. Στην συνέχεια παραθέτουμε δύο προτάσεις για μελλοντική δουλειά.

0.6.1 Αναγνώριση Εφικτών Άνω Χρονικών Ορίων

Η προσέγγιση μας προσπαθεί να εξυπηρετήσει ένα αίτημα χρήστη μέσω ρύθμισης του χρόνου εκτέλεσης της εφαρμογής σε χρόνο που δεν ξεπερνά το άνω όριο που έχει θέσει ο χρήστης. Όμως, συχνά τα άνω όρια που έχει θέσει ένας χρήστης δεν είναι εφικτά να εξυπηρετηθούν, π.χ. λόγο κακής εκτίμησης του χρήστη, με τους διαθέσιμους πόρους. Προτείνουμε λοιπόν την ανάπτυξη ενός εργαλείου που θα ταξινομεί τα άνω όρια χρήστη σε δύο κατηγορίες: Εφικτά και μη εφικτά. Μ' αυτό τον τρόπο μπορούμε να πετύχουμε γρηγορότερη και ποιοτικότερη εκπαίδευση στον πράκτορα της βαθιάς ενισχυτικής μάθησης.

0.6.2 Επέκταση προς Αγνωστικιστική Φύση του προτεινόμενου Εργαλείου

Το προτεινόμενο εργαλείο της συγκεκριμένης διπλωματικής εργασίας έχει σχεδιαστεί να λύνει το πρόβλημα της δρομολόγησης για την εφαρμογή ανάλυσης βίντεο που περιγράφηκε

σε προηγούμενη ενότητα. Μία σημαντική προσθήκη στην υπάρχουσα δουλειά θα ήταν μια πιθανή τροποποίηση του εργαλείου ώστε να γενικεύει την εφαρμογή που πρόκεται να ενορχηστρώσει και έτσι το φάσμα των προβλημάτων που θα είναι ικανό να λύσει να μεγαλώσει.

Κεφάλαιο 1

Introduction

The emergence of cloud computing has led to the drastical change of running workloads in on-promise server rooms to executing them into public cloud environments. This change has enabled developers to shift the responsibility of server and infrastructure management to a cloud service provider. Before that, application developers had to buy or lease dedicated server infrastructure to operate their systems. This required high advance investment in hardware, while operational costs increased from hiring specialized personnel to operate and maintain the infrastructure. In addition, increasing the computing capacity required long lead times, as the procedures of ordering, installation and configuration of the fresh hardware had to be done before it was ready to be used. Nowadays, it is possible to provision new server infrastructure within seconds using commercial cloud platforms.

With contemporary cloud computing solutions, deploying services is a low time-to-market product, and developers do not need to manage the underlying server infrastructure. Commercial cloud platforms are continually evolving to provide new service models, which would allow even more of the operational responsibility to be taken away from application developers and to be transferred to the cloud providers. The newest addition to this set of cloud services is serverless computing, which promises, as the word implies, to free the user entirely from server management.

Function-as-a-Service (FaaS) is a serverless cloud computing model, which allows the developers to deploy individual functions to the cloud. FaaS has become prevalent, owing to OS-level virtualization. Serverless computing enables a new way of building and scaling applications and services by allowing developers to break traditionally monolithic server-based applications into finer-grained cloud functions, without having to worry about their provisioning, scaling and managing.

1.1 Scope & Goal

When deploying an application to a serverless platform, from the cloud provider's point of view, there are some factors that have to be taken into consideration when trying to serve efficiently the end-user's request. Among these factors are: the *Granularity* of the application in terms of how many functions compose the workflow, *Interference* from third-party workloads that happen to be deployed the same time and lastly, *Heterogeneity*

that often characterizes the machines within a cloud environment.

The main goal of this thesis is the construction of a scheduling framework which aims to improve the deployment of a workflow above a serverless platform. Our proposed solution utilizes real time decision making on migrating and scaling serverless functions among a cluster, so as to serve successully user requests while coming in accordance with the system's constraints under dynamic conditions.

1.2 Structure of the thesis

This thesis is organized in six chapters: In Chapter 2, related work is presented, as we examine from all aspects the problem of serverless scheduling. In Chapter 3, background knowledge is offered for all technologies utilized for the completion of this thesis. In Chapter 4, we measure and analyze the impact a list of factors could have on the performance profile of a serverless application. This list includes factors such as heterogeneity, inteference and granularity. In Chapter 5, we present our developed DRL-based dynamic scheduling framework and its architectural characteristics. Chapter 6 is about evaluating our work and proving the efficacy of our proposed solution. Finally, in Chapter 7, we discuss future work and conclude the thesis.

Κεφάλαιο 2

Related work

In this chapter, we present related work that has been conducted regarding QoS-aware serverless frameworks, workload scheduling on cloud infrastructure and runtime resource allocation for serverless functions. Lastly, our approach for tackling the scheduling serverless functions problem is introduced.

2.1 QoS-aware Serverless Frameworks

The criticality of enhancing the performance of serverless workflows has been discussed in various research works [7], [8], [9, 10, 11], which achieve to address the user-defined latency requirements for a specific workload, by decreasing the functions inter-communication. Faastlane [7], executes functions of a workflow instance on separate threads of a process to minimize function interaction latency. Faastlane's objective is to minimize function interaction latency by striving to execute functions of a workflow as threads within a single process of a container instance, which eases data sharing via simple load/store instructions. Specifically, for functions that operate on sensitive data, Faastlane provides lightweight thread-level isolation. Lastly, when parsing a workflow, Faastlane exploits opportunities for parallelism by spawning new container instances to serve parallel functions of a workflow. Yet, heterogeneity or resource interference that may cause unpredictable performance variability are not taken into consideration. Respectively, in Sonic [12] it is thoroughly studied in which ways inter-function data exchange could be implemented, in terms of storage technologies, in order to save execution time and costs. Sonic is a data-passing manager whose task is to optimize application performance and cost by choosing the optimal data-passing method for each edge of a serverless workflow DAG and implementing communication-aware function placement in a multi-node environment. It is also capable of adjusting the best data-passing method on the fly while infrastructure changes are going to take place.

2.2 Workload Scheduling on Cloud Infrastructure

Much research has been conducted regarding the placement of applications on Cloud clusters[9, 13, 14]. In [14] the authors design an interference-aware scheduler for socket-level workloads placement on a pool of homogeneous physical machines. The framework

is designed mostly for batch workloads, thus it does not support service migration. Cirrus [9] improves the performance of ML training serverless workflows (time-to-accuracy) by employing several techniques to extend AWS Lambda offerings at infrastructure-level, i.e., data-prefetching, data-streaming, as well as in application-level, i.e., training algorithms redesign. Therefore, while it achieves significant performance improvement, both developer effort (for custom algorithm design), and domain-specific tuning knobs make it difficult to be generalized for serverless workflow management.

2.3 Runtime Resource Allocation for Serverless Functions

In [15], Reinforcement Learning (RL) is employed for defining the concurrency level, i.e., the per-function concurrent request allowance before auto-scaling out, taking only into account the application-level virtual resources utilization. However, while cloud is characterized by resource heterogeneity and multi-tenancy, neglecting system-level resource interference [16], may impose increased latency. SequenceClock[17] employs a PID controller for dynamic CPU quotas allocation on serverless function workflows under different system-pressure levels. Nonetheless, vertical CPU scaling was not capable of reducing latency in order to address QoS requirements.

2.4 Our Approach

In this thesis, firstly we discuss factors that influence the performance of a serverless application deployed to a cluster of nodes. Moreover, deltas between different configurations and placements are highlighted as we try to showcase the problem's complexity. After identifying the inefficacy of logic-based rules to schedule a serverless workflow under dynamic cluster conditions, we employ a deep reinforcement learning approach to tackle the problem of dynamic serverless scheduling. We focus on DRL-based dynamic scheduling and scaling of functions for video-analytics serverless workflows in order to meet end-to-end latency constraints. We differentiate from prior art, in the following points: i) we utilize low-level system metrics monitoring to capture resource interference [15], ii) we consider workflow composition and node heterogeneity as model parameters[7], iii) adapt the decisions both to fluctuating system-level resource pressure, as well as to variable QoS requirements that change over time[11, 12, 14], and iv) we support not only service scaling, but also service migration to another node when needed[17]

Κεφάλαιο 3

Background

This chapter introduces fundamental concepts for understanding cloud and serverless computing as well as the technologies which came before and paved the paths towards serverless workloads. Section 3.1, briefly discusses virtualization, containers and Kubernetes. Section 3.2, examines different cloud computing models that are predecessors of serverless computing. Furthermore, Section 3.3 is all about serverless computing and serverless platforms. Finally, in Section 3.4 & 3.5, a minimal analysis of machine learning and deep reinforcement learning is provided as it is mainly utilized in our approach.

3.1 Virtualization & Containers

Virtualization is the essense of serverless and cloud computing in general. Virtualization allows better portability of workloads and enables cloud service providers to provision new computing capacity to their customers rapidly. In this section, we are going to discuss about two types of virtualization: hypervisor-based and container-based.

3.1.1 Hypervisor-based virtualization

Virtualization has started to dominate running computing workloads in the past couple of decades, but its idea is even older. Specifically, IBM [1] had perceived and used this revolutionary idea in the 60s and 70s in their systems. Hypervisor-based virtualization offers a complete virtual machine by virtualizing all system's hardware. A full operating system can run on top of a virtual machine, and the fact is that the operating system does not need to be aware that the hardware is virtualized. Therefore, virtual machines can support any operating system.

In hypervisor-based virtualization, a hypervisor, which is also known as Virtual Machine Manager (VMM), manages virtual machines and their resources.
Hypervisors can be divided in two categories, which are listed below [2]:

- **Type 1**: Bare-metal hypervisor, which runs directly on the computing hardware. The bare-metal hypervisor provides better performance in comparison to the hosted hypervisor, because of the missing overhead generated by the host OS

- **Type 2**: Hosted hypervisor, which runs within the operating system of the host machine.

Hypervisor-based virtualization allows cloud service providers to run multiple virtual machines using only one physical computer. Virtualization offers great portability as long as the virtual machine images can easily be transferred between different hosts. Furthermore, the resources of a single machine can be shared between multiple virtual machines, which makes more efficient use of hardware. With hypervisor-based virtualization, cloud providers can abstract hardware from the user and so the user can easily provision computing capacity by simply creating new virtual machines. Last, but not least, provisioning virtual hardware is an efortless job. Users, for example, are able to increase system's memory by changing the virtal machine's configurations. These tasks would be much more laborious with the absense of virtualization.

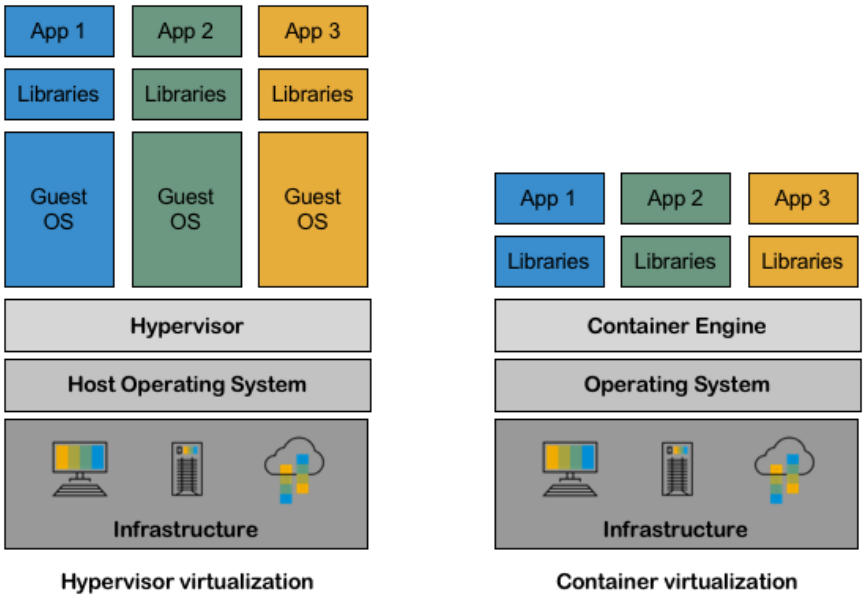
3.1.2 Container-based virtualization

In contrast to hypervisor-based virtualization which essentially runs a full OS on top of virtualized hardware, container-based virtualization, also known as operating system virtualization, functions at operating system's level avoiding hardware virtualization's overhead. The containers, which implement the isolation at the OS level, run on top of the same operating system kernel. Sharing the operating system of the underlying host machine instead of full hardware virtualization, allows more lightweight virtualization. Multiple studies have concluded that the overhead created by container-based virtualization is negligent [18]. Additionally, when a container startup happens, there is no need for a full operating system to boot, so the startup times of containers are reduced to seconds or even milliseconds. On the contrary, the time taken for a virtual machine to startup could be a minute-long period.
In Figure 3.1, the differences between hypervisor-based and container-based virtualization are shown.

Isolation and resource allocation between containers is achieved in the operating system level. For example, in Linux-based containers, these properties are implemented with the help of *cgroups* and the *namespaces*. Specifically, the Linux kernel provides the cgroups functionality that allows limitation and prioritization of resources (CPU, memory, block I/O, network, etc.) without the need of starting any virtual machine, and also the namespace isolation functionality that allows complete isolation of an application's view of the operating environment, including process trees, networking, user IDs and mounted file systems [19]. There are multiple namespaces which handle the isolation of different components:

- **Cgroup namespace:** allows isolation of cgroup root directories by virtualizing the view of a process's cgroups.

- **Inter-process communication (IPC) namespace:** provides isolation for different IPC resources like message queues, semaphores, and shared memory segments.



Σχήμα 3.1: *Hypervisor-based virtualizaton and Container-based virtualization architecture. The Guest OS's overhead is missing in the container virtualization.*

- **Network namespace:** provides isolation for networking components like network devices, protocol stacks, routing tables and firewall rules.
- **Mount namespaces:** allows mountpoint isolation between containers
- **PID namespace:** allows containers to have their own process ID space, which means that two containers can use the same process ID, because they are in different namespaces.
- **User namespace:** provides isolation for security-related identifiers and attributes like user and group IDs.
- **UTS namespace:** allows isolation of system identifiers, namely hostname and NIS domain name.

Although containers are better than virtual machines in terms of portability, resource usage and maintaince, they lose some ground because of lacking security measures. Containers provide lightweight isolation from the host operating system and containers within the same system, and as a result the host kernel is exposed to the containers, which could be an issue in multi-tenant environments. As analyzed in [20], there are a lot of concerns regarding a container's security including: i) container protection for applications inside it ii) inter-container protection iii) protecting the host from containers iv) container protection from a malicious or semi-honest host. For example, when a container is running in "privileged" mode with root access rights, its processes have nearly the same access rights as the processes running natively on the host. This problem can be soften though, by running the containers inside virtual machines.

Container-based virtualization is super essential for the serverless computing model. The applications in serverless services are usually injected into containers, which operate on top of server infrastructure that is managed by the cloud service provider.

3.1.3 Kubernetes

Kubernetes [3], also known as K8s, is an open source system for automating deployment, scaling and management of containerized applications. It was originally designed by Google, but now, the Cloud Native Computing Foundation (CNCF) maintains the project. The initial realease happened in June 7, 2014 and the source code is written in Go. The greatest thing about Kubernetes is the fact that gives you the freedom to take advantage of on-premises, hybrid, or public cloud infrastructure and makes the moving of workloads effortless. Kubernetes builds upon 15 years of experience of running production workloads at Google, combined with best-of-breed ideas and practices from the community.

When we deploy Kubernetes, we get a cluster. This cluster is consisted of Master and Worker nodes as it follows the primary/replica architecture [21]. In the following parts of this section, the Kubernetes cluster is analyzed while figure 3.2 visualizes the below mentioned architecture.

Kubernetes Master node(s) components

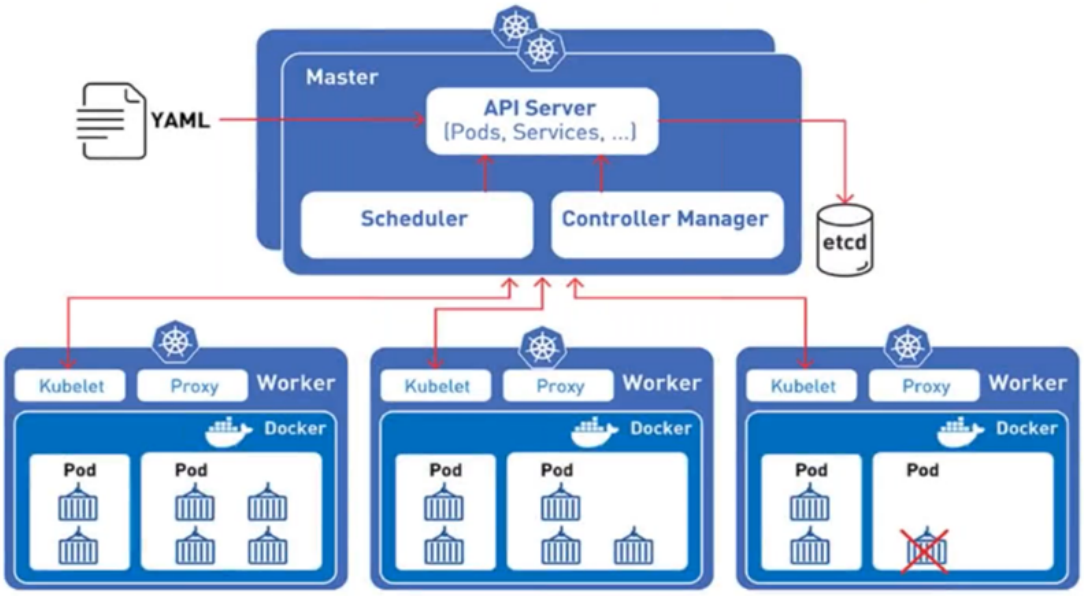
Kubernetes Master node(s) serve as the cluster's control plane. The control plane consists of various components that make global desicions about the cluster, such as scheduling, responds to cluster events as well as directs communication across the system.

- **kube-apiserver:** Component that exposes the Kubernetes API, which serves as the system's frontend.
- **etcd:** Consistent and highly-available key value store used as Kubernetes's backing store for all cluster data.
- **kube-scheduler:** Component that watches for newly created Pods with no assigned node, and selects a node for them to run on based on various factors such as resource requirements, data locality, user-system constraints.
- **kube-controller-manager:** Service that manages a set of core Kubernetes controllers, including: Replication Controller, DaemonSet Controller, Cloud Controller.

Kubernetes Worker node(s) components

Worker node(s) are the place any workload is deployed. For the deployment to happen, every worker node must run a container runtime such as Docker, as well as the below-mentioned components, that are compulsory for network communication reasons.

- **kubelet:** Agent that runs on each node of the cluster. It is responsible for running containers in a Pod.
- **kube-proxy:** Network proxy that runs on each node of the cluster, implementing part of the Kubernetes service concept. For example, it balances load across the containers in a Pod.
- **container:** Runs inside a Pod and represents the lowest abstraction level of a microservice, which holds the running application, libraries, and their dependencies.



Σχήμα 3.2: *Kubernetes components architecture*

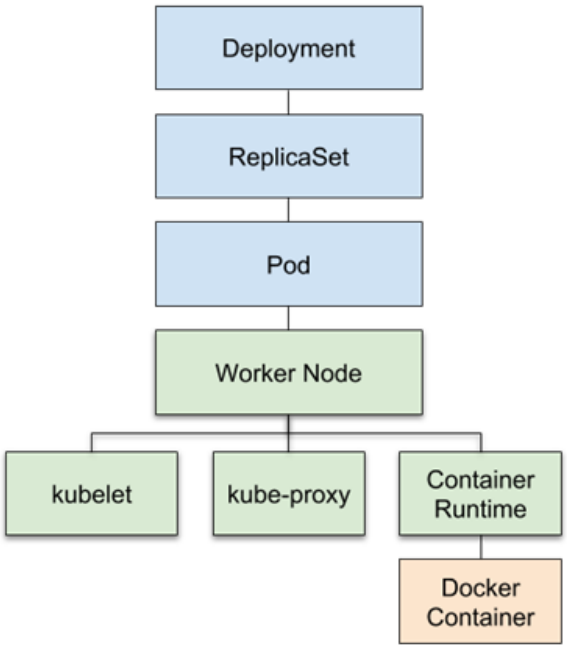
Architecture

Kubernetes architecture encapsules all the above-mentioned Master and Worker components and the ways they communicate in order to operate as a container orchestration system. The concepts and ideas upon which Kubernetes is engineered, are characterized by multiple abstraction levels. In descending order of abstraction these are: Deployment, ReplicaSet, Pod, Cluster Node, Node Process and Docker container. Deployments create and manage ReplicaSets, which create and manage Pods, which run on Cluster Nodes, which have a container runtime, which run the code the user inputs in a Docker image.

If we want to make an application that runs continuously, we need to create a Deployment which allows us to configure the application without downtime as well as to specify restarting Pods strategies. After that, the Deployment creates a ReplicaSet that will ensure our application is healthy and running as we have previously configured. ReplicaSets will create and scale Pods based on the triggers we specified in the Deployment. Pods are the building blocks upon which Kubernetes concepts operate on. A Pod contains a group of containers and other useful information that are important for handling them. Pods

live in Worker nodes, are ephemeral in nature and are subjects to restarting when they die.

In figure 3.3, the levels shaded blue are higher-level K8s abstractions and the green levels represent Nodes and Node subprocesses.



Σχήμα 3.3: *Kubernetes abstraction layers visualized.*

The six abstraction levels are going to be shortly discussed below, so as to get a better view of Kubernetes functionalities.

- **Deployment:** A Deployment provides declarative updates for Pods and ReplicaSets. The user describes the desired state in a Deployment, and the Deployment Contoller changes the current state to the desired state at a controlled rate. The following is an example of a Deployment. It creates a ReplicaSet to bring up three nginx Pods.

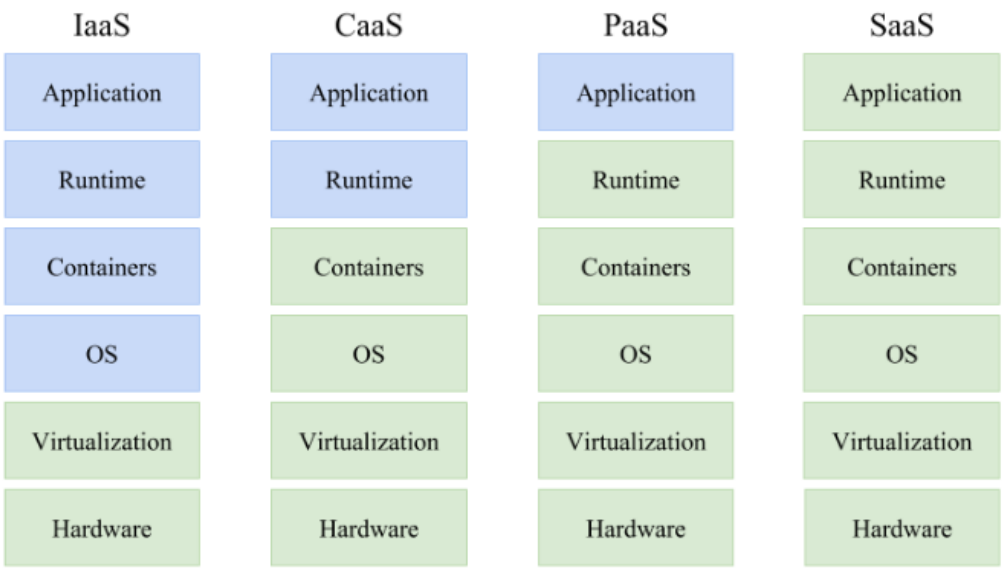
```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
```

```
  metadata:
    labels:
      app: nginx
  spec:
    containers:
    - name: nginx
      image: nginx:1.14.2
      ports:
      - containerPort: 80
---
```

- **ReplicaSet:** A ReplicaSet's purpose is to maintain a stable set of replica Pods running at any given time. Specifically, a ReplicaSet manages the deployment and scaling of a set of Pods, and provides guarantees about the ordering and uniqueness of these Pods. Like a Deployment, a StatefulSet controls Pods that are based on an identical container spec. Unlike a Deployment, a StatefulSet maintains a bond with each of its Pods in the form of a unique identity. It uses the same identity whenever it needs to reschedule those Pods.
- **Pod:** Pods are the smallest deployable computing units that we can create and manage in Kubernetes. A Pod is a group of one or more containers, which share storage and network resources, and a specification for the way they should run the containers. A Pod models a "logical host" for containers that are relatively tightly coupled such as services of one or more applications. New Pods, when created, are scheduled to run on a Worker Node of the cluster, where they remain until they finish their execution. If a Node dies, the Pods scheduled to that Node are scheduled for deletion after a timeout period, thus are ephemeral in nature.
- **Worker Node:** A Worker Node is a worker machine in Kubernetes which is managed by the control plane. It is the place where Pods are scheduled and deployed. A Worker Node can have multiple Pods running.
- **Node Process:** Each Node runs at least three processes: kubelet, kube-proxy and a container runtime that are essential for the services provided. All of them are described above.
- **Container:** The lowest level of Kubernetes abstraction.

Role

As mentioned before, in order to host a serverless platform that deploys containers on demand (functions), in our private infrastructure that is consisted of four heterogeneous virtual machines, we deployed Kubernetes. Out of the four nodes, one served as Master Node while the remaining three served as Worker Nodes.



Σχήμα 3.4: *The figure displays a comparison between the main cloud computing service models. The green components are managed by the cloud provider and the blue ones are managed by the user.*

3.2 Cloud computing

Cloud computing, according to the official definition published by the NIST (National Institute of Standards and Technology) [22] is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g, networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or user-service provider interaction. Cloud computing enables faster provisioning of computer resources, and it abstracts many of the management tasks from the user, letting him/her concentrate on delivering software services without worrying too much for the underlying infrastructure. The different levels of infrastructure abstraction control the amount of user's responsibilities and is dependent on the cloud computing service model.

This chapter introduces the widely identified service models for building software architecture in the cloud. Service models related to serverless computing will be introduced later in section 3. Figure 3.4 visualizes how the management responsibility is divided between the user and the cloud service provider in each kind of service model.

3.2.1 Infrastructure as a Service

Infrastructure as a service (IaaS) is the type of cloud computing model where the capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software which can include operating systems and applications. IaaS is out of the four main types, the one with the minimum level of user abstraction. Commercial examples of IaaS services are: *Amazon Elastic Compute Cloud, Google Compute Enginge,*

Azure Virtual Machines [23]

3.2.2 Containers as a Service

Containers as a ervice (CaaS) is a relatively fresh term of cloud computing service and it is used to describe container orchestration services. It allows software developers and IT departments to upload, organise, run, scale and manage containers by using container-based virtualization. Without CaaS, software development teams need to deploy, manage and monitor the underlying infrastructure that containers run on, which consists of cloud machines and network routing systems. Well known tools that offer this capability are: *Docker Swarm, Kubernetes, Google Kubernetes Engine, Amazon EKS* [24]

In comparison to IaaS, CaaS provides better re-usability and portability because of containers' nature that can be moved to the cloud, or deployed across private and public clouds. The service provider usually offers automation and tools for provisioning hosts where containers are run, rescheduling failed containers, updating new ones and scaling the cluster up or down. But, the user is still responsible for security patches to be applied. CaaS enables software development teams to design and operate at the higher order of container level instead of spending valuable time for lower level infrastructure and systems.

3.2.3 Platform as a Service

When it comes to platform as a service (PaaS), the capability provided to the user is to deploy onto the cloud infrastructure user-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The user does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration setting for the application-hosting environment. [25]

PaaS model makes application management and deployment more straightforward by providing automatic scaling, healing, and load balancing functionalities. The cloud provider, also, usually offers pre-configured auxiliary services like monitoring and logging for the application's needs. Therefore, the developers can focus more on application source code and logic without needing to manage the underlying operating system. However, this kind loss of control can reduce the application's portability by causing a vendor lock-in for the users.

3.2.4 Software as a Service

Last but not least, Software as a service (SaaS) is the cloud computing model with the largest market share among the others. Specifically, the SaaS market is by far the largest market, according to a Gartner study [26] that reported that enterprises spend $182B on cloud services, with SaaS services making up to 43% of that spend.

In SaaS, the capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. These applications are accessible from a variety of

client devices through either a thin client interface, such as a web browser, or a program interface. The consumer, obviously, does not manage or control the underlying cloud infrastructure including network, servers, operating systems or storage [27]. This ease, offered to the user, comes at the cost of limited user-specific application configuration settings.

3.3 Serverless computing

Serverless computing represents the idea, where the user does not need to manage any server infrastructure but instead deploys the application code to the cloud provider's platform. Whatever is needed for the application logic to be executed, is scaled and billed on-demand for the user, without any running costs. It was in 2014 when the term gained massive popularity, after Amazon announced their AWS Lambda service [28]. Since 2015, there have been numerous other serverless platforms including IBM Cloud Functions, Google Cloud Functions, Microsoft Azure Cloud Functions and more. There are also multiple open source serverless platforms to run serverless workloads in private cloud infrastructures, such as OpenWhisk, OpenFaaS, Fission, Knative and Kubeless. [4, 5, 29, 30, 31]

Despite the etymological meaning of the term "Serverless" which indicates the absence of physical servers, servers do exist but are not managed by the service users. What it means is that the servers' maintenance is abstracted from the user. The cloud provider is responsible to manage, maintain, scale and update the underlying infrastructure according to the user's needs. As a result, by automating these tasks, the application developers have more time to focus on application code and so the *time-to-market* for such software products is significantly decreased. In some sense, Serverless is the "child" of microservices, container virtualization and event-driven programming.

3.3.1 Serverless cloud computing models

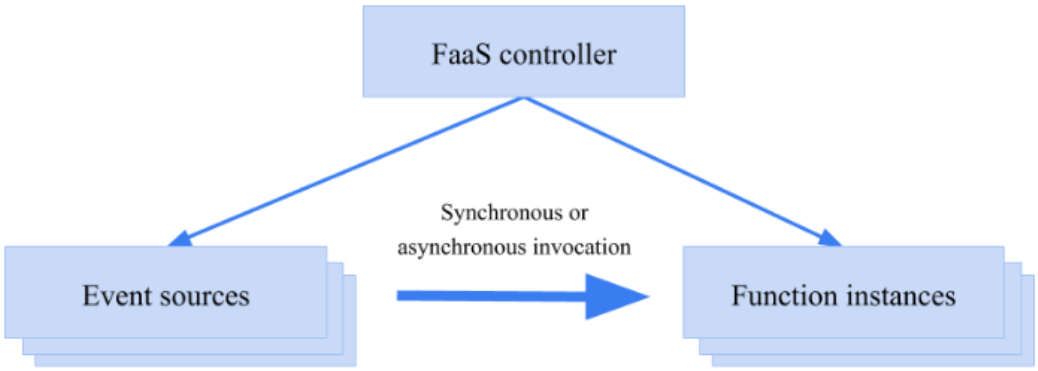
Cloud Native Computing Foundation (CNCF) and Berkeley, divide serverless into two cloud computing service models: Backend as a Service (BaaS) and Function as a Service (FaaS). [32]

Backend as a Service

Backend as a service (BaaS) refers to third-party services, which provide an API to replace or enhance a subset of functionalities in an application. BaaS vendors provide pre-written and transparent software for activities that operate on servers, such as user authentication or database management.

Function as a Service

Function as a service (FaaS) refers to the logic where cloud platforms allow users to execute code in response to events. These events could be triggered, for example, by an HTTP request, database operation or a new message in a message queue. Developers



Σχήμα 3.5: *FaaS model components.*

deploy granular, event-driven functions to the cloud platform which operate in a stateless manner when an event invokes them. FaaS functions are small, modular, highly scalable and relieve developers from writing code for the underlying infrastructure management.

In contemporary FaaS platforms, each function is executed on a seperate container and thus are stateless in nature. Their lifetime is generally short (a matter of milliseconds) but it is up to the cloud provider how long a function instance actually lives. A typical FaaS platform environment contains three main participants: FaaS controller, sources of events, and function instances. Figure 3.5 visualizes the connection among these participants. FaaS controller serves as an interface between the event sources and the function instances. When a new event is created by the event sources, the FaaS controller is going to trigger the right function to operate.

The life-cycle of a FaaS function starts when the developer feeds the platform with the function's code along side with configuration settings that would determine the function's runtime profile such as its scaling behaviour or useful environment variables. In the provider's side, the platform takes the code and transforms it into a container, which upon demand could be deployed as a function instance.

3.3.2 Benefits and drawbacks

Serverless comes around with a great list of benefits. Widely perceived benefits [33] of serverless are listed below:

- **Granular billing:** The price paid by the end users does vary with the load because of the on-demand philosophy. Serverless has a linear cost structure and there is no charge when the function code is idle. This leads to great cost cuts when used effectively.

- **Scalability ease:** Developers have no scaling responsibilities. Their service would be offered via the cloud provider, to any system load automatically and transparently, without having them to worry.

- **Efficient use of resources for cloud providers:** Cloud providers offering serverless computing services are able to distribute their infrastructure load effectively as the FaaS functions are short-lived and could be deployed on demand. This means providers can share resources between customers while there is no need to reserve resources for idle capacity.

- **Low time to market:** As the end users have more time to focus on application code rather than spending time to manage, maintain and scale the underlying infrastructure, they can create software products with significantly lower *time-to-market*.

- **Machine learning's gateway:** Recently, some machine learning use cases are forged because of serverless computing. More specifically, by creating machine learning functions that are subject to user requests, one can employ a machine learning tool on demand without having to worry for overpricing and thus make artificial intelligence more publicly accessible.

Although serverless has its advantages, there are also drawbacks. While making progress in serverless platforms and tools these drawbacks could be improved later. Some of the commonly identified drawbacks are listed below:

- **Vendor lock-in:** Every provider-specific offering comes with varying sets of features and requirements which have to be fullfilled to actually start using it. These include specific data formats, custom configuration settings and different likelihoods in programming languages. Consequently, the serverless developers are tighlty coupled to their current cloud service provider.

- **Cold starts:** The cold start problem is widely known across the cloud computing field. Depending on the dependencies and the runtime environment of each function, the container, when invoked for the first time, that carries its instance suffers from relatively large start-up times (1-3 seconds long) that can cause dramatic impact on certain types of applications. On the other side, when the desired container pre-exists the function request (warm container) and the resources are already allocated there is no such delay. The frequency of cold starts needs to be deeply examined when designing a serverless workload.

- **Data transfer load:** As commented above, FaaS functions are stateless. Thus no state information can be saved between the invocations and an external database has to be used for that particular reason. This can lead to high traffic load when serverless workflow execution happens.

- **Vendor and end users tradeoff:** A serious dilemma is present in the serverless stakeholders ecosystem and has to do with the cloud provider and the end-user. From the cloud provider's point of view, it is beneficial if executing a function's container takes longer times as this tactic is more profitable because the resource allocation is more lasting. But end-users demand low latency execution for obvious reasons and in many cases they are not willing to sacrifice their service's end-to-end

latency in some way. This problem is taken into consideration from both sides when billing matters kick in.

3.3.3 Serverless platforms

This section discusses different FaaS platforms, both commercial and open source, as well as OpenWhisk and OpenFaas, the open source platforms we chose to work our project on.

Commercial FaaS platforms

In the cloud market there are numerous commercial FaaS platforms and each has its selection of features and runtimes supported. Some of the most popular names in this field are: AWS Lambda, IBM Cloud Functions, Google Cloud Functions, Microsoft Azure Functions.

Depending on a range of parameters such as billing, memory asigned to function or runtimes supported, each platform offers its own package in order to be competitive enough in the cloud market. In table 3.1 we can have a look at these existing projects. We need to take into consideration the fact that serverless lives in the realm of tradeoffs. Thus depending on the user needs and constraints, the choice of a serverless platform can vary at all times.

	AWS Lambda	**Google Cloud Functions**	**IBM Cloud Functions**	**Microsoft Azure Functions**
Memory (MB)	{128 ... 10240}	$128 \times \iota$, $\iota \in \{1, 2, 4, 8, 16, 32\}$	{128 ... 2048}	up to 1536
Billing	Execution time based on memory	Execution time based on memory & CPU-power	Execution time based on memory	Execution time based on memory used
Billing interval	1ms	100ms	100ms	1ms
Configurable Resource	memory	Memory & CPU-power	memory	n/a

* Source: A Comparison of Serverless Function (FaaS) Providers [33]

Πίνακας 3.1: *Commercial FaaS platforms comparison*

Open source FaaS platforms

Open source projects cannot be missing from the serverless field. In fact, open source platforms provide access to IT innovations so many developers are interested in those. Some of the most well-known open source FaaS platforms include: Apache OpenWhisk, OpenFaas, Knative, Kubeless and Fission. Each of them comes with its special attributes, functionalities and weakenesses.

If one wants to avoid the vendor lock-in problem and wants not to depend on a specific cloud provider, open source FaaS platforms is the way to go. Open source platforms allow running serverless workloads in a private cloud where the end user is responsible for

scaling and maintaining the underlying infrastructure. This is not the exact case where serverless technology is fully utilized but when it comes to a big company, this could be easily tackled by having an operations team to manage the serverless platforms infrastructure, which would be used by the developer team with the appropriate abstaction level to promote productivity.

When choosing an open source FaaS platform to work on, there are multiple important factors to consider and decide afterwards. Some of them are: language support, supported container orchestrator, performance, developer community and function trigger types. These factors are to be discussed below.

- **Language support:** All open source Faas platforms make available a wide range of runtimes. Furthermore, most platforms offer tools to create custom runtimes to enrich their language support. It has to be mentioned that each language runtime comes with different performance metrics and so a single runtime would not behave the same way in any platform.

- **Supported container orchestrator:** In order to manage the underlying containers, a FaaS platform needs a container orchestrator. Kubernetes is the container orchestrator every platform supports. But, OpenFaas also supports Docker Swarm and more custom orchestrators.

- **Performance:** The perfomance of a Faas platform is directly affected by its underlying architecture and execution overheads. When perfomance is critical for the user, a detailed examination on every platform cannot be avoided.

- **Developer community:** Any open source project is by default community driven. This means that if a project does not have a good enough community support, its development and maintainance might be abandoned in any time. On the opposite side, when an open source project is widely adopted by developers and organisations, then it is going to evolve in many directions.

- **Function trigger types:** There are three different types of function triggers: HTTP, event and scheduled. HTTP triggers are HTTP requests, which start the execution of a function and return a response to the client. Event function triggers are usually created by a new addition in a message queue of any kind. Scheduled triggers are the ones that are planned to arrive on certain frequency, e.g every two hours, twice a day or ten times per month. Knative, Fission, Kubeless and OpenWhisk support all three of them and OpenFaas supports HTTP and event triggers.

3.3.4 Apache OpenWhisk

History

OpenWhisk is an Apache project which provides a complete Serverless platform, client SDKs and intergration tooling. Specifically, the project started in February of 2015 with a small team of IBM researchers and was named *Whisk*. A year later, when the project

was open sourced on GitHub, it got a new name: *OpenWhisk*. OpenWhisk was developed with two main goals:

- Discovering the promising offerings of serverless computing
- Building the foundations upon which the open source, research community would push forward the serverless computing domain

Architecture

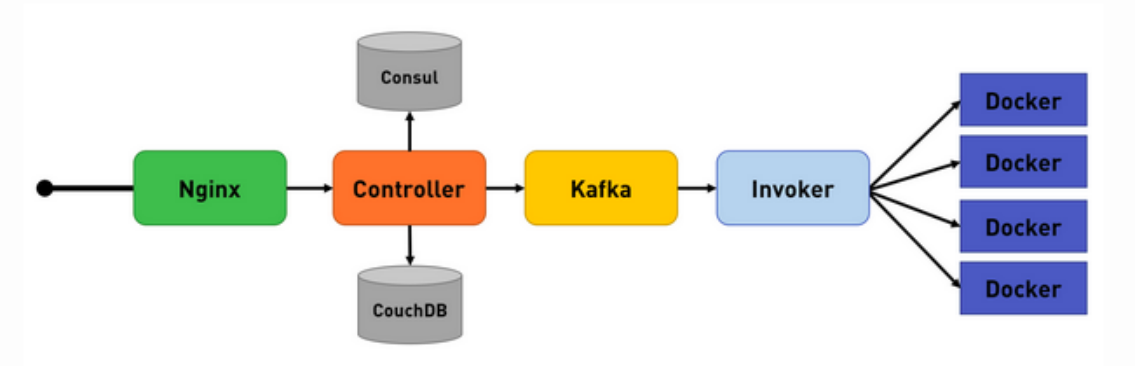
Developers write functions, in a variety of programming languages, and deploy them to OpenWhisk as actions. Actions are trigger-driven and that means when a new valid event is created by the event source, it would trigger the execution of the appropriate action. It is up to the developers to create one or more triggers and bind them to one or more actions. By default, no action is bound to any trigger and that is the reason of *rules* existence. Rules are the tool the developer uses in order to associate a trigger with an action.

The OpenWhisk system contains a Nginx server, an API Controller, CouchDB, a Consul key-value store, a Kafka and Invokers.

- **Nginx server:** It is an HTTP server that acts as a reverse proxy for the interface between clients and OpenWhisk's API.
- **API Controller:** Responsible for authentication and orchestration processes, API controller serves and forwards incoming requests to the appropriate component.
- **CouchDB:** It is the place where system's state is saved. CouchDB is an open source JSON data store and keeps record of all types of data including credentials.
- **Consul:** For distributed systems operations, a distributed key-value store is needed for managing the system's state. Consul is accessible by all OpenWhisk components
- **Kafka:** Apache Kafka is an open source, high perfomance distributed event streaming platform which in the OpenWhisk use case, connects the Controller with Invokers. Any message sent by the Contoller, is buffered into Kafka and then it is delivered to an Invoker.
- **Invoker:** It is the runtime plane of OpenWhisk, as upon request it spins up a container that executes the invoked action. An Invoker communicates with CouchDB both for configurating the container specs and saving the execution's result for later retrieval.

Why OpenWhisk

It is super essential the fact that OpenWhisk has got a long list of built-in supported language runtimes but also offers the capability to the developer to create and customize special runtimes, with the help of Docker SDK, which are not supported by the platform.



Σχήμα 3.6: *Apache OpenWhisk architecture.*

To add, the deployment of OpenWhisk to Kubernetes could be managed and provisioned with ease as the documentation is pretty detailed. Last but not least, the OpenWhisk components are modular enough to enable the operation of custom monitoring services as well as the construction of higher-level custom tools for the execution of serverless workloads.

3.3.5 OpenFaas

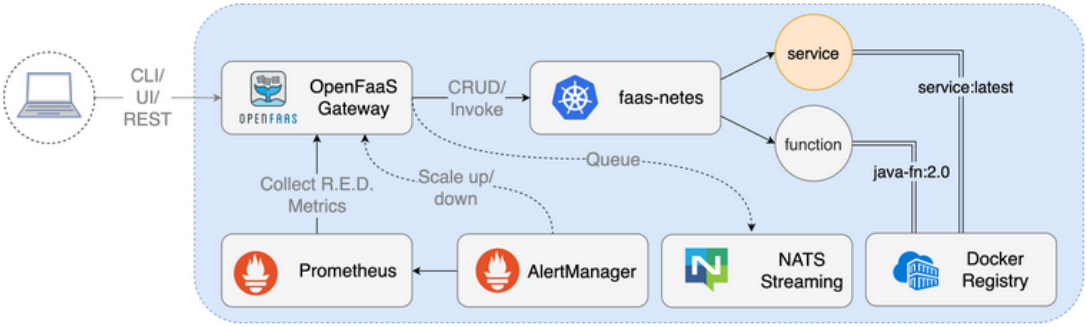
History

OpenFaas is an independent open source project created by Alex Ellis and started back in 2016. In 2017 after Dockercon, Kubernetes support was added and the community seemed to be really enthusiastic about it. Since then, OpenFaas has had the highest adoption rate, as compared to its alternatives, with more than 20k GitHub stars. That results in huge community contributions, which are crucial for the project's evolution. Along the way, OpenFaas Pro was realeased, which is a commercial distribution of OpenFaaS for companies and enterprises with some extra included features, suitable for production usage.

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. Figure 3.7 visualizes how these components interact each other.

- The **Gateway** can be accessed through its REST API, via the CLI or through the OpenFaas UI
- A **Watchdog** component is intergrated into each container and provides a common interface between the user and the function
- **Prometheus** collects metrics which are available via the Gateway's API and which are later used for auto-scaling purposes



Σχήμα 3.7: *OpenFaas architecture and components.*

- OpenFaas enables long-running tasks or function invocations to run in the background through the use of **NATS streaming**
- The **Queue-Worker** processes asynchronous function invocations requests
- **Alert manager** reads usage (requests per second) metrics from Prometheus in order to know when to fire an alert to the API Gateway

Asynchronous function invocations

OpenFaas enables function invocations to run in the background while it handles HTTP transactions for other function calls.

By exploiting the asynchronous function invocation capability, we can achieve parallelism by scaling the number of queue-workers deployed in the cluster which is by default set up to run a single task at a time. By tuning the queue-worker's "max-inflight" option to a value greater than one, we can also increase the level of parallelism. Essentially, for n replicas of queue-workers we can have up to $n*$ *max-inflight* parallel invocations. The asynchronous pipeline works as follows:

- An initial connection is formed to the gateway
- the user's request is serialized to a queue via the queue-worker and NATS
- at a later time, the queue-worker dequeues the request, deserializes it and forwards it to the function (directly or via the gateway using a synchronous call)

MAYBE MORE THINGS TO ADD HERE.

Why OpenFaas

Choosing OpenFaas as the major serverless platform for this thesis was not an obvious solution. At first, we started to work on this thesis with Apache Openwhisk. But, the main drawback of OpenWhisk was the difficulty to operate at the pod-level of the cluster. Specifically, the cluster manager is not capable of placing a deployed function to a desired node. The only way, one may approach this is by making changes to Openwhisk's source

code, something considered as counter-productive. On the opposite side, OpenFaas is by far more Kubernetes-friendly than OpenWhisk and that issue could easily be resolved. Moreover, creating functions and managing language dependencies was not that easy in OpenWhisk as it was in OpenFaas where the conceptual workflow of function development is highly modular and manuverable.

3.4 Machine Learning

Machine learning is a core subfield of artificial intelligence, which is broadly perceived as the capability of a machine to emulate intelligent human behaviour while executing a properly defined task. These tasks are complex in nature and usually include written text understanding and reasoning, object recognition and classification as well as mimicing physical world actions. Machine learning essentially serves as a mean of deploying artificial intelligence.

A common misconception with machine learning, when compared to the traditional way of programming computers, is that in machine learning the machine is capable of learning on its own without human intervention, whereas in the "old-school" way one should provide the machine with a full-complete instruction set in order to perform a job, which is frequently referred to as a painful task to do. But there is little truth in this particular statement, as machine learning requires massive amount of human effort in calibrating a machine in the early stages of its development. Once carefully calibrated though, machines operate on their own and are capable of solving complex problems with high efficiency. The impact of machine learning is already enormous in a wide variety of domains and it is considered to grow even in a larger scale in the near future.

Machine learning has to do with data manipulation and these data could be processed by a broad set of techniques that are currently under development and would allow us to deal with more and more interesting problems. These techniques are categorized in three different subcategories in terms of the way the machine is interacting with the available data:

- **Supervised** machine learning models are trained with labeled data sets. While training, they are forced to map an input to an output by making use of example input-output pairs.

- **Unsupervised** machine learning is where a program tries to find patterns in unlabeled data that humans are not explicitly looking for.

- **Reinforcement** machine learning trains a machine through trial and error to take the best out of them by establishing a reward (or penalty) policy when the program makes good or bad decisions respectively.

3.5 Deep Reinforcement Learning

From an abstract point of view, the efficiency and the effectiveness of a machine learning solution highly depends on the quality and characteristics of data and the perfomance of the learning algorithms. Deep learning is a subset of machine learning which distinguishes itself from classical machine learning by the type of data it works with and the methods in which it learns. While machine learning leverages structured, labeled to make predictions, this does not imply that it does not use unstructered data; it is just necessary to pre-process those data in some way to organise it into a structured format. Deep learning eliminates some of data pre-processing that is typically involved with machine learning, because of its ability to ingest large amounts of data such as text and images and decide whether some features of the input data are important or not for making a successful prediction.

Deep learning is a type of machine learning that is computationally expensive by design [34]. Deep-learning architectures such as deep neural networks, recurrent neural networks or convolutional neural networks have been applied to fields including computer vision, speech recognition, natural language processing and more, where they have produced high quailty results which in some cases are surpassing human expert perfomance. Artificial neural networks (ANNs) were inspired by information processing and communication nodes in biological systems, the human brain for example. The adjective "deep" in deep learning refers to the use of multiple of such layers in a network. Deep learning approaches may also be coupled with reinforcement learning methods, as it currently enabling reinforcement learning scale to problems that were previousle intractable, such as learning to play video games directly from pixels [35].

3.5.1 Reinforcement Learning

Reinforcement learning (RL) has a clear objective: to maximise expected cumulative rewards, which seems simple at first sight but developing efficient algorithms to optimise such objectives usually involve a pipeline of research, experimentation and investigation that quite often takes a lot of time and effort.

In a great range of problems including dynamic workload scheduling and robotics, RL-based techniques have achieved significant performance. In general, an agent whose target is to solve a problem, is fitted inside an environment and interacts with it via a series of actions, observations and rewards. Upon observing the sequences of its actions, the agent can learn to alter its own behaviour in response to rewards received. This paradigm of trial-and-error learning has its roots in behaviourist psychology, and is one the main foundations of RL. Thus creating an ideal learning environment is a vital job for a domain expert.

In the RL set-up, an autonomous agent, controlled by a machine learning algorithm, observes a *state* s_t inside its environment at timestep t. The agent interacts with the environment by taking an action a_t in state s_t and so it transitions to a new state s_{t+1}. Every time the environment transitions to a new state, it offers the agent a reward r_{t+1} as

a feedback to the selected action. The agent's goal is to discover a policy π that maximises the expected return. Formally, RL is a Markov decision process which consists of:

- A set of states S, plus a distribution of starting states $p(s_0)$.
- A set of actions A.
- Transition dynamics $\mathcal{T}(s_{t+1} \mid s_t, a_t)$ that maps a state-action pair at time t onto a new state s_{t+1} at time $t+1$.
- A reward function $R(s_t, a_t, s_{t+1})$.
- A discount factor $\gamma \in [0, 1]$, where lower values mean more emphasis on immediate rewards.

3.5.2 Reinforcement Learning Algorithms

There are two main approaches to solving RL problems: *Value functions* and *policy search*.

A. *Value Functions*

Value functions methods are based on estimating the value of the agent's presence in a given state. The *state-value* function $V^\pi(s)$ is the expected reward when starting in state s and following policy π henceforth:

$$V^\pi(s) = \mathbb{E}[R \mid s, \pi] \tag{3.1}$$

The optimal policy, π^*, has a corresponding state-value function $V^*(s)$, and vice-versa, the optimal state-vale function can be defined as

$$V^*(s) = max_\pi V^\pi(s), \forall s \in S. \tag{3.2}$$

If we had $V^*(s)$ available, the optimal policy could be retrieved by choosing among all actions available at s_t and picking the action a that maximises $\mathbb{E}_{s_{t+1} \sim \mathcal{T}(s_{t+1} \mid s_t, a)}[V^*(s_{t+1})]$.

In the RL setting, the transition dynamics $\mathcal{T}$ are unavailable though. Therefore, we construct another function, the *state-action-value* or *quality function* $Q^\pi(s, a)$ which is similar to V^π, except that the initial action a is provided, and π is only followed from the succeeding state onwards:

$$Q^\pi(s, a) = \mathbb{E}[\mathbb{R} \mid s, a, \pi]. \tag{3.3}$$

The best policy given $Q^\pi(s, a)$, can be found by choosing $\boldsymbol{a}$ greedily at every state: $argmax Q^\pi(s, a)$. Under this policy, we can also define $V^\pi(s)$ by maximising $Q^\pi(s, a)$: $V^\pi(s) = max_a Q^\pi(s, a)$

Dynamic Programming: To actually learn Q^π by exploiting the Markov property and defining the function as a Bellman equation, the following recursive form occurs:

$$Q^\pi(s_t, a_t) = \mathbb{E}_{s_{t+1}}[r_{t+1} + \gamma Q^\pi(s_{t+1}, \pi(s_{t+1}))]. \tag{3.4}$$

This means that Q^π can be improved by bootstraping, i.e, we can use the current values of our estimate of Q^π to improve our estimate for future states. This is the foundation of Q-learning and the state-action-reward-state-action (SARSA) algorithm:

$$Q^\pi(s_t, a_t) \leftarrow Q^\pi(s_t, a_t) + a\delta \tag{3.5}$$

where a is the learning rate and $\delta = \Upsilon - Q^\pi(s_t, a_t)$ the temporal difference (TD) error; here, Υ is a target as in a standard regression problem. SARSA, an *on-policy* algorithm, is used to improve the estimate of Q^π by using transitions generated by the behavioural policy, which results in setting $\Upsilon = r_t + \gamma * Q^\pi(s_{t+1}, a_{t+1})$. An *on-policy* algorithm essentially estimates the value of the policy being followed. Q-learning is *off-policy*, as Q^π is instead updated by transitions that were not necessarily generated by the derived policy. Instead, Q-learning uses $\Upsilon = r_t + \gamma * max_a Q^\pi(s_{t+1}, a)$, which directly approximates Q^*

B. *Policy Search*

When speaking about policies, formally we say that an agent "follows a policy". For example, if an agent follows policy π at time t, then $\pi(a \mid s)$ is the probability that $A_t = a$ if $S_t = s$. This means that, at time t, under policy π, the probability of taking action a in state s is $\pi(a \mid s)$.

Policy search methods do not need to maintain a value function model, but their direct goal is to find an optimal policy π^*. Typically, a parameterised policy π_∂ is chosen, whose parameters are updated to maximise the expected return $\mathbb{E}[R \mid \partial]$ using either gradient-based or gradient-free optimisation. Neural networks that encode policies have been successfully trained using both gradient-free and gradient-based methods. Gradient-free optimisation enables solving of low-dimensional parameter spaces, but despite some successes in applying them to large networks, gradient-based training remains the *way-to-go* for most of DRL algorithms, being more sample-efficient when policies possess a large number of parameters. In simple words, policy search focuses on finding good parameters for a given policy parameterization.

3.6 Scheduling and Migration of Serverless Functions

3.6.1 Why is Scheduling of Serverless Functions (SSF) needed?

When a cloud provider takes the responsibility of providing on demand serverless functions' execution, a lot of parameters have to be taken into consideration because of the heterogeneity of users' requests, complexity of code injected into a function and of course infrastructure level issues such as intereference, machine availability and maintainance

costs. As a concequence, scheduling effieciently serverless workflows is crucial. With a poor scheduling strategy, one may result in unbearable traffic, unpredictable utilization and end-user disatisfaction as their requests would suffer from low quality of service (QoS).

3.6.2 How Does Scheduling of Serverless Functons Work

Quite a few ways of tackling the scheduling of serverless functions, or containers in general, exist and are applicable depending on the problem that has to be resolved. Most of the solutions do not take into account the runtime profile of the function as they decide to deploy it in a machine where the Docker image of the appropriate function is pulled already if possible or place the function to the most resource-rich machine. But this kind of tactics are not going to be efficient in all deployment scenarios apparently. For example, the most resource-rich machine might be under resource-stress conditions and so it would not be able to outperform its fellow machines inside the cluster as expected. In most cases, a serverless framework employs a container orchestrator, such as Kubernetes, under the hood for scheduling the deployed functions. However, these schedulers are more generically designed; thus custom approaches should be adopted to solve less generic problems. An important conclusion to be drawn here is the fact that static solutions in scheduling problems definitely are not the best approach when trying to solve a dynamic problem.

Κεφάλαιο 4

Motivational Analysis

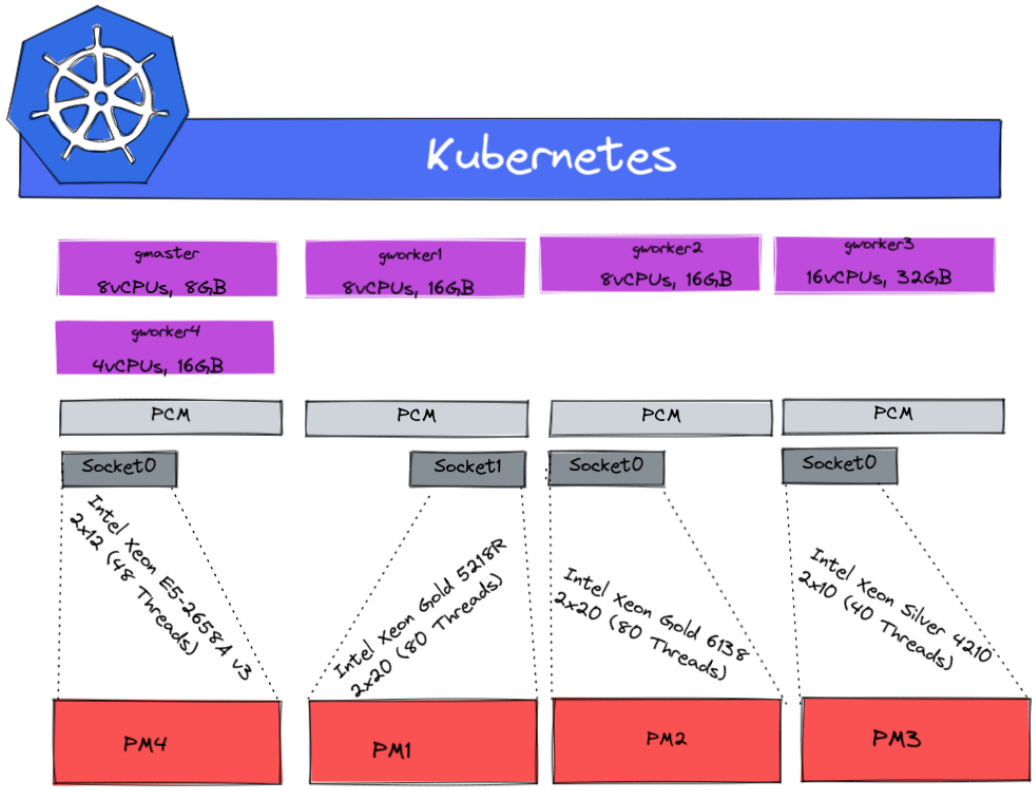
Cloud computing advanced significantly after the hardware virtualization and virtual machine ideas. For years, a lot of paradigms and platforms were created and developed that prompt to major long-term cost savings for businesses of all sizes. The next big step after hardware virtualization was the operating system level virtualization, also known as containerization. But, the need for even higher elasticity and more fine-grained billing has recently led to the proliferation of serverless. Serverless, lately, is examined not only for event-driven stateless applications but also for data analytics, machine learning and high perfomance computing workflows and that is because serverless execution environments offer ease-of-use and pay-as-you-go billing to scientists that have little experience in provisioning clusters at scale. As pointed out in Eric Jonas et al work[36], even at UC Berkeley, as it was examined via informal surveys, it is ordinary for machine learning graduate students not to have written cluster computing jobs due to the complexity of setting up cloud platforms.

In this chapter, we analyze and verify the key role of granularity when deploying a w-orkflow on cloud premises regarding metrics like latency and utilization. Furthermore, we present the impact interefence and heterogeneity could have on a serveless application's performance and also on the end-user experience. Firstly, a description of our experi-mental setup is offered in Section 4.1, including tools and software that are integral to this thesis. Secondly, in Section 4.2, we are referring to implementation tools. Moreover, in Section 4.3 the workflow we studied is characterized so as to get a clear view of its computational complexity profile, while in Section 4.4, Section 4.5 and Section 4.6 gra-nularity's, inteference's and heterogeneity's impact are investigated. Finally, in Section 4.7 we highlight the reasoning behind our approach to tackle the serverless functions scheduling problem.

4.1 Experimental infrastructure

4.1.1 System setup

For the rest of the thesis, we consider four multi-processor systems *PM1*, *PM2*, *PM3* and *PM4* as shown in table 4.1. In order to simulate a proper cloud environment, all of the referenced tools and workloads have beed containerized with the aid of the Docker platform [37]. Furthermore, as the nodes of our cluster, five virtual machines (VMs) have



Σχήμα 4.1: *Cluster Architectural Overview*

beed deployed on top of physical machines with diverse specifications. Specifically, each VM's cores range from 4 up to 16 and RAM size ranges from 7.77(GB) to 31.4(GB) and, we use KVM as our hypervisor. In fact, in PM_i we have deployed the virtual machine $w0_i$, where $i \in 1, 2, 3, 4$

The approach of employing VMs with deployed containers is the common practice of deploying cloud clusters at scale, since it establishes reliability and robustness. The virtual cores of each VM have been mapped on physical cores of the servers using the CPU pinning options of the **libvirt** library, to eliminate context-switching and monitor clearly VM-specific metrics. On top of the aforementioned VMs, Kubernetes [3] is deployed as our container orchestrator. The system holistically is presented in figure 4.1. The Kubernetes cluster is configured with a single-master node with the corresponding VM serving as master deployed in a different socket than the VM that serves as the fourth worker that is also deployed on the same server (Cheetara).

4.1.2 Monitoring and Communication

In this subsection, the monitoring and communication means that were used for this work are going to be introduced. First of all, in order to get information about the real system metrics we utilized the Perfomance Counter Monitor (PCM). PCM is a tool developed by Intel. It is deployed as an agent in machine and extracts from it a wide range of metrics. The Intel®Perfomance Monitor Counter provides C++ routines and utilities to estimate the internal resource utilization of the Intel® Xeon® and Core™processors.[38].

VM/Server	vCPUs	Memory	Underlying CPU(Intel ®Xeon ®)	L3(MB)
W01/PM1	8	15.6GB	Gold 5218R @ 2.10GHz	28
W02/PM2	8	15.6GB	Gold 6138 @ 2.00GHz	28
W03/PM3	16	31.4GB	Silver 4210 @ 2.00GHz	14
W04/PM4	4	15.6GB	E5-2658A @ 2.20GHz	30

Πίνακας 4.1: *Technical characteristics of heterogeneous nodes*

The CPU utilzation does not tell you the actual utilization of the CPU. CPU utilization number otained from operating systems (OS) is a metric that has been used for product sizing, compute capacity planning, job scheduling, and so on. The current implementation of this metric (the number that the UNIX* "top" utility and the Windows* task manager report) shows the portion of time slots that the CPU scheduler in the OS could assign to execution of running programs or the OS itself; the rest of the time is idle. For compute-bound workloads, the CPU utilization metric calculated this way predicted the remaining CPU capacity very well for architectures of 80ies that had much more uniform and predictable performance compared to modern systems. The advances in computer architecture made this algorithm an unreliable metric because of introduction of multi core and multi CPU systems, multi-level caches, non-uniform memory, simultaneous multithreading (SMT), pipelining, out-of-order execution, etc. [39]

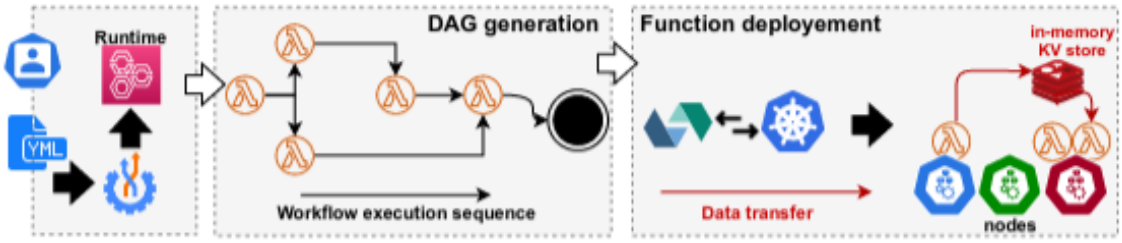
Using PCM we were able to extract core, socket and system level metrics from our cluster. Those metrics were interpretated as the state of our system. The metrics that were used are the following:

- **Instructions Per Cycle (IPC)** IPC describes the instructions required to execute a piece of code divided by the the number of hardware cycles done at this time.

- **Memory Reads / Writes** Memory Reads/Writes describe the number of reads/writes from/to the memory. It is provided only at socket and system level and extracted on a set time interval.

- **L3 Misses** L3 Misses counts the L3 cache misses occured in a certain time interval. For the socket and system level, L3 misses are aggregated for all cores that are included in the socket or the system.

- **C-States(C0, C1)** For energy saving reasons during CPU's idle state, the CPU could be forced to enter a low-power mode. Each core has three scaled idle states: C0, C1 and C6. C0 is the normal CPU operating mode and so the CPU is 100% active. The higher the C index is, the less activated is the CPU.



Σχήμα 4.2: *Simple-sw architectural overview.*

4.2 Implementation tools

4.2.1 Simple-sw

During the early stages of this thesis, the Faas platform we chose to work on was Apache OpenWhisk. Due to the need of finding a way to deploy a FaaS workflow, we looked for an open source runtime tool that could orchestrate OpenWhisk functions in a DAG-like manner so as to use it accordingly to our needs. From an abstract point of view, our vision to be implemented required deploying FaaS functions to a Serverless platform and afterwards defining rules and conditions upon which functions would communicate, before and after their execution in order to operate as a single unit.

Despite a tightly coupled function chaining capability offered by OpenWhisk itself, nothing more interesting was found online. So, we decided to build our own runtime orchestrator tool with the aid of *Serverless Workflow*. Serverless Workflow is an open source project developed by Cloud Native Computing Foundation and offers standards-based DSL, open source dev tools and runtimes within a vendor neutral, community-driven workflow ecosystem. One of Serverless Workflow's offerings is *Specification* [40], which enables developers define DSL-based workflows. Specification comes together with developer SDKs in a variety of programming languages.

We leveraged Specification and GoSDK and created *simple-sw*. Simple-sw is a tailor-made workflow orchestrator that manages communication and invocations among deployed OpenWhisk functions. By providing a YAML file according to Specification's DSL, simple-sw is going to execute each state of the DAG workflow defined in the YAML input file. In more details, Specification's DSL offers a variety of executable states for a workflow DAG, such as event state, operation state, for-each state. Under simple-sw, we implemented these functionalities in the extend that a complex enough workflow would be deployable within our runtime tool.

4.2.2 MinIO

MinIO offers a high-perfomance, cloud native, open source object storage that can be easily deployed to a Kubernetes cluster. It is released under GNU Affero General Public License v3.0 and is written in Go. It is API compatible with Amazon S3 cloud storage service and can handle unstructured data such as photos, videos, log files and container images with (currently) the maximum supported object size of 5TB. [41]

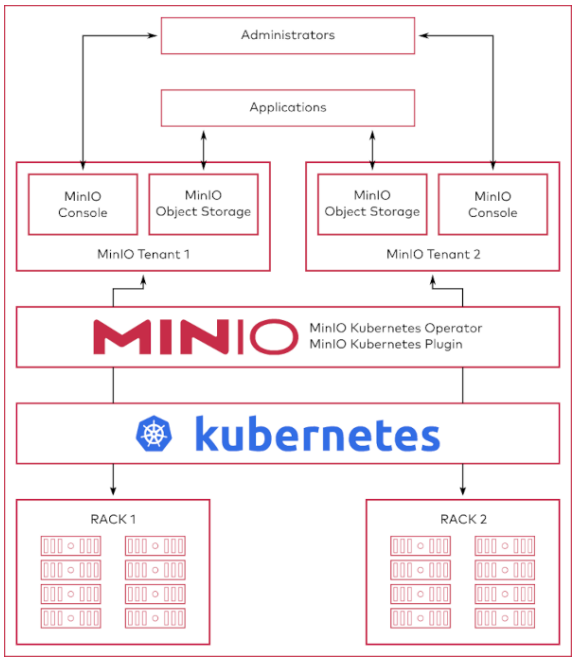
MinIO is designed in a cloud native manner to scale sustainably in multi-tenant environments. Kubernetes is the main orchestration platform that provides a perfect cloud native environment to deploy and scale MinIO. For example, we can set up multiple MinIO tenants that are managed by a MinIO operator and act as different servers in a Kubernetes cluster. MinIO cloud storage server is designed to scale efficiently while remaining a lightweight application so that can be bundled effortlessly with the application stack. Moreover, MinIO server is hardware agnostic, thus can be installed on physical, virtual machines or a Docker container deployed on platforms like Kubernetes.

- **Architecture:** By installing the MinIO Operator in a Kubernetes cluster, we can deploy as many MinIO Tenants as we need. Each MinIO Tenant represents an independent MinIO Object Store within the Kubernetes cluster and requires sufficient Persistent Volumes for binding. In figure 4.3 this architecture is visualized.

- **MinIO Operator:** The MinIO Operator extends the Kubernetes API to support deploying MinIO-specific resources as a Tenant in a Kubernetes cluster. The MinIO Kubernetes Operator automatically generates Persistent Volume Claims (PVC) as part of deploying a MinIO Tenant. Furthermore, the MinIO Operator installs and configures the Console for each tenant by default. Through the Console, a wide variety of tasks could be performed including policy configuration, information overview and usage percentages.

- **MinIO Tenant:** A MinIO Tenant deployed within a Kubernetes cluster is a independent MinIO server instance that needs sufficient resources to allocate. Throught the Operator, multi-tenant environments could be created.

- **Role:** For the shake of intermediate storage for a serverless workflow execution there was a need for a cloud native, high perfomance object storage. As explained above, there is no better solution than MinIO. We chose to deploy a single MinIO tenant that was used for ephemeral file storage as well as efficient state transfer across the FaaS functions when a workflow instance was invoked.



Σχήμα 4.3: *MinIO deployed above a Kubernetes cluster.*

4.2.3 Faas-flow

Faas-flow [6] is an open source tool for FaaS function composition with OpenFaas. It is written in Go while it is stateless by design. Faas-Flow allows us, by defining a simple pipeline, to orchestrate and execute a complex workflow that is consisted of multiple functions without having to worry about the internals.

It promotes function reusability as it accepts a single function to be used in more than one workflows. Furthermore, FaaSFlow makes possible to manage with great ease functions that execute different application logic while offering programmatical power for a variety of different implementations.

Faas-flow is deployed and provisioned just like any other OpenFaas function. In some sense, the faas-flow function could be considered as a wrapper function that handles the functions that are included in its given pipeline. In this way, faas-flow takes advantage of many rich functionalities available on OpenFaas.

- **Design:** Faas-flow's design principles are set based on following written goals: leveraging the OpenFaas platform, not to violate the notions of FaaS function and providing flexibility, scalability and efficiency

- **Yet another OpenFaas function:** Faas-flow is deployed just like any other OpenFaas function, which is crucial for perfomance reasons as it allows faas-flow take advantage of OpenFaas's functionalities.

- **Isolation:** Faas-flow follows the adapter pattern, where the adaptee is the pipeline's functions and the adapter if the faas-flow function. Any type of composition or configuration that changes the pipeline behaviour is done in the faas-flow function

level and needs no modification to be made in the rest functions. This logic promotes reusability and flexibility.

- **Event driven iteration:** Faas-flow leverages the OpenFaas platform which uses Nats for event delivery. As a result, faas-flow's runtime is event-driven designed. Node execution starts by a completion event of one or more previous nodes. A completion event denotes that all the previous nodes have completed. With events, faas-flow asynchronously executes all nodes by iterating over and over till no more execution happens.

- **Coordinating key-value store:** Faas-flow operates as a distributed system and distributed systems need a centralized service for successful coordination. Execution state and intermediate data are saved thus in any external synchronous key-value store.

- **Role:** As discussed above, this thesis is all about deploying the appropriate instace of a serverless workflow in order to achieve optimal perfomance and utilization. This deployment cannot happen without a tool that would make all functions of an application instance communicate and exchange data one another. Faas-flow makes a perfect match for our use case, as it orchestrates the application instance in an really effective and efficient manner. Specifically, faas-flow provides elasticity and transparency in the construction of a workflow because it operates as a white box system. This is crucial when optimization goals come in discussion due to the fact that if we want to step up the level of serverless workflow execution, we need to dig deeply into all possible paths.

4.2.4 Custom Runtime Engine

Although *Faas-Flow* is a robust framework for OpenFaas functions orchestration, it is developed to serve only synchronous functions communication and not asynchronous. This creates no room for parallel execution and thus no accelaration could be achieved when executing a workflow instance. As a result, we chose to develop a custom runtime engine for this thesis that enables function invocations of both kinds. The engine is written in Python and employs the *subprocess* python module that handles command line calls over the OpenFaas framework. These calls include async function requests, polling logs of OpenFaas components and functions intercommunication. The engine is fed with an input request regarding the video used for inference and outputs the end-to-end latency of its execution. We can accelerate the execution by scaling the Queue-Workers but no change needs to be made in the engine as it is Queue-Workers agnostic and is able of utilizing all Queue-Workers existing in the cluster at a certain moment.

4.3 Description of Serverless workflow and Interference micro-benchmarks

Latest use cases of serverless except from simple event-driven functions, include machine learning training and inference, video processing and other complex workloads which typically consist of multiple stages and require intermediate results to be shared between tasks. This seems odd at first, when considering the stateless nature of FaaS functions, but it is proven that in many scenarios the abstractions offered to the user by serverless computing could be drastically beneficial. Not all developers are familiar with provisioning clusters and managing distributed systems efficiently. Not to mention the low cost services provided by serverless computing as it strongly forwards optimal utilization. Towards this vision, we chose a complex enough video analysis inference workflow to analyze and examine its behaviour when deployed to private infrastructure under all shorts of conditions and constraints.

4.3.1 iBench

In order to apply specific and constant pressure on our cluster of VMs, we used the iBench suite. iBench provides micro-benchmarks which can simulate multiple-level stress to machine resources such as the CPU, caches and memory bandwidth. We fully utilized those benchmarks and created artificial pressure on our machines so as to analyze in depth the relation between cluster's state and application's granularity. Each micro-benchmark was deployed as a Deployment object in our Kubernetes cluster. Specifically, in this thesis we use the following micro-bencmarks:

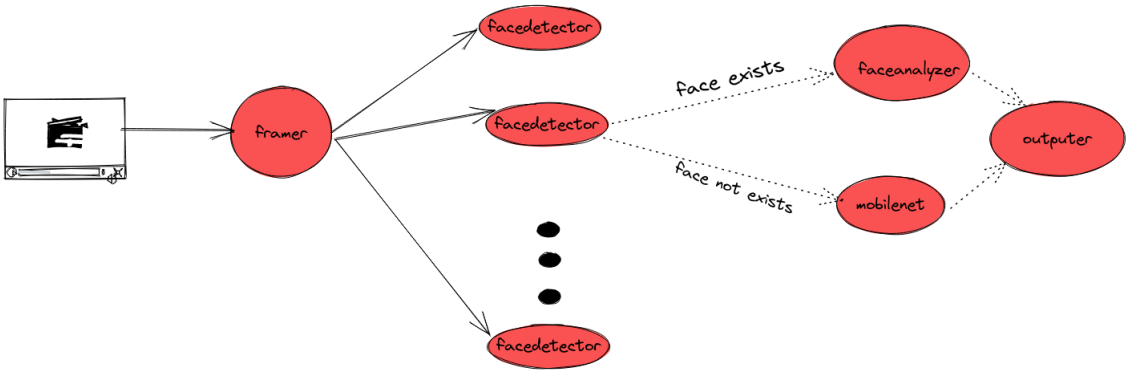
- L3 cache pressure
- CPU pressure
- Memory Bandwidth pressure

4.3.2 Serverless Workflow

From a higher level of abstraction, our application's workflow processes a video file (mp4) by extracting frames and inferencing those frames with models to detect and analyse a human face or a random object.

When the workflow receives at its input a video file, it will extract frames from it based on parameters set previously by the user. At the next state, each image captured is passed to a face detection filter that forwards each frame depending on whether it contains a human face or not to a face analyzer model or an object recognition model respectively. The last stage aggregates the results for all frames processed and writes them at once in a text file that is uploaded to persistent storage or creates online notifications accompanied with the respective result and image to a public endpoint.

Having coded these functions, we needed a component/framework to orchestrate their holistic execution. Our first attempt was to use the custom function orchestrator, 4.2, we developed for OpenWhisk functions but due to OpenWhisk's incompatabilities with



Σχήμα 4.4: *Version1 architecture visualized.*

Kubernetes this path was aborted. The next step was to migrate from OpenWhisk to OpenFaas, a process that required minimum overhead effort. The interesting part with this migration was the utilization of the aforementioned OpenFaas framework, faas-flow [6].

Using *faas-flow* we managed to created our workflow and execute in various versions. Below we present four versions of our application workflow, where the former has the most granular architecture and is named as *Version1* while the latter is the most coarse-grained one and is named as *Version4*. All versions are handled by wrapper functions provided by the faas-flow framework. The functions that constitute these versions are OpenFaas functions and are written in Python, the most popular language for machine learning and image processing. The faas-flow wrapper functions are written in Go.

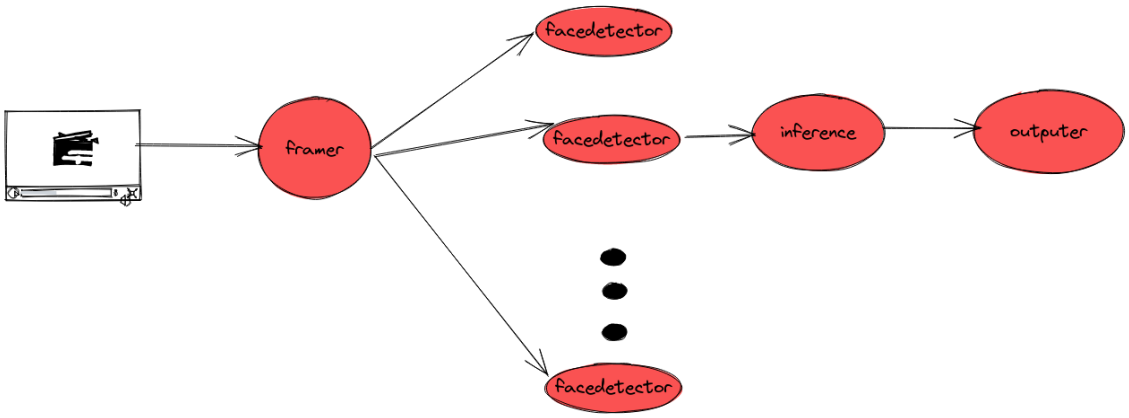
Four application versions are presented below, along with their inner functions.

Version1

The fully decomposed version comprises of five different functiosn as shown in figure 4.4, the most of all cases. Dotted lines represent parralel execution steps while solid lines stand for sequential execution.

- **Framer:** Reads a video and depending on the user input parameters cuts it into a collection of frames. The user provides the starting and the ending point of the video's duration between those the frames are going to be extracted. The user request also contains the interval between two successive frame extractions. When all desirable frames are gathered, *framer* sends all captured frames to a MinIO bucket for temporal save. The frame extraction process is done within the OpenCV library which is the most appropriate for this particular objective.

- **FaceDetector:** This function is responsible for processing each frame extracted in the previous step. It examines for each frame created by *framer*, if it contains a human face or not. If the answer is *yes*, it forwards the frame to the *faceanalyzer* function. If the answer is *no*, it forwards the frame to the *mobilenet* function. Alike the frame extraction, the face detection process is implemented using the OpenCV library.



Σχήμα 4.5: *Version2 architecture visualized.*

- **FaceAnalyzer:** Runs an emotion analysis on frames that contain human faces and returns whether the contained face is Angry, Disgusted, Feared, Happy, Neutral, Sad or Surprised. The inference model in this stage is loaded from pretrained weights that are saved in a json file. The model is available via tensorflow.

- **Mobilenet:** Classifies the frames that do not contain faces to a wide variety of objects. The model used here is the vanilla edition of ResNet50 [42] offered by tensorflow.

- **Outputer:** Aggregates the results of all frames processed and writes them to a text file that is uploaded to a MinIO bucket.

- **Version1-wr:** This is the wrapper function offered by faas-flow. It handles all communication and synchronization between the rest functions so as the execution of the workflow to be successful. It is written in Go, behaves exactly like any other OpenFaas function and uses as mentioned above MinIO and Consul for data saving. Specifically, the wrapper sequentially executes the *Framer*, meaning that it would not forward already extracted frames until all desirable frames are collected. After that, it applies *FaceDetector* to the frame collection in a parallel fashion, aggregates the results and sends them dynamically to *FaceAnalyzer* / *Mobilenet* respectively with the aid of a conditional branch offered by faas-flow. Finally, after a second aggregation process, calls the *Outputer* to upload the final text file to a MinIO bucket.

Version2

The second version unifies the frame inference models in a single compact function. Version2 is consisted of four functions as shown in figure 4.5 and represents the next level of granularity after Version1.

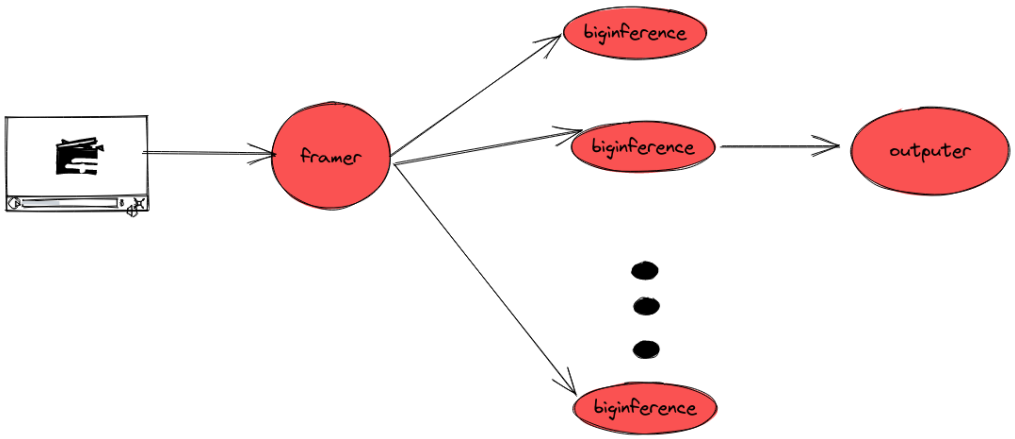
- **Framer:** Defined in 4.3.2

- **FaceDetector:** Defined in 4.3.2

- **Inference:** It is a compact function with a dual role. It actually implements both *Mobilenet* and *FaceAnalyzer* roles as mentioned above in *Version1*. The *Inference* function comprises the same software dependencies as *Mobilenet* and *FaceAnalyzer*.
- **Outputer:** Defined in 4.3.2
- **Version2-wr:** This is the wrapper function for Version2 which as in *Version1* starts with a sequential frame extraction process, continues with parallel face-detection for all frames but then the dynamic conditional branch is skipped, as both kinds of frames (face,no-face) are passed to the inference function which executes the appropriate operation. Lastly, the outputer writes the text file which is going to be uploaded to a MinIO bucket.

Version3

The third version fully merges the face-detection process with the inference processes in a single function. It is consisted of three functions as shown in figure 4.6 and represents the next natural step of making Version2 more granular.

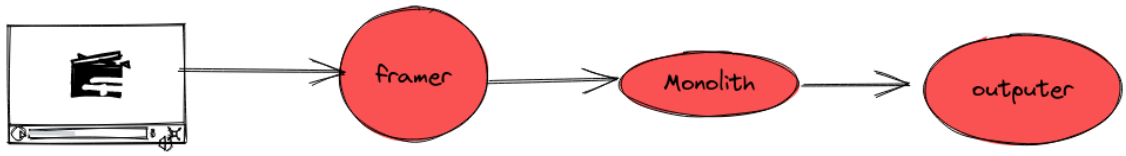


Σχήμα 4.6: *Version3 architecture visualized.*

- **Framer:** Defined in 4.3.2
- **Biginference:** This function fully implements the facedetection and inference processes.
- **Outputer:** Defined in 4.3.2
- **Version3-wr:** Version3's wrapper orchestrate version3 execution in an even simpler way by invoking the framer function and after that passes the frame collection to the biginference function in a parallel way, which both detects faces in the extracted frames and applies the inference models as needed.

Version4

Version4 is consisted just by two functions, as visualized in figure 4.7.



Σχήμα 4.7: *Version4 architecture visualized.*

- **Monolith:** This function encapsules the whole application logic that is distributed in *Framer, FaceDetector, FaceAnalyzer, Mobilenet* functions of the Version1 architecture. Moreover, the *Monolith* executes the application logic sequentially, even the parts that in *Version1* are handled in parallel pattern, due to its inability to take full advantage of faas-flow's potential.

- **Outputer:** Defined in 4.3.2.

- **Version4-wr:** Another faas-flow wrapper, one that does not utilize parallel execution at all. It simply invokes the monolith function which sequentially executes the required processes emulating the previous mentioned versions.

4.4 Impact of Granularity on the Workflow's Perfomance

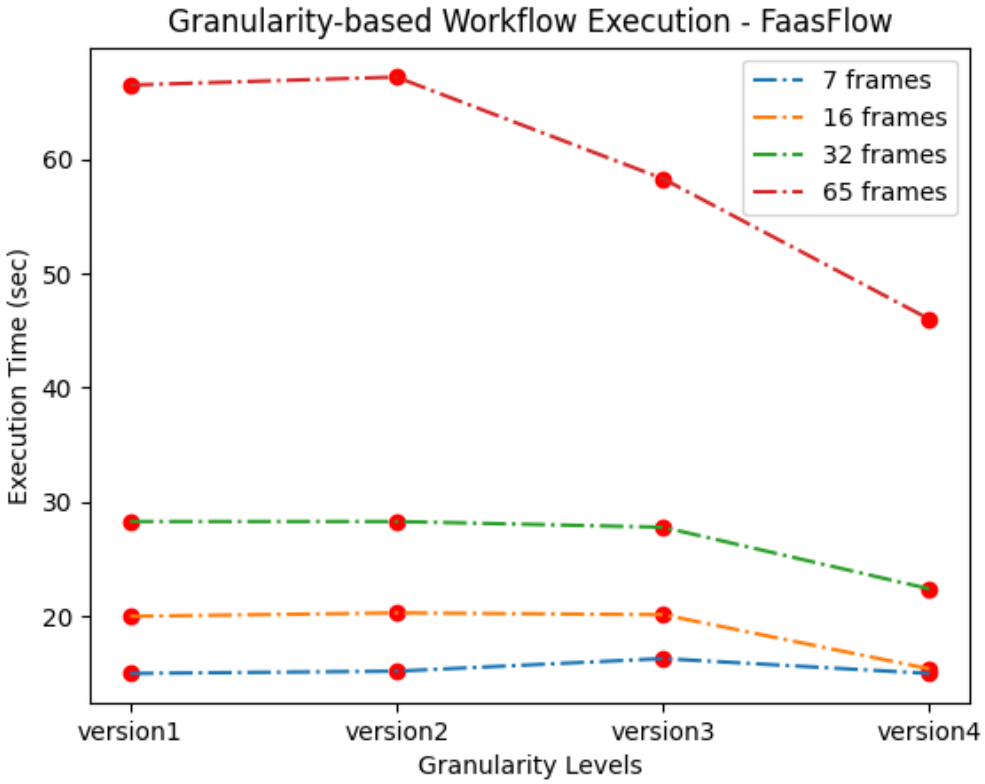
A serverless application is usually consisted of multiple functions that are chained one another to form a DAG which is going to execute an application's workflow. The level of granularity that characterizes a workflow influences the data transfer needs among the functions and thus plays a significant role in perfomance. This influence is enhanced by the fact that contemporary serverless platforms execute each function by deploying a different container instance. For example, when two functions are merged in a single one so as to reduce application's granularity, some virtualization and data exchange overheads are avoided but this comes at the cost of decreased capability of parallelizing functions' execution and increased limitations in function placement. E.g., as proposed in ExCamera [43] by Fouladi et al., it is common to attempt and split a processing workflow into tiny tasks that could be executed in parallel. This tradeoff depends on a wide variety of parameters that need to be considered when designing the architecture of a desired workflow. For example, having relatively smaller functions to manage, the cloud service provider is closer to maximizing resource utilization.

Especially, Ashraf Mahgoub et al.[12] highlight the fact that in data-intensive applications passing data through remote storage, which is the most frequent approach nowadays, takes over 75% of the computation time. Consequently, granularity level of a workflow and function intercommunication affects directly its end-to-end perfomance.

In order to examine the impact of granularity on the workflow's perfomance, we firstly tested four levels of application granularity with the aid of the *faas-flow* framework and secondly two levels of application granularity executed through our custom Python runtime engine.

4.4.1 Faas-Flow Runtime Approach

As mentioned above, four versions of our application were developed, where each one represents a different level of application granularity. Specifically, *Version1* is the most granular, *Version2* is less granular than *Version1* but more than *Version3*. *Version4* is essentially a monolith as it merges all *Version1's* functions in a single function named *monolith*. In figure 4.8, the average execution latency of each version is displayed. All versions are invoked with the same 10-minute long video and are requested to process a different amount of frames each time. The input sizes are yet again 7, 16, 32 and 65 frames. It is obvious that the more coarse grained a workflow is, the faster it is been executed. Indeed, in the largest input (65 frames) where the more deltas are observed, *Version4* is 30% faster than *Version1*. Furthermore, it is quite interesting that *Version2* is not faster than *Version1*. It seems that making the branch desicion does not put any signifcant overhead on the overall execution time. In the rest three cases, only *Version4* looks to differentiate from the others and that is due to the fact that 7, 16 and 32 frames are not too many for the more granular versions to lose much ground to the coarser grained ones.

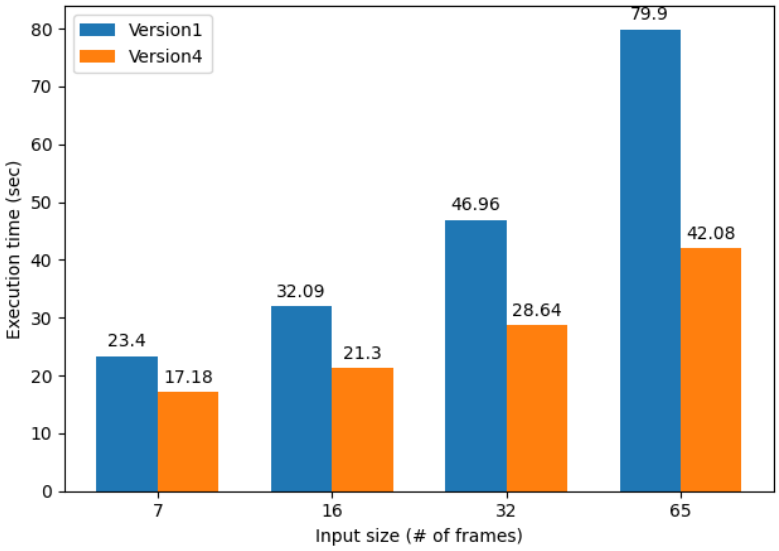


Σχήμα 4.8: *Versions' average latency for 65 frames processed*

4.4.2 Custom Runtime Approach

In a second phase of experiments, we used our tailor-made runtime for OpenFaas functions in order to investigate furthermore the granularity's effect on the execution of

the aforementioned machine learning workflow. We chose to compare the most granular version with the monolithic one in order to test how great a factor faas-flow is on executing the same application with a different granularity level. As clearly shown in 4.9, the deltas between *Version1* and *Version4* are more significant in all input sizes. In particular, for the 65 frames input, the delta reach 46.5% whereas in the faas-flow case the corresponding rate is 30%. Consequently, a certain conclusion could be drawn: independently of the runtime engine, a workflow's granularity would affect its execution time regardless of its simplicity. Thus when deploying a serverless workflow on cloud premises, it is crucial to handle this aspect carefully so at to offer the best possible QoS to the user.



Σχήμα 4.9: *Version1 and Version4 comparative executions at the w01 node.*

4.5 Impact of Interference on the Workflow's Perfomance

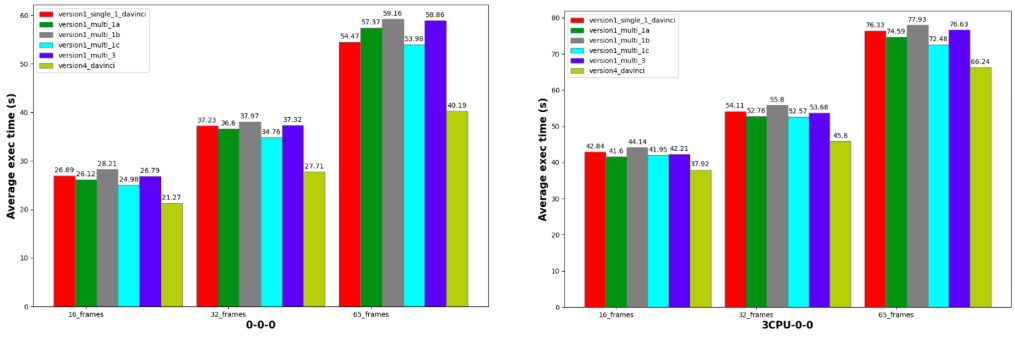
In this section, we evaluate the behaviour of *Version1* variations and *Version4* when intereference is applied on the cluster nodes. It is useful to acquire knowledge of various execution profiles under resource stressed conditions so as to offer the best possible quality of performance. For example, getting to know if our application is "hungry" for CPU resources, then deploying a monolithic version of it in a node that also lifts third party workloads would be definitely an inefficient decision. In the first and second part, using the faas-flow framework and the custom runtime we are targetting to investigate the way a interferenced granular workflow version competes with a monolithic one. In the third part, via the custom runtime, we further examine the sensitivity of the functions to resource interference. Lastly, in the fourth part, we are conducting experiments through our tailor-made runtime in order to check if scaling the *Queue-Worker* component of OpenFaas accelerates the workflow execution and how this acceleration behaves under inteference.

4.5.1 Interference impact with FaasFlow

As described in section 4.1, with three different kinds of pressure, we could simulate cluster interference. We created a YAML file for each possible pressure deployment, e.g. cpuPressure1.yaml in order to apply CPU-pressure to the first node. Moreover, by configuring each pod's replicas we are able to scale up or down the desired pressure. The aggregated results are shown in figures 4.10, 4.11

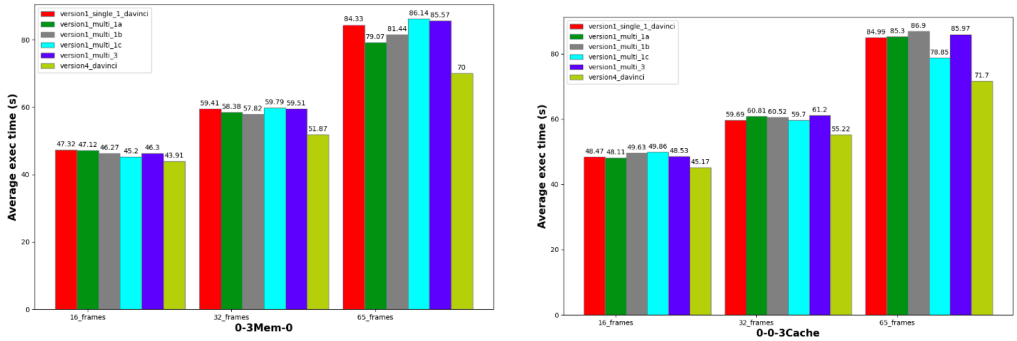
Functions	Singlenode	Multinode 1a	Multinode 1b	Multinode 1c	Multinode 3	Version4
Framer	w01	w01	w01	w01	w01	w01
Facedetector	w01	w02	w02	w01	w01	w01
Faceanalyzer	w01	w03	w02	w02	w01	w01
Mobilenet	w01	w01	w01	w02	w02	w01
Outputer	w01	w02	w02	w02	w02	w01
Wrapper	w01	w01	w01	w02	w02	w01

Πίνακας 4.2: *Configurations of functions placement*



(α') *Zero Interference* (β') *CPU Interference*

Σχήμα 4.10: *Version1 configurations and Version4(1)*



(α′) *Memory Interference* (β′) *L3 cache Interference*

Σχήμα 4.11: *Version1 configurations and Version4(2)*

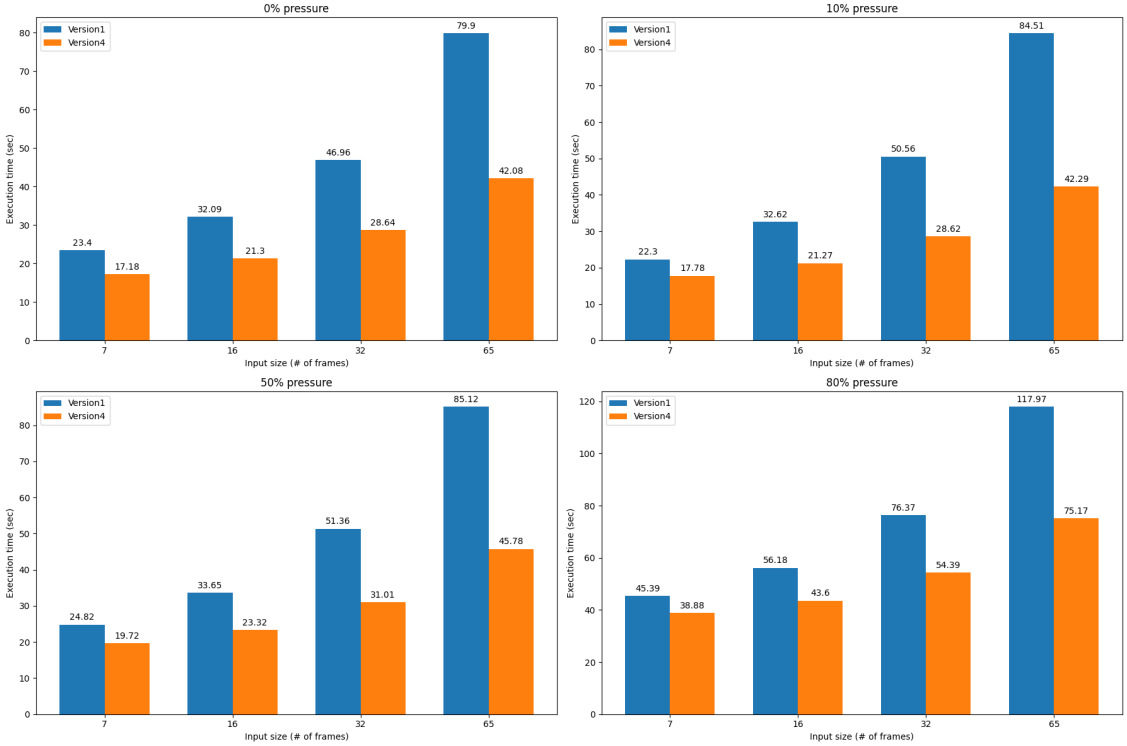
We examined the results from various aspects. First of all, the input size seemed to under-influence the configurations' competition regarding the minimum time of execution. The intuition behind this is the fact that the long-lived function, the *Framer*, was located at PM1 in all of the experiments. The virtual machine hosted at the PM1 server is the best performing one among all. Besides that, moving the other functions that are short-lived across the cluster did not make any particular difference in the configurations' comparative latencies.

Furthermore, *Version4* was at all times the faster configuration, which was utterly anticipated as it generates the least amount of traffic and thus operates without the functions' intercommunication. When pressure was applied at specific nodes, *Version4*'s gap to other configurations was smaller though, due to the fact that other configurations had some of their functions deployed on non stressed nodes in comparison to *Version4*'s single function (*Monolith*) that was placed in a highly intereferenced node.

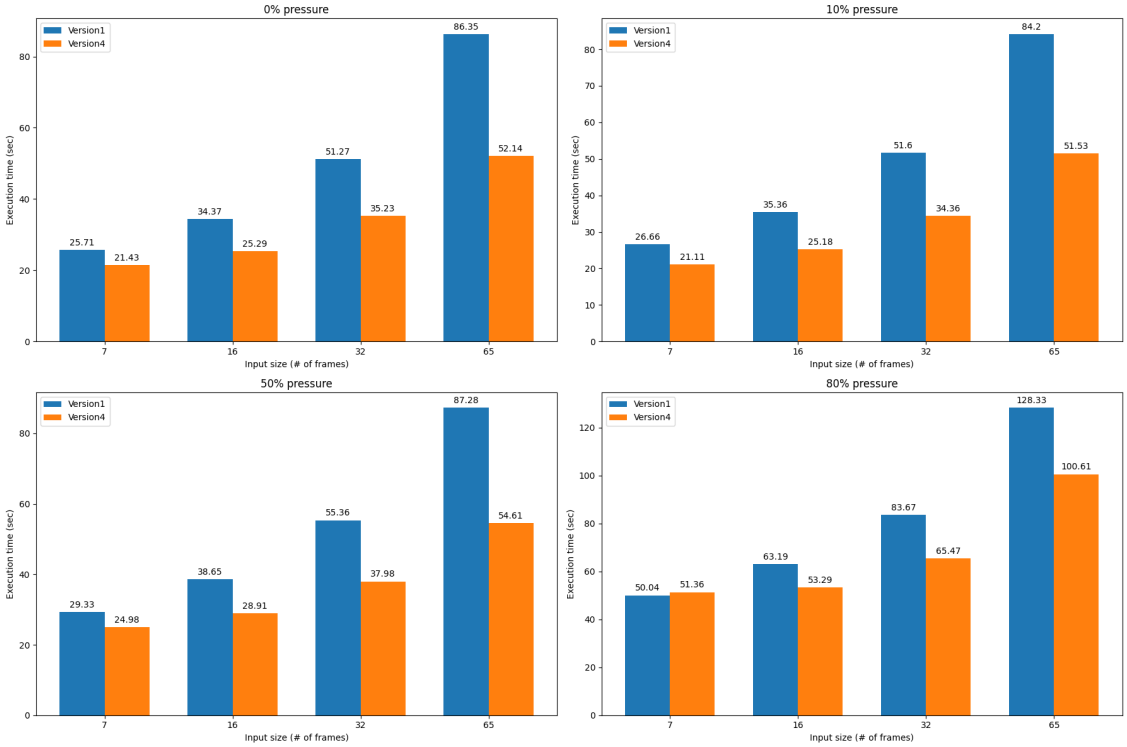
Out of the three pressure scenarios, it was the L3 cache pressure the one that allocated the largest amount of resources and therefore produced the greatest level of intereference. In other words, workflow executions under l3-pressure lasted longer than the rest.

4.5.2 Interference impact with Custom Runtime

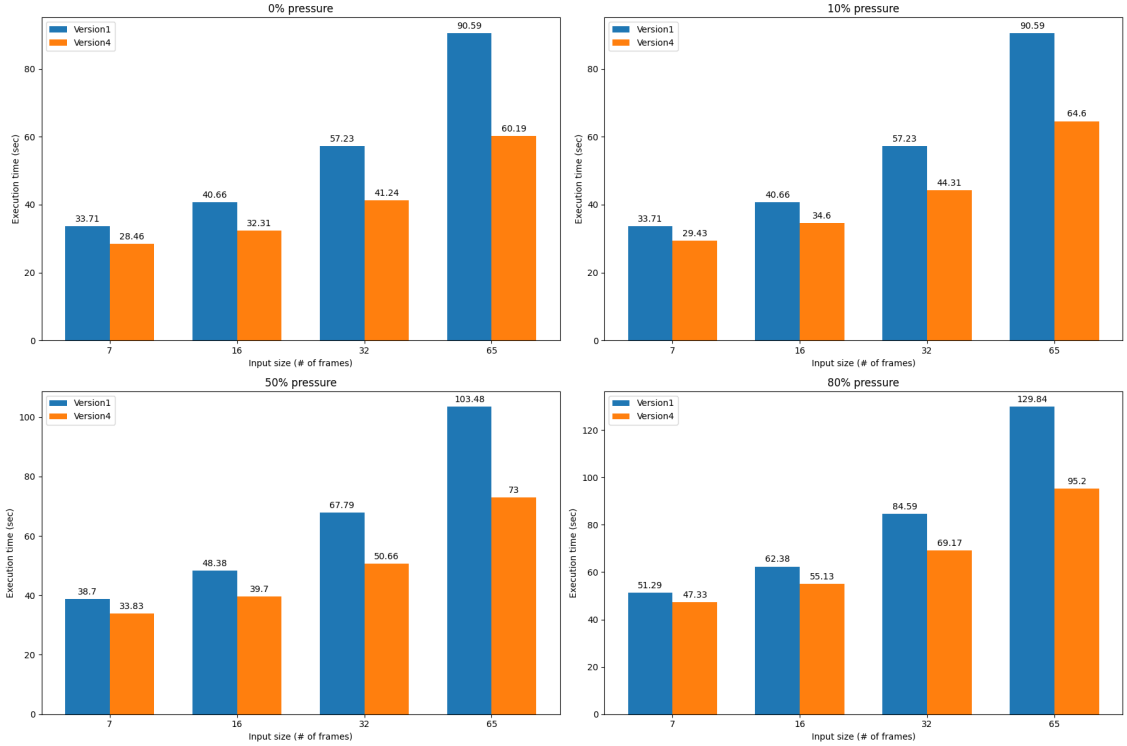
Moving on to a next series of experiments, we deployed *Version1* (holistically) and *Version4* in a single node. More specifically, in the direction of acquiring a clear combined view of heterogeneity's and interefence's impact, for each cluster node (w01, w02, w03, w04) we ran the same experiments with four discrete levels of CPU pressure: $0\%, 10\%, 50\%, 80\%$. All information taken from this process is shown in figures 4.12, 4.13, 4.14, 4.15.



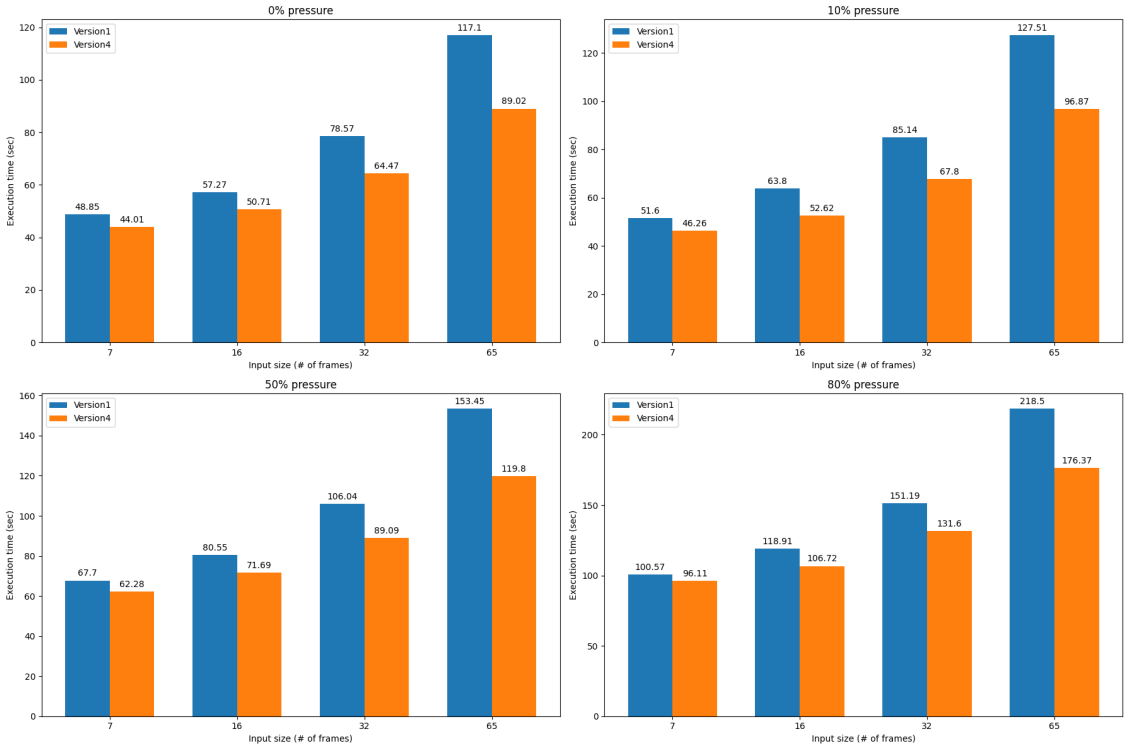
Σχήμα 4.12: *Version1 and Version4 comparative executions at the w01 node.*



Σχήμα 4.13: *Version1 and Version4 comparative executions at the w02 node.*



Σχήμα 4.14: *Version1 and Version4 comparative executions at the w03 node.*

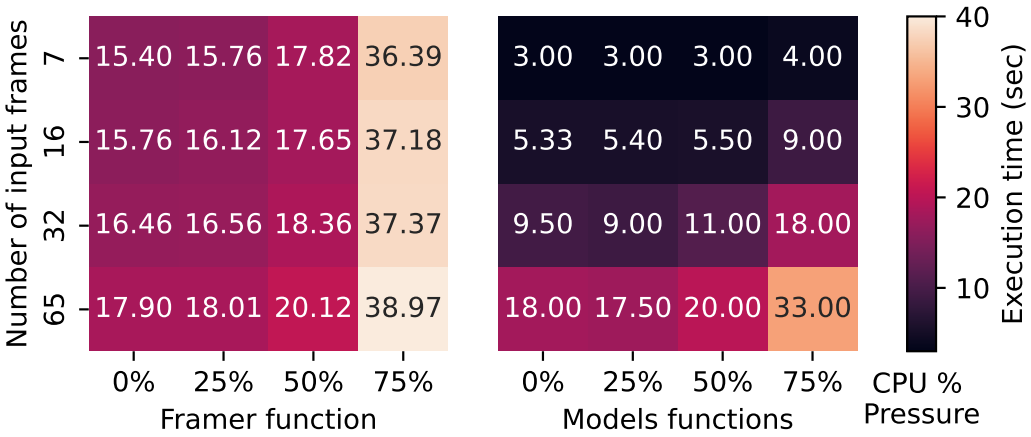


Σχήμα 4.15: *Version1 and Version4 comparative executions at the w04 node.*

Regarding the presence of iBench applications allocating the machines' resources, we draw the below conclusions:

- The greater the pressure from interference, the smaller the latency gap between Version1 and Version4
- The delta between having 0% and 80% interference results in 33%, 33%, 30% and 47% total slowdown in an instance's execution for w01, w02, w03 and w04 nodes respectively
- The 10% and 50% levels of interference are nearly "absorved" by the w01, w02 and w03 nodes because they leave unallocated, in the worst case, half of the machine's cores that are enough for executing this specific workload with multithreading. The 50% level in the w04 node allocates two out of four cores, thus with two cores available, multithreading could not be efficiently performed.

In order to characterize the sensitivity of the functions to resource interference, we spawn different amount of cpu micro-benchmarks from the iBench suite, which increase the computational load of the underlying VM. As depicted in figure 4.16, our pipeline presents great performance variability w.r.t. CPU interference, that reach up to 57.6% in the Framer case and up to 47.2% worse performance compared to isolated execution. Moreover, the imposed degradation does not present a linear relationship with the interference load, with CPU pressure levels below 50% imposing minimal performance degradation to all the functions.



Σχήμα 4.16: *Impact of interference on serverless functions*

4.6 Impact of Heterogeneity on the Workflow's Perfomance

4.6.1 Faas-Flow Runtime Approach

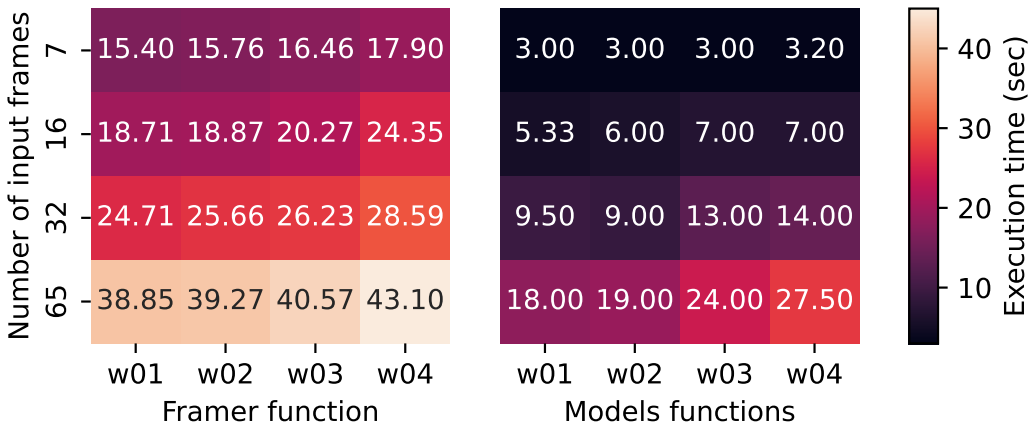
For the shake of examining heterogeneity's impact on the performance profile of our workflow, we spreaded *Version1*'s functions across the cluster and so five different configurations of function placements were created which are shown in table 4.2. More specifically, we measured the execution times of those configurations with three different input sizes: 16, 32, 65 frames so as to obtain safer insights and check whether a correlation between time and input size actually exists. The results are again presented in figures 4.10 and 4.11.

4.6.2 Custom Runtime approach

Regarding the presence of heterogeneous machines in a cluster, we shall look into the way each one behaves before making deployment decisions that are willing to serve a user's request strictly. By examination of the figures 4.12, 4.13, 4.14 and 4.15 by looking through the heterogeneity aspect, we conculde to the following:

- It is obvious from the figures that the w01 node is the most dominant in terms of perfomance regardless of the input size. w02 node comes second, while w03 and w04 nodes come third and fourth respectively.
- If we focus on the largest input size (65 frames), we get to view that the delta between *Version1* and *Version4* declines linearly with the machines' "strength". On the w01 node, this delta peaks at 45% while it is up to 39%, 33%, 23% in the w02, w03, w04 nodes respectively (under no-interefence circumstances)
- This also happens more or less in all input sizes, but it is in the largest one that peaks.
- All functions could be executed with multiple threads in a machine's available cores. This is the main reason that in the w04 node, which is employed only with four cores, the execution times are even bigger than the rest as the workflow's multithreaded profile is not utilized enough.

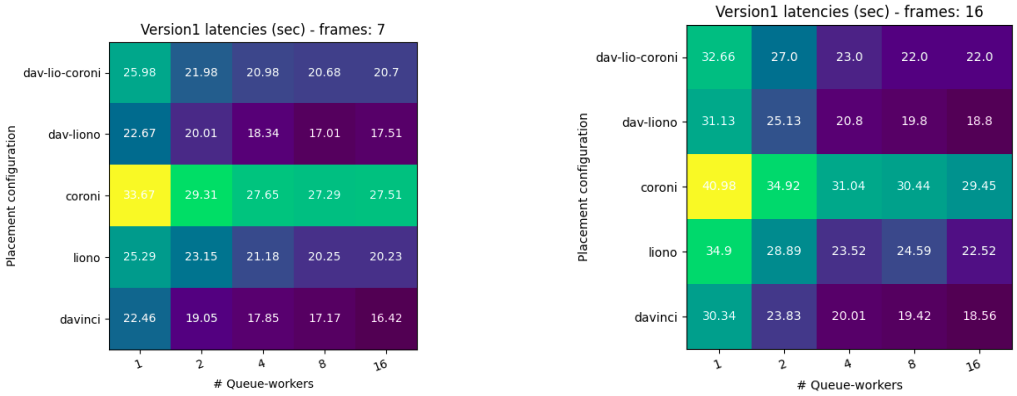
Figure 4.17 shows the performance variation of the examined functions, w.r.t. hardware heterogeneity. For the *Framer* function we find deltas with a maximum value of 23% and a minimum of 10% performance variation in the 16-frames and 65-frames deployments respectively. Moreover, for the ML-models functions, the measured deltas have a maximum value of 34% and a minimum of 5% variation respectively. Overall, we observe that the impact of resource heterogeneity becomes more perceptible as the number of frames increase, due to the accumulated computational burden to less powerful hardware resources. Also, despite the variation in the available resources (vCPUs, memory) per VM, *w01* and *w02* provide the overall best performance, due to the lack of vertical-scaling mechanisms within the functions.



Σχήμα 4.17: *Impact of heterogeneity on serverless functions*

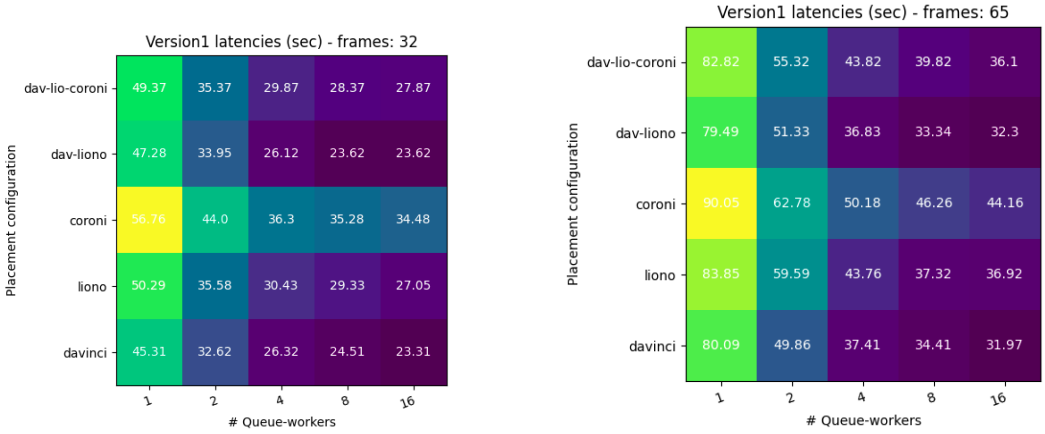
4.6.3 Queue-Workers and Accelerated Execution

As described in 3.3.5, OpenFaas comes with the capability of handling asynchronous invocations of serverless functions. By these means, a serverless function, i.e., a Docker container, can process n requests at a time, where n is the number of queue-workers deployed in the cluster. So, if the workflow is parallelizable a lot of time could be saved by increasing the queue-workers' replicas. Therefore, we inspected thoroughly the way this component affects the execution time of our parallelizable workflow with regards to different function placement configurations and input sizes.



(α') *7 frames* (β') *16 frames*

Σχήμα 4.18: *Version1: placement configurations and number of replicas*



(α') *32 frames* (β') *65 frames*

Σχήμα 4.19: *Version1: placement configurations and number of replicas*

As shown in figures 4.18 and 4.19, we conducted the same experiment with four different input sizes: 7, 16, 32 and 65 frames. For each experiment, the functions were placed in five distinct ways:

- davinci: all functions placed at the w01 node
- liono: all functions placed at the w02 node

- coroni: all functions placed at the w03 node
- davinci-liono: *framer* placed at the w01 node, rest functions placed at w02
- davinci-liono-coroni: *framer* placed at the w01 node, *facedetector* at the w02 node and the rest functions at w03

Several conclusions were drawn after this series of evaluation. Firstly, the proportional performance acceleration achieved was minimum for the smaller-sized input cases. This makes perfect sense when taking into account the fact that parallelizing a larger instance creates more space for latency minimization. More specifically, a proportional improvement of 60% was the maximum accelaration (davinci placement along with the 65 frames input), whereas the minimum one was observed was at the order of 20% (davinci-liono-coroni placement along with the 7 frames input). Moreover, by employing the queue-workers capability one can manage to shorten the latency gap between the smallest and the largest input size execution instance. With a single queue-worker, this gap could be up to 58 seconds while with 16 queue-workers deployed it is lessened down to 15 seconds. Subsequently, it is a minimization of 74%.

An important aspect of this acceleration process should also be mentioned: A queue-worker replica is deployed as a container in a Kubernetes cluster. This container allocates a tiny amount of a node's available RAM, most of the times not more that 10 MB. The fact that RAM usage is not affected by scaling this component and one can achieve execution accelaration without creating a resource contention problem makes this method extremely valuable.

4.7 Discussion

As stated above, scheduling a serverless workflow dynamically over cloud infrastructure is a complex task due to uncertainty and its probabilistic nature. The expected performance might be affected by the application's granularity, cluster's inteference or worker nodes' heterogeneity. In order to tackle such a challenge, one needs to utilize state of the art methods like deep reinforcement learning so as to bridge the gap between static and dynamic scheduling. Deep reinforcement learning is powerful in solving big data problems, where handling a great amount of data should be done under low latency times. Other rule-based approaches fell short when followed for this kind of problems and that is because of the inability a set of rules, generated by humans, present to fully cover all possible scenarios of operation. Furthermore, designing logic-based rules for such a problem demands a lot of time and effort of measurements that could be skipped if a deep reinforcement learning approach is adopted. It is a common practice embracing machine learning for reducing manual intervention for scheduling workloads of all types. [44] The action-space and the state-space of this category of problems are not discrete and this is a main concern while trying to solve problem instances and server user requests effectively.

Κεφάλαιο 5

Dynamic Scheduling of Serverless Functions

This chapter includes our work on designing a dynamic scheduling framework for managing video analytics pipelines in serverless infrastructures. The proposed framework exploits low-level performance monitoring events to identify interference phenomena on the underlying cluster and combines this information with user-defined QoS requirements in order to regulate end-to-end latency of video analytics pipelines, through horizontal-scaling and migration of the pipeline's functions. Our solution manages to successfully orchestrate the functions under different dynamic conditions, i.e., system-pressure fluctuations and dynamically changing QoS criteria.

Specifically, we are interested in the following question: What is the "best" way of serving a user request upon the deployed workflow so as to meet the user-defined QoS while "sacrificing" minimum resources possible. Our intention of providing minimum resources is primarily to help the underlying system become highly respondent to heavy load and secondly to create more serving "space" for future high-demanding requests.

In Section 5.1, the design principles of the DRL-based scheduler are presented, while in Section 5.2 a detailed analysis on the framework's architecture and specifications is provided.

5.1 Design Principles

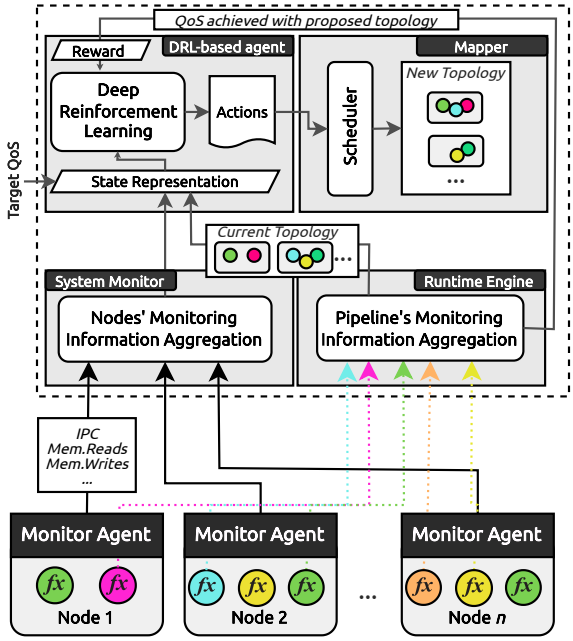
The DRL-based dynamic scheduler is designed using the following design principles:

- Appriori knowledge of an application's execution profile is not required. This means no offline time needs to be spent in order to profile the application that is going to be scheduled. This is a real world scenario for unknown applications that happen to be deployed on cloud premises because this kind of information could not be obtained.
- Scheduling decisions of the DRL scheduler are granular and based on the underlying infrastructure. The deep network needs to be aware of the nodes that serve the cluster upon which it operates.
- When the cluster's state changes, the DRL scheduler in short time is able to reevaluate and place the functions accordingly in order to achieve the user-defined QoS, if possible.

5.2 Architecture and Specifications

The DRL scheduler is depicted in figure 5.1 and it consists of:

- *System Monitor* which monitors and collects metrics that are going to represent the current system's state. The current system state is fed into the DRL-based agent as input for further processing.
- *DRL-based Agent*, which reads the system metrics and calculates an action to be performed regarding the user-defined QoS.
- *Runtime Engine*, that given the functions' placement is responsible for executing the workflow independently of the location of any function. During the training phase, the measured time of executing a single workflow instance is passed to the DRL-based agent in order to calculate its reward upon the decision it made in the previous step.
- *Function Mapper*, that executes the DRL-based agent's action and maps a function to a specific node.

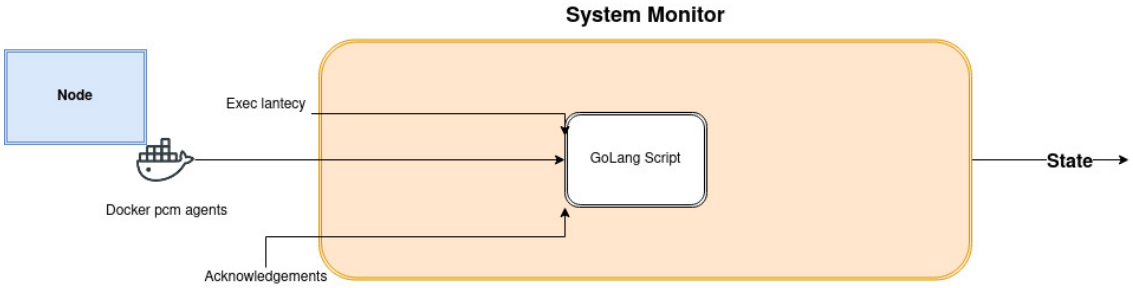


Σχήμα 5.1: *DRL Scheduler Framework Overview*

5.2.1 System Monitor

The System monitor is responsible for gathering the required data for system-related state representation, i.e., low-level performance hardware events. It aggregates system-, socket-, and core-level information from distributed PCM monitoring agents (MA). All metrics extracted by the MAs are improted to an InfluxDB [45] instance where they are stored in a 500ms interval, in batch mode for increased throughput on a time-series database. A GoLang script, afterwards, retrieves the stored metrics and serves them to the System

Monitor. For the rest of this paper, we employ the following five performance counters: Instructions Per Cycle (IPC) which is an approximate indicator of the performance of the processor and gives insight information of the performance of the deployed functions. Memory Reads/Writes that depict the access patters from/to the DRAM memory, which is considered a major bottleneck in modern server systems. The amount of memory reads and writes performed in a time period could be a highly accurate indicator of a system's load. L3 Cache Misses lead to increased memory reads/writes, thus it is critically representative of performance. C-States (C0, C1): For energy saving reasons during CPU's idle state, the CPU could be forced to enter a low-power mode. Each core has three scaled idle states: C0, C1 and C6. C0 is the normal CPU operating mode, where the CPU is 100% active. The higher the C index is, the less activated is the CPU, which differentiates the utilization ratio A concrete overview of the system monitor is offered in figure 5.2.



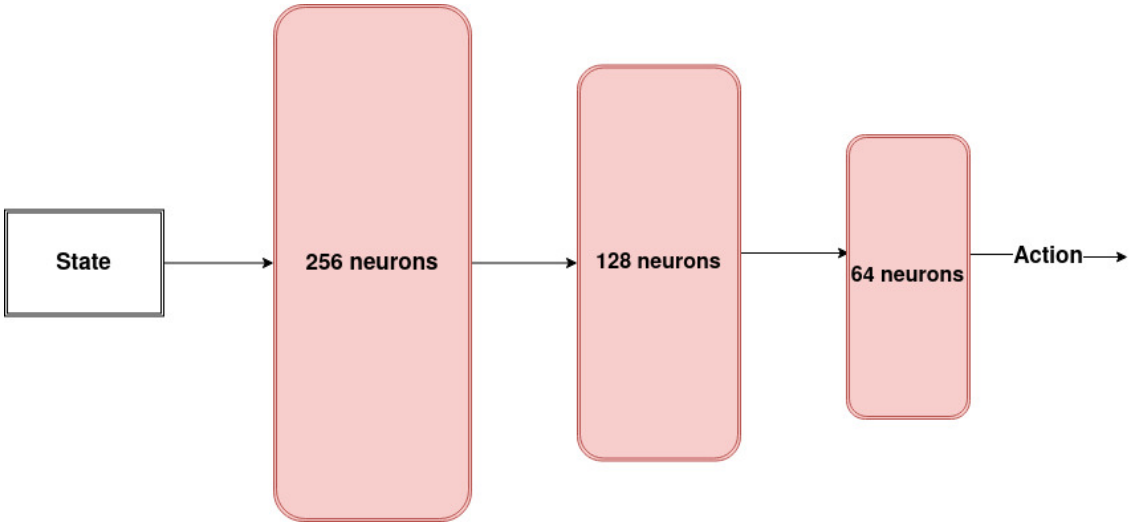
Σχήμα 5.2: *Architectural Overview of the System Monitor*

5.2.2 DRL-based Agent

The Deep-Q-Network is a reinforcement learning agent that tries to learn the "optimal" decisions based on a given reward function by interacting with the environment. The agent utilizes the exploration-exploitation dilemma during its training, where not only it needs to exploit the "best" solution found so far but also to explore other solutions that may or may not perform better than the current "best". The probability of chosing not the current "best" solution so far is captured with epsilon ϵ. But in this particular setup where the action space is relatively large and complex we use an ϵ that decays over time. Specifically, ϵ starts with exploration-initial-eps and ends up with the value of exploration-final-eps where exploration-initial-eps > exploration-final-eps so as to have a more exploratory start in order to explore thoroughly the action space and later conclude with a more exploitable policy for the "best" found solutions. During training, the agent collects all state data and performs an action which could be generated either deterministically or randomly. Depending on the action's result, the agent receives a reward or a penalty. Essentially, the dynamic scheduler transforms the problem into a Markov decision problem that is solved by a Deep-Q-Network.

The Deep-Q-Network agent is implemented using Stable Baselines 3 [46] while the application's environment is implemented by the OpenAI Gym which is a standard API for reinforcement learning environments [47]. We used OpenAI Gym's API to build a custom environment for our needs. The custom environment encapsulates and wraps

up all the underlying processes of training such as monitoring, rewarding, executing and normalizing.



Σχήμα 5.3: *Overview of the DRL-based agent*

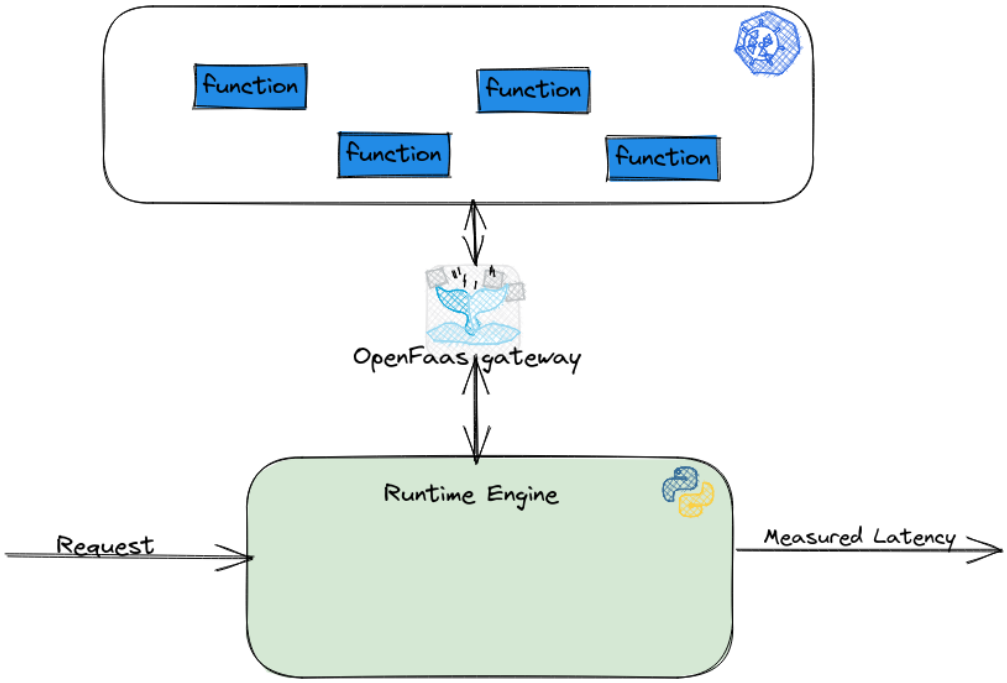
Reward Function

When attempting to solve a deep reinforcement learning problem, the key to be considered is the reward function which determines whether the agent receives an either positive or negative reward for the action it just performed. Identifying a convenient reward function is not an straightforward task and needs excellent understanding of the problem one tries to solve and all of its aspects. The reward function we decided to employ was the result of a series of experimentations that led to the below result:

$$Reward = \begin{cases} \frac{N}{sp} + \frac{Rs}{r} + \frac{L_a}{L_t} \times k_1, & \text{if } L_a \leq L_t \\ \max(-k_2, -k_3 - \frac{L_a}{L_t}), & \text{otherwise} \end{cases} \tag{5.1}$$

The reward function is described in equation 5.1. The incentive behind its construction is the regulation of the execution latency by not violating the latency threshold L_t, while attempting to minimize the number of utilized servers sp(maximum of N), and the replica count r (maximum of Rs), parameters that depict the cost of resource reservations. $K1, k2, k3$ are parameters that can be tuned accordingly, depending to the strictness of the violation penalty ($L_a > L_t$).

Firstly, if the QoS is not achieved and therefore the ratio of *latency* to the *QoS* is greater than 1, the agent should be definitely penalised in order to force it set a top level priority for not violating the desired QoS. For the shake of bounding the value of the negative reward, we cap it by the number k_2. This value was found empirically after trying lots of numbers in the range of (-50, -1). More specifically, when capping with a relative large numbers like -50, we got to view that the agent finds it difficult to explore enough the action space in the earlier training stages. So, the balance was found in the value of -6 for the purposes of this thesis. Secondly, if the ratio of $\frac{L_a}{L_t}$ is less or equal than 1, the desired QoS has been met and consequently the agent should be given a positive reward.



Σχήμα 5.4: *Overview of the Runtime Engine*

But it is wise offering the agent a greater reward if its decision requires less resources or presents more utilization when already having accomplished the desired QoS. In detail, three terms are included to the positive reward case:

5.2.3 Runtime Engine

The Runtime Engine component used here is the one that is previously described in 4.2.4. An overview is provided for a more concrete representation.

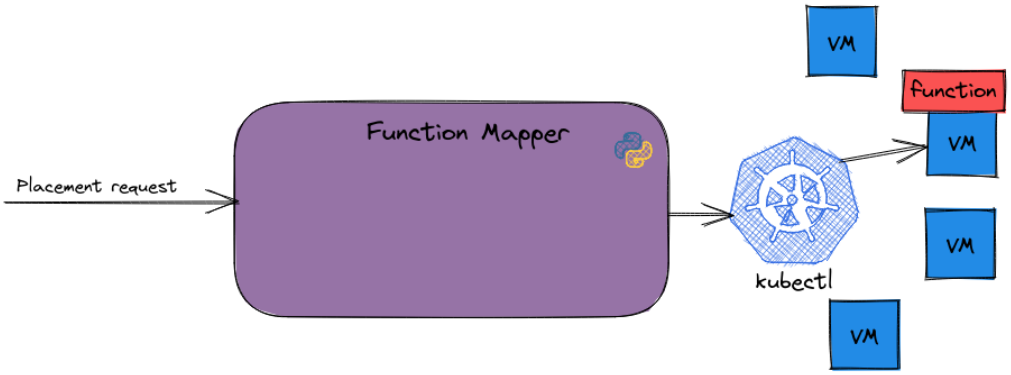
5.2.4 Function Mapper

The Function Mapper component is responsible for deploying the function in the determined by the DQN agent node of the cluster. It is written in python and uses the *subprocesses* module to interact with kubectl in order to apply the corresponding node affinity for the function's container.

5.2.5 Technical Implementation

Critical Measurements

As mentioned above, a wide variety of measurements are happening under the hood. For the end-to-end latency of the workflow instance execution we use the result provided from the runtime engine 4.2.4. The end-to-end latency has a max of 400ms deviation of the actual number because of the polling process it operates on. Furthermore, the system monitor 5.2.1 gathers PCM metrics that are used for the system's state representation.



Σχήμα 5.5: *Overview of the Function Mapper*

Neural Network Parameters

The neural network parameters were tuned through an experimental analysis of the this task by training enough agents that converge to the most valuable end result. We set the minibatch size to 32, meaning that a gradient update happens just after a batch of 32 inputs. Adam is used as the optimizer configured with a learning rate of 0.0025, while gamma (discount factor) is set to 0.99. The target network is updated every 60 training steps in order to mitigate the unstable learning problem. For the exploration-exploitation matter, the initial value of ϵ is set to 1 and over time it decays to the final value of 0.01. The exploration fraction used for this decay is set to 0.2 and it indicates the training period over which the exploration rate is reduced. For the deep neural network, three hidden layers are used with 256, 128, 64 nodes accordingly. ReLU is used for the neurons activation. Finally, in order to simulate the agent's experience at a certain moment we use a buffer with a size of 10^6.

OpenAI Gym Environment

By making use of the OpenAI Gym API, we create a custom python class that inherits the base Gym Environment class. This approach was selected because the base class is quite modular and one can extend it accordingly to the desired problem needs by overriding the constructor, reset and step methods.

Starting from the constructor, we have to define all class variables needed for our environment to operate such as *self.spread, self.replicas, self.actionTexts, self.state*. Moreover, we have to configure variables that are provided from the base class such as *self.actionspace, self.observationspace*. *Self.actionspace* is set to *gym.spaces.Discrete(12)* as the total actions available for the agent are 12, whereas *self.observationspace* is set to *gym.spaces.Box(low=0, high=20, shape=(35,), dtype=np.float64)* as the state vector comprises 35 floats in the range of 0-20.

The step function encapsulates the transition of the system from one state to another. In our case, the step function requests from the system monitor the current state metrics, passes the action to be performed to the function mapper and afterwards through the runtime engine executes the workflow instance and collects its end-to-end latency. Later,

after an 8.5 seconds period, it collects again the system metrics, calculates the reward to be given and transmits the agent to the next state. In the mean time, loggers inside the step function store all information needed for future evaluation of the training phase.

With the reset function, we force the agent's state back to its initial configuration where we need to reset all the state-based variables in our custom environment.

After initializing both the custom environment and the agent(deep neural network), we inject the agent into the environment and trigger the learning process. We chose the location of all functions at start to be on the davinci node.

Κεφάλαιο 6

Experimental Evaluation

In this chapter, we evaluate the results of our proposed framework in various scenarios. In Section 6.1, the conditions under which we conducted our experiments are analyzed, as well as the evaluation criteria we set for this venture. In Section 6.2, we introduce the four DRL-based schedulers we designed in order to examine our scheduling framework from various angles, while in Section 6.3 we are presenting a performance evaluation of the four distinct DRL-based schedulers. A comparative evaluation of all schedulers is provided in Section 6.4.

6.1 Experimental Conditions

A lot of conversation has beed made regarding the early stage this domain finds itself. First of all, we deploy each DRL-based scheduler to the master node of our Kubernetes cluster so as not to add noise and overload the rest worker machines. Each scheduler shares the same responsibility with others: migrating OpenFaas functions from one worker node to another, if necessary, in order to serve a user-defined QoS. We chose two distinct values of QoS to be served, 35 and 26. The certain selection derives from offline profiling of the workflow we are executing. In fact, we would like to examine whether our schedulers could manage converging or not and at what speed, in those topology configurations after appropriate training.

For making the schedulers' task more complex, we are demanding both QoSs to be served under conditions of with and without interference. For generating interference, we used the CPU pressure iBench described in Section 4.3. We simulated four levels of CPU pressure: 0%, 25%, 50%, 75%. If the CPU inteference ratio is equal to x, it means that x% of the node's total cores are unavailable due to resource contention with "third-party" generated workloads. Due to the fact that our machines are heterogeneous and have different number of cores, a 50% in the davinci node (w01) means "4 cores occupied", while a 50% in the coroni node (w03) essentially means "8 cores occupied". Nodes' specifications are analyzed in depth here.

What is expected from the schedulers is to be adaptable to QoS changes or sudden interference. If this is achieved, we pave the path for more elaborate solutions in the future for handling even more complex usage scenarions.

6.1.1 Training events

With a more granular view on the conducted experiments, the scenario we simulated during training is the following:

$$QoS = \left\{ \begin{array}{ll} 35seconds, & \text{if } 0 \leq trainingsteps \leq 300 \\ 26seconds, & \text{if } 300 \leq trainingsteps \leq 500 \end{array} \right\} \tag{6.1}$$

$$CPUpressure = \left\{ \begin{array}{ll} 25\%PM1, 50\%PM4, & \text{if } 0 \leq trainingsteps \leq 100 \\ 50\%PM1, 25\%PM4, & \text{if } 100 \leq trainingsteps \leq 200 \\ 50\%PM2, 25\%PM3, & \text{if } 200 \leq trainingsteps \leq 300 \\ 0\%, & \text{if } 300 \leq trainingsteps \leq 400 \\ 50\%PM1, 25\%PM2, 75\%PM3 & \text{if } 400 \leq trainingsteps \leq 500 \end{array} \right\} \tag{6.2}$$

6.1.2 Inference Events

With a more granular view on the conducted experiments, the scenario we simulated during infrence is a minature of the training one and is as following:

$$QoS = \left\{ \begin{array}{ll} 35seconds, & \text{if } 0 \leq trainingsteps \leq 150 \\ 26seconds, & \text{if } 150 \leq trainingsteps \leq 250 \end{array} \right\} \tag{6.3}$$

$$CPUpressure = \left\{ \begin{array}{ll} 25\%PM1, 50\%PM4, & \text{if } 0 \leq trainingsteps \leq 50 \\ 50\%PM1, 25\%PM4, & \text{if } 50 \leq trainingsteps \leq 100 \\ 50\%PM2, 25\%PM3, & \text{if } 100 \leq trainingsteps \leq 150 \\ 0\%, & \text{if } 150 \leq trainingsteps \leq 200 \\ 50\%PM1, 25\%PM2, 75\%PM3 & \text{if } 200 \leq trainingsteps \leq 250 \end{array} \right\} \tag{6.4}$$

6.2 Examined Schedulers

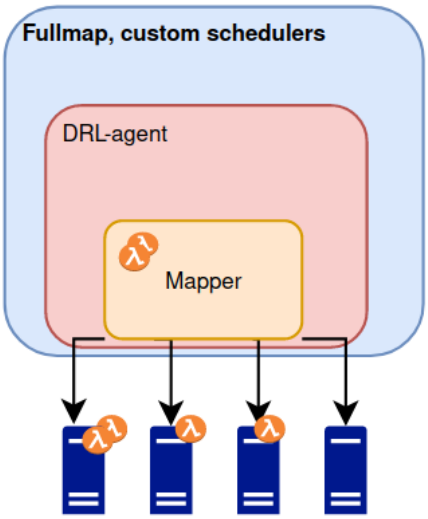
We examine four different schedulers as part of the Mapper component (Mapper, so as to determine the inter-relationship between the DRL-agent's proposed actions and the employed scheduling mechanism. With this approach, we aim to quantify the impact of i) the scheduling granularity when migrating functions and ii) heterogeneity- and interference-awareness with our proposed framework. Specifically, we developed four distinct schedulers, all differing in their actionspace, i.e., Fullmap-based, Custom- based, Kubernetes-based, Oracle-based. The former two decide both the migration and the destination of a function, while the latter two decide just whether a function should be migrated or not and a third-party scheduler, Kubernetes and Oracle respectively, locates the migrating function to a node, with its own policy. The Oracle-based scheduler leverages offline profiling information of the performance of deployed functions and decides the optimal scheduling policy accordingly.

6.2.1 Fullmap-guided DRL-based Scheduler

The fullmap-guided DRL-based scheduler is a specific solution regarding the underlying infrastructure. Specifically, the agent has an actionspace of size 15 which includes twelve actions that place each available function to the all available nodes, two actions that scale up and down the *faceanalyzer* and *mobilenet* functions and a action for remaining in the same state.

6.2.2 Custom-guided DRL-based Scheduler

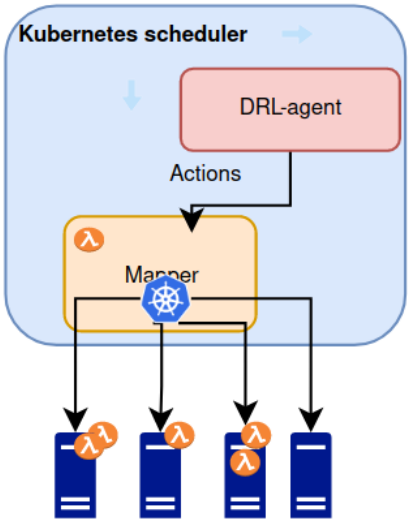
The custom-guided DRL-based scheduler is a tailor-made solution based on evidence of repeative training of various agents. Specifically, the agent has an actionspace of size 12 which includes eight actions that place the *framer* and *facedetector* functions to all nodes available, one action that moves the *faceanalyzer* and the *mobilenet* functions to the node with the minimum C0 score, two actions that scale up and down the *faceanalyzer* and *mobilenet* functions and one more function for preserving the current state. The C0 score, essentially, is obtained from the PCM metrics and is metric that tells how active the cores of this particular machine are at the specific moment. More details are offered here.



Σχήμα 6.1: *Fullmap-, Custom-based schedulers overview*

6.2.3 Kubernetes-guided DRL-based Scheduler

The kubernetes-guided DRL-based scheduler, as the name states, is an approach where functions' migration is also depending to the k8s scheduler. More precisely, this scheduler has an actionspace of size 4, which includes a function for moving the *framer* function, one for moving the *facedetector*, one for *faceanalyzer* and *mobilenet* functions and lastly one action for preserving the current state. The new location of the moved function is determined from the k8s scheduler and not by the agent. For example, if the agent's action is "Move facedetector", then *facedetector* would be moved and placed to some node that k8s scheduler has decided upon its own logic.



Σχήμα 6.2: *Kubernetes scheduler overview*

6.2.4 Oracle-guided DRL-based Scheduler

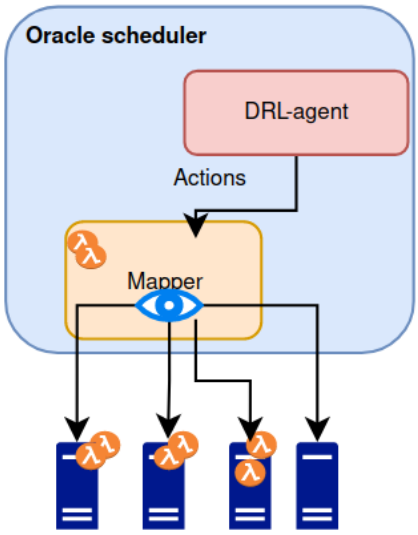
Last, but not least, we developed an oracle-guided DRL-based scheduler. An oracle observation (the information that is invisible during online decision making, but is available during offline training) is a way to facilitate learning and this is what we leveraged in this final scheduler. Offline, in the early stages of this thesis, we performed a multivariational profiling on our workflow. With the oracle-guided scheduler, we utilized the offline knowledge of execution profiles under mixed interference conditions in order to place the candidate function to the best node possible depending on oracle observations we have got in our property. Thus, the agent would be responsible of indicating the function that has to be migrated to another node, and if so, oracle is telling the destination node of this migration. This scheduler has an actionspace of size 4, which includes a function for moving the *framer* function, one for moving the *facedetector*, one for the *faceanalyzer* and *mobilenet* and lastly one for preserving the current state and therefore make no migrations.

6.3 Performance Evaluation of DRL-based Schedulers

In this section, a performance evaluation under training for each DRL-based scheduler is offered as we get to view how much close the models succeed to be regarding the goals we have set before. All schedulers have the same DQN architecture described in Section 5.2 but differ on the agents' actionspace and the ways those actions are actually performed. These deltas are to be examined in the following subsections.

Training of Fullmap-based

We trained the fullmap scheduler for 500 steps and the results are depicted in figure 6.4. As we can see in the first plot, the average reward the agent gets is not that stable and thus stability concerns are raised for this component. In the second plot, whereas



Σχήμα 6.3: *Oracle scheduler overview*

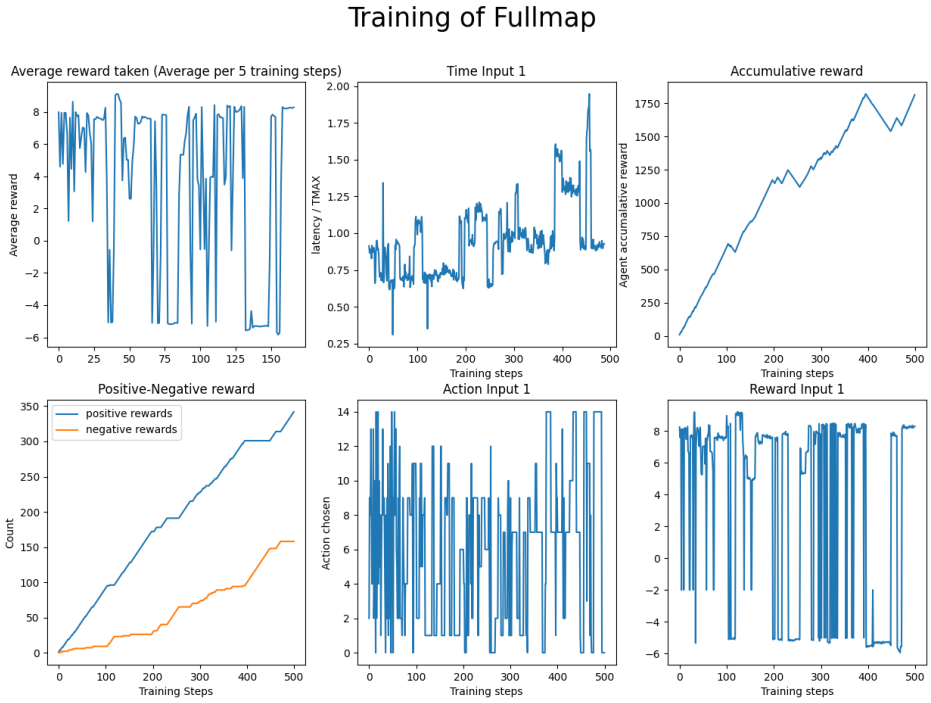
the QoS quotient is plotted, we can see that the model occasionally exceeds the value of 1, meaning that it violates the desired QoS. But generally, a mild convergence is obtained on most of the time. Regarding the reward plots, it is evident that the scheduler manages a lot of times to achieve its goal and thus the positive rewards outbalance the negative ones, but not by far. At the time of the 400th training step, in particular, it seems that the agent found it really difficult to find a solution to the interference applied at that stage while the desired QoS was at its "stricter" form at the value of 26.

Training of Custom-based

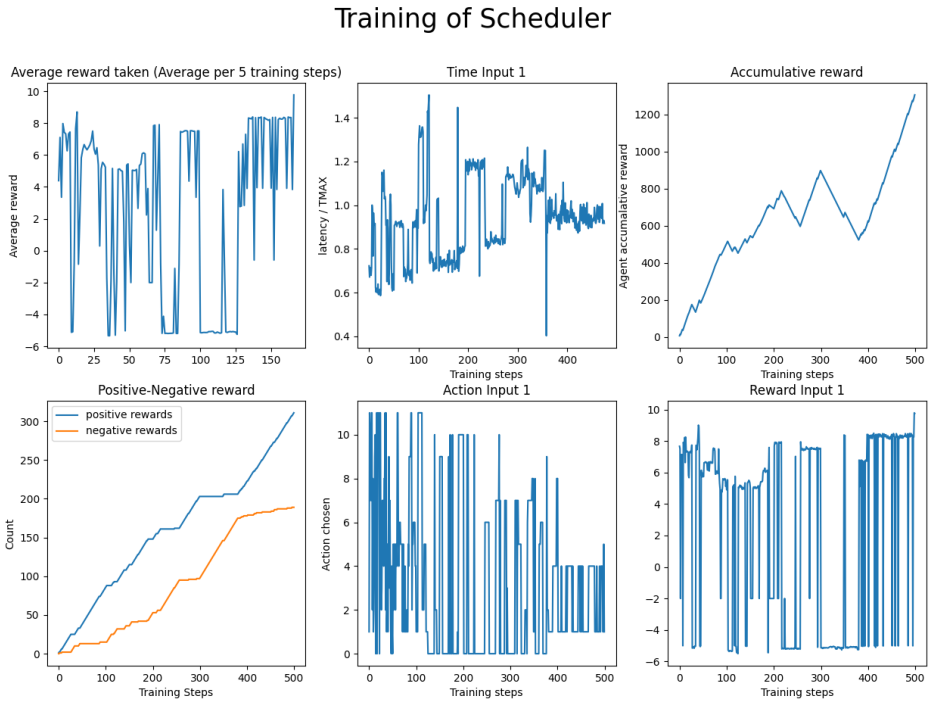
We trained the custom scheduler for 500 steps and the results are deplicted in figure 6.5. In the first plot we can see a nearly balanced earning of average reward (average on 5 steps), even though at times it seems unstable and prone to deep and high peaks. In the second plot, we are having a converged plot of the QoS quotient that indicates the agent's ability to achieve the desired QoS without overperforming and overutilizing the underlying resources. In the following plots, the accumulative reward and the number of successful servings and violations are provided. Especially in the 300th step, the agent finds it difficult to adjust to a stricter QoS but eventually it manages to serve it with great success.

Training of Kubernetes-based

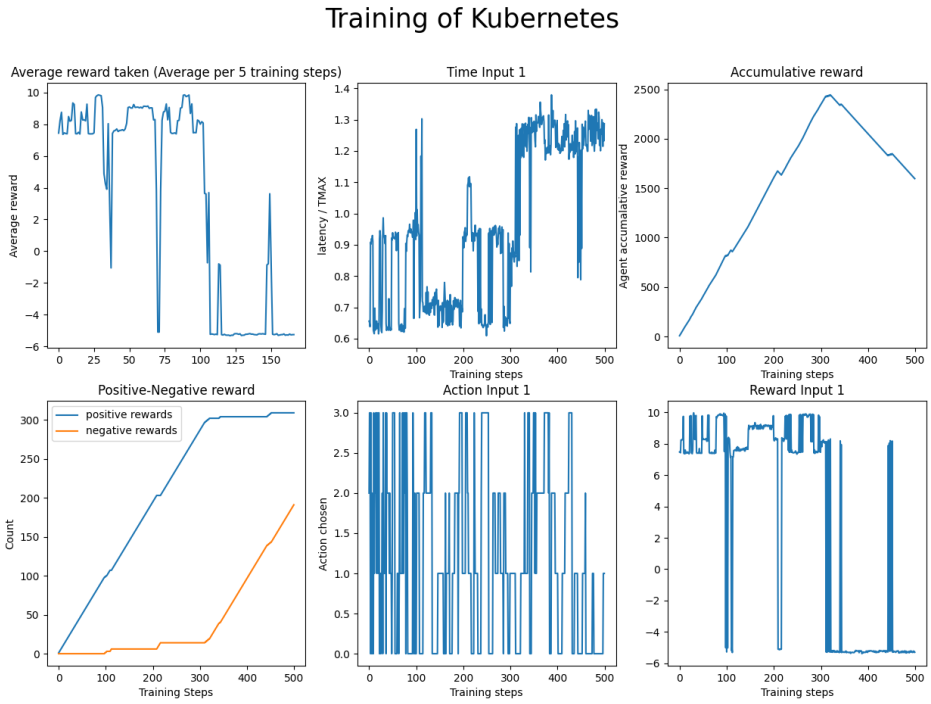
The kubernetes-guided scheduler was also trained for 500 steps and the results of its training are shown in figure 6.6. In the first plot, we get to view a good start but in the future it fails to get positive average rewards. The same thing is also present in the second plot, where stability is not found enough. From the rest plots, it is evident that this scheduler was quite sensitive and adaptable in the first half of training but failed drastically to adjust when a different QoS was requested in the second half and especially after the 300th training step.



Σχήμα 6.4: *Training plots of fullmap-guided DRL-based Scheduler*



Σχήμα 6.5: *Training plots of custom-guided DRL-based Scheduler*



Σχήμα 6.6: *Training plots of kubernetes-guided DRL-based Scheduler*

Training of Oracle-based

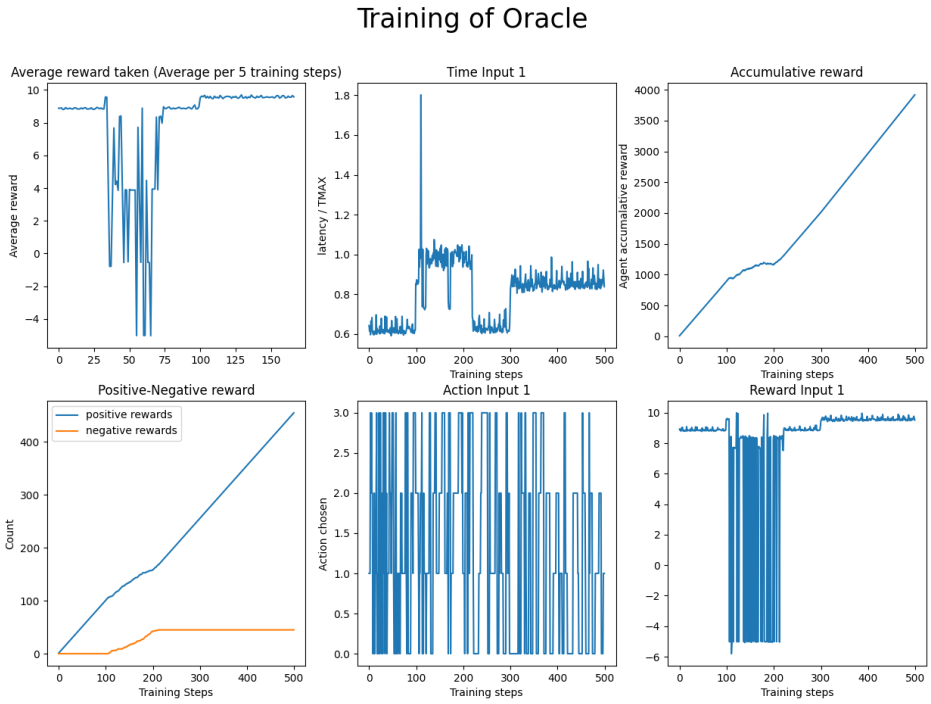
The oracle-guided scheduler was trained for 500 steps and the results are shown in figure 6.7. By keeping the same simulation scenarios with the above mentioned schedulers, the scheduler achieved relatively fast a high quality adaptability. Its behaviour is quite converging as we can see in the second plot, where at all times the scheduler has made correct decisions that often nearly touch the best possible value of the QoS quotient which is one. One is an upper limit for serving the user-defined QoS, a value greater than one results in a violation.

6.4 Comparative Evaluation of Schedulers during Training

In this section, we are presenting a comparative evaluation of the four DRL-based schedulers we developed. The comparison will be made in terms of QoS quotient, Cumulative Reward, QoS violation ratio, required time for convergence and scalability. First, we examine the ability of the DRL-agent to learn the appropriate actions to effectively adapt to dynamic interference conditions and QoS requirements during training. For the first 300 training steps, as mentioned above we set a loose QoS value of 35 seconds while and for the rest 200, a stricter QoS of 26 seconds.

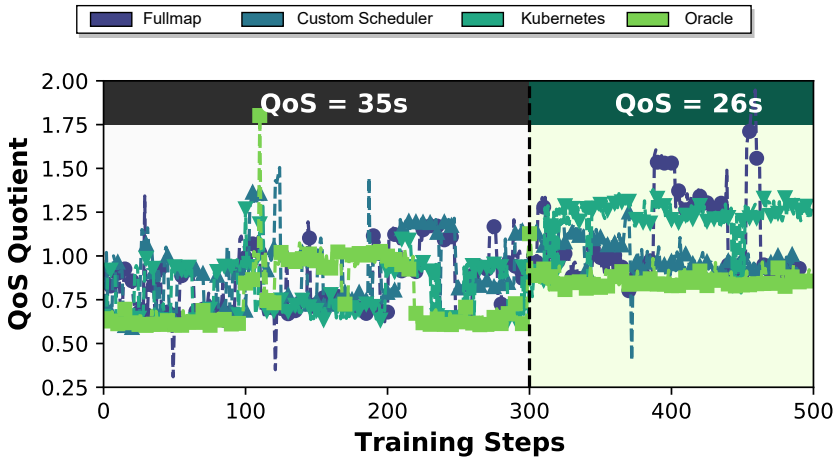
6.4.1 QoS Quotient

The QoS *quotient*, which is the quotient of the execution latency achieved divided by the target, user-defined latency requirement. A quotient less than or equal to one, implies



Σχήμα 6.7: *Training plots of oracle-guided DRL-based Scheduler*

a successful QoS serving while a quotient greater that one suggests a QoS violation. Furthermore, the closer to one a successful QoS serving is, the more regulated the end-to-end execution latency is considered, i.e., the QoS_{target} is achieved without over-allocating resources and thus utilization optimization is fulfilled. As depicted in figure 6.8, despite the *Oracle-based*, the *Custom-based* approach also presents valuable stability regarding the changes in resource stress levels and the change to a stricter QoS. It manages to adapt relatively fast while remaining close to the value of 1. A similar but less stable and less closer-to-1 behaviour is introduced by the *Fullmap-based* scheduler which might was affected by a larger actionspace comparatively to the *Custom-based*. Last, the *Kubernetes-based* approach even though at first seems to be expectantly adaptive, during the strict QoS period due to its heterogeneity-, interference-unawareness fails to adjust its decisions to the occurring conditions.

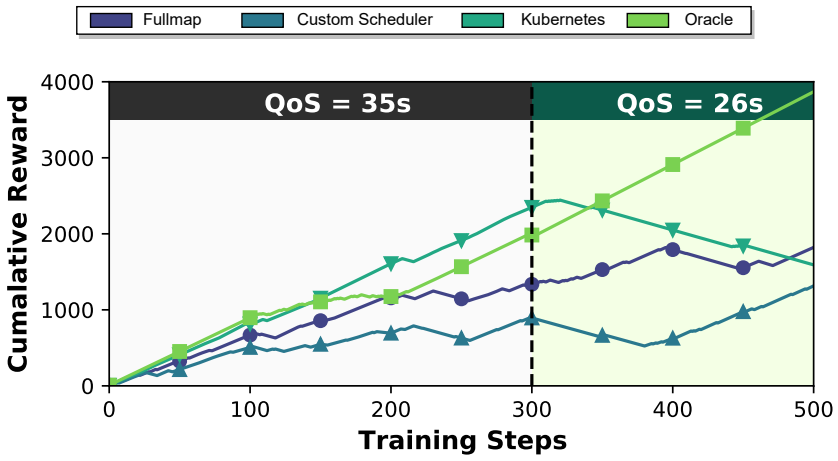


(α')

Σχήμα 6.8: *QoS quotient over training*

6.4.2 Cumulative Reward

The reward achieved per scheduler reveals its effectiveness identify a more efficient function topology in terms of resource utilization, since our proposed reward function is designed to maximize the reward w.r.t. resource efficiency. Figure 6.9 shows the received rewards per agent over time. Again, as expected, the *Oracle-based* approach is the most dominant. The *Fullmap-based* scheduler seems to secure a greater amount of reward than the *Custom-based* but the *Custom-based* approach presents a steeper slope at the latest stages of training which hints a better knowledge of the given task that leads to better results in the future. Finally, the *Kubernetes-based* scheduler as highlighted above because of its inability to adjust to the QoS change, it failed to receive positive rewards in the second half of the training period.

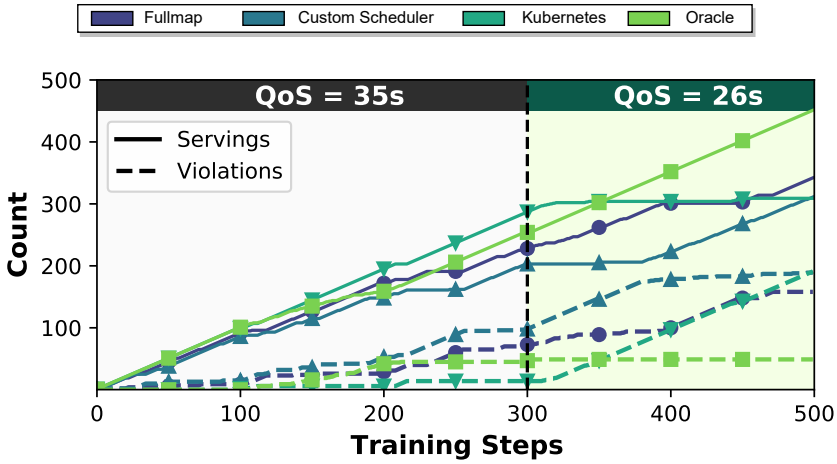


(α')

Σχήμα 6.9: *Cumulative reward over training*

6.4.3 QoS Violation Ratio

Figure 6.10 shows the successful QoS servings and QoS violations per scheduler. *Fullmap-based* and *Custom-based* prove the ability of DRL to tackle the dynamic scheduling problem, since QoS successful servings outnumber the QoS violations. The *Oracle-based* agent with great ease serves most of the requests, sealing the ability of DRL to collaborate with oracle observations, showing the criticality of a highly-efficient scheduler. Last, but not least, the *Kubernetes-based* scheduler, as mentioned above, seems to be the last performer among all approaches.



(α')

Σχήμα 6.10: *QoS violations over training*

6.4.4 Time Required for Convergence

In terms of training cycles needed for convergence, the oracle-guided scheduler is again the winner for the same reason as hinted above. Despite that, the custom-guided scheduler manages to learn faster the application's profile and migrate the key functions on the right nodes with higher stability. The fullmap-guided scheduler's training is closely related to the custom-guided as we could easily detect by observing figures 6.4 and 6.5. For the kubernetes-guided approach, convergence is not achieved enough as we can see in the second plot of figure 6.6 a lot of spikes, which are clear signs of instability.

6.4.5 Scalability

The scalability issue is of great importance and should always be adressed in order to be aware of the migration overhead needed before making the scheduler available to a different infrastructure or a different application. The fullmap-guided and the custom-guided schedulers have an action space that is specific to the deployed application and thus are not plug and play solutions to a workflow-agnostic environment. Moreover, their actions are dependent to the underlying infrastructure as well, making it a little bit laborious when redeploying them to a different topology. As a result, if one wants to serve a different workflow on a different topology, an offline migration of the actionspace is always needed. The kubernetes-guided scheduler is infrastructure-agnostic and could

be plugged to any cluster that is managed by kubernetes. The "kubernetes DNA" of this scheduler makes it the most scalable of all. Furthermore, its action space is less related to the deployed application than the first two schedulers, so less migration is needed for a different application. Lastly, the oracle-guided one is not fully bound to the underlying infrastructure and no needs offline migration if deployed to a different cluster. Although, this solution demands offline application profiling before any training would be performed and consequently more latency is added to the overall migration.

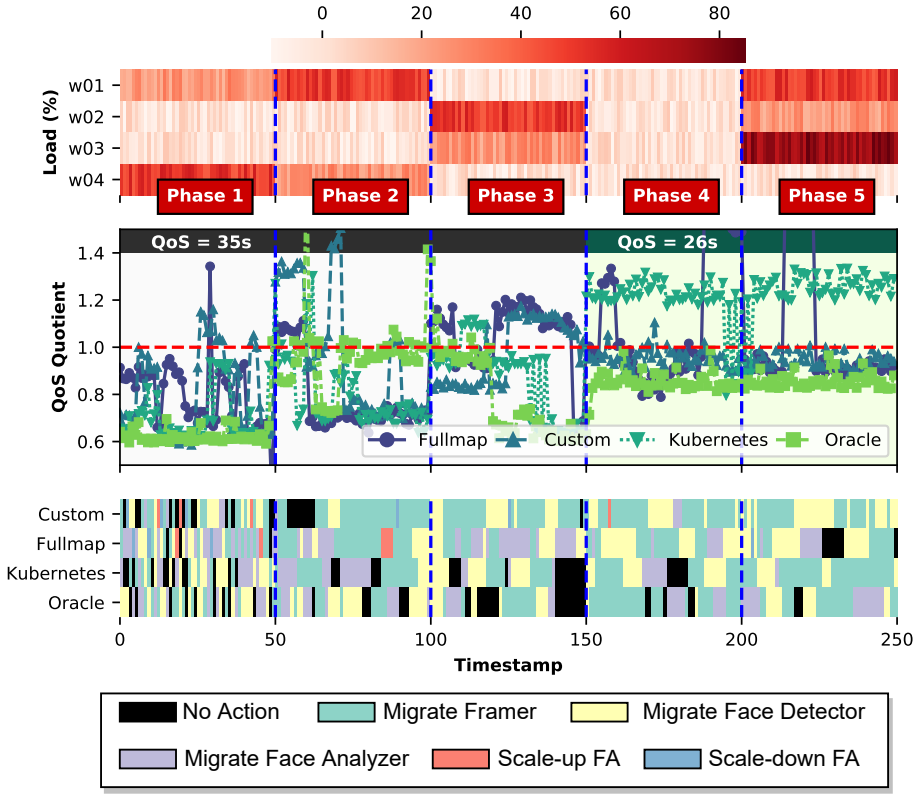
6.4.6 DRL-based vs native Kubernetes Scheduling

Last we compare our DRL-agent with a naive orchestration approach, where containers are orchestrated solely by Kubernetes, without any interaction with the DRL component. We deploy the video pipeline as separate containers sequentially, with one replica per container and we run 500 iterations of the workflow with different QoS constraints and resource interference. For the loose QoS constraint, Kubernetes manages to satisfy the target QoS only 34% of the times, whereas for strict QoS, it fails to satisfy the constraint 100% of the times. In contrast, the DRL achieves the target almost 95% of the times for loose and 75% for strict QoS on average respectively.

6.5 Evaluation Summary

As shown previously, the final reward calimed by the DRL-agent highly depends on the integrated scheduling logic. So a question that arises is: *"Does the DRL-agent alter its decision based on the integrated scheduler?"* To evaluate the "intelligence" of the DRL for different schedulers we freeze the parameters of the DQN, and examine the action space per scheduler during deployment. We explore different phases, where each one is characterized by different interference and QoS levels. Figure 6.11 shows the respective results, where the top figure reveals the interference pressure per VM and the middle and bottom ones the QoS Quotient and the actions made per scheduler respectively. This figure reveals three major insights: *i)* The first phase is characterized by high diversity in the action space, since none of the schedulers is able to satisfy the target QoS. *ii)* In the second phase, the Oracle-based agent is the only capable of meeting the target QoS. Compared to the rest of the schedulers that mostly migrate the Framer function, the Oracle-agent satisfies the QoS by migrating the Face-Detector, even though the Framer accounts for the greater part of the workflow's latency (Sections 4.5, 4.6) *iii)* Last, even in cases with minimal interference (Phase 4) or with similar decision patterns with the Oracle (Phase 5), the Kubernetes-based agent is unable to satisfy the target QoS, due to its unawareness both regarding the interference of the underlying infrastructure and the functions' performance variability due to heterogeneity.

We propose a DRL-based scheduler for dynamic function scaling and scheduling of serverless video analytics. Our solution exploits low-level system monitoring as part of the RL state representation and can be implemented using a variety of schedulers. It adapts to resource pressure fluctuations and achieves up to 91.6% compliance to changing QoS



Σχήμα 6.11: *Interference level, QoS Quotient and decision making of the DRL-agent under different scheduling policies*

targets.

Κεφάλαιο 7

Conslusion and Future Work

7.1 Summary

In this thesis, we discussed the needs that led to the full leverage of cloud computing and the ways serverless furthermore contributes to a wide spectrum of domains. Moreover, we presented some obstacles to be overcome in order to make the most of this promising cloud computing paradigm, since there exist a lot of tradeoff surfaces that should be taken into consideration before utilizing it.

Additonally, we analyzed and presented a set of factors that are able to undermine the performance of a serverless video analytics application and create an important performance variance when deployed on a cloud environment. More specifically, application's granularity, resource level interference from third-party workloads hosted on the same premises and worker nodes' heterogeneity could have a large impact on the important metrics such as end-to-end latency, billing and resource utilization.

Lastly, we design and propose a dynamic scheduling framework based on Deep Reinforcement Learning that aims to orchestrate the migration and scaling of functions regarding the dynamic state of the system and varying user requests. We proved that a DRL-based agent is able to learn, adapt and schedule serverless functions effectively while meeting the end-user time contraints 91.6% of requests on average during resource pressure fluctuations.

7.2 Future Work

Investigations, analysis and approaches described in this thesis are a first step towards identifying how Deep Reinforcement Learning can tackle the serverless function scheduling problem when granularity, heterogeneity and interference matters kick in a cloud environment. In the following subsections, we provide two suggestions for future work.

7.2.1 Identification of Doable User Requests

Our current approach tries to serve a user request regarding a user-defined upper time limit. It is often the case though, that an upper limit could not be served sufficiently

regardless the decisions made by a scheduler. In this scenario, we propose the development of a tool that classifies a user-defined upper time limit either as doable or not. So, unsavory attempts to serve an un-doable request can be skipped, something that may lead to faster training and convergence for the DRL agent.

7.2.2 Framework Expansion towards Application Agnosticism

Our proposed DRL-based framework is designed to solve the scheduling problem for the video-analytics pipeline described in the relative sections. A quite impactful approach would be to generalize the solution for serving all kinds of similar applications. For example, a possible solution towards that direction would be to design a tool that parses the desired application into serverless functions and then apply the current's framework capabilities to the newly configured functions

Βιβλιογραφία

[1] *IBM*. https://www.ibm.com/gr-en.

[2] *types of Hypervisors*. https://www.vmware.com/topics/glossary/content/bare-metal-hypervisor. Accessed data: 13-12-2021.

[3] *Kubernetes*. https://kubernetes.io/. Accessed date: 18-03-2022.

[4] *Apache OpenWhisk*. https://openwhisk.apache.org/.

[5] *OpenFaas*. https://www.openfaas.com/.

[6] *faas-flow*. https://github.com/s8sg/faas-flow. Accessed date: 26-6-2022.

[7] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy και Arkaprava Basu. *Faastlane: Accelerating Function-as-a-Service Workflows*. *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, σελίδες 805–820. USENIX Association, 2021.

[8] Vikram Sreekanti και others. *Cloudburst*. *Proceedings of the VLDB Endowment*, 13(12):2438–2452, 2020.

[9] Joao Carreira και others. *Cirrus: A Serverless Framework for End-to-End ML Workflows*. *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, σελίδα 13–24, New York, NY, USA, 2019. Association for Computing Machinery.

[10] Simon Shillaker και Peter Pietzuch. *Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing*. *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, σελίδες 419–433. USENIX Association, 2020.

[11] Ana Klimovicet al. *Pocket: Elastic Ephemeral Storage for Serverless Analytics*. *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, σελίδες 427–444, Carlsbad, CA, 2018. USENIX Association.

[12] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji και Saurabh Bagchi. *SONIC: Application-aware Data Passing for Chained Serverless Applications*. *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, σελίδες 285–301. USENIX Association, 2021.

[13] Christina Delimitrou και Christos Kozyrakis. *Paragon: QoS-aware scheduling for heterogeneous datacenters*. *ACM SIGPLAN Notices*, 48(4):77–88, 2013.

[14] Achilleas Tzenetopoulos και others. *Interference-Aware Orchestration in Kubernetes*. Berlin, Heidelberg, 2020. Springer-Verlag.

[15] Lucia Schuler, Somaya Jamil και Niklas Kühl. *AI-based Resource Allocation: Reinforcement Learning for Adaptive Auto-scaling in Serverless Environments*. *CoRR*, αβ-σ/2005.14410, 2020.

[16] Dimosthenis Masouros και others. *Rusty: Runtime Interference-Aware Predictive Monitoring for Modern Multi-Tenant Systems*. *IEEE Transactions on Parallel and Distributed Systems*, ΠΠ:1–1, 2020.

[17] Ioannis Fakinos και others. *Sequence Clock: A Dynamic Resource Orchestrator for Serverless Architectures*. *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, σελίδες 81–90. IEEE, 2022.

[18] Zheng Li, Maria Kihl, Qinghua Lu και Jens A. Andersson. *Performance Overhead Comparison between Hypervisor and Container Based Virtualization*. *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*. IEEE, 2017.

[19] *Linux Containers*. https://en.wikipedia.org/wiki/LXC. Accessed date: 10-12-2021.

[20] Sari Sultan, Imtiaz Ahmad και Tassos Dimitriou. *Container Security: Issues, Challenges, and the Road Ahead*. *IEEE Access*, 7:52976–52996, 2019.

[21] *Master/slave techonology*. https://en.wikipedia.org/wiki/Master/slave_(technology). Accessed date: 13-01-2022.

[22] *Definition of Cloud computing*. https://www.govinfo.gov/app/details/GOVPUB-C13-74cdc274b1109a7e1ead7185dfec2ada. Accessed data: 13-12-2021.

[23] *IaaS definition*. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf. Accessed data: 13-12-2021.

[24] *CaaS definition*. https://www.atlassian.com/continuous-delivery/microservices/containers-as-a-service. Accessed data: 13-12-2021.

[25] *PaaS definition*. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf. Accessed data: 15-12-2021.

[26] *Gartner study*. https://www.forbes.com/sites/louiscolumbus/2019/04/07/public-cloud-soaring-to-331b-by-2022-according-to-gartner/?sh=61d762105739. Accessed date: 15-12-2021.

[27] *SaaS definition*. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf. Accessed date: 15-12-2021.

[28] *AWS Lamda*. https://aws.amazon.com/lambda/. Accessed date: 05-01-2022.

[29] *Fission*. https://fission.io/.

[30] *Knative*. https://knative.dev/docs/.

[31] *Kubeless*. https://kubeless.io/.

[32] *Serverless Definition*. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.pdf. Accessed data: 05-01-2022.

[33] *Why use serverless*. https://www.cloudflare.com/learning/serverless/why-use-serverless/. Accessed date: 22-10-2022.

[34] Neil C Thompson, Kristjan H. Greenewald, Keeheon Lee και Gabriel F. Manso. *The Computational Limits of Deep Learning*. *ArXiv*, αβσ/2007.05558, 2020.

[35] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage και Anil Anthony Bharath. *A Brief Survey of Deep Reinforcement Learning*. *arXiv e-prints*, σελίδα αρΞι:1708.05866, 2017.

[36] *Occupy the Cloud*. https://dl.acm.org/doi/pdf/10.1145/3127479.3128601. Accessed date: 01-02-2022.

[37] *An introduction to Docker and Analysis of its Performance*. http://paper.ijcsns.org/07_book/201703/20170327.pdf. Accessed date: 30-7-2022.

[38] *Intel Performance Counter Monitor*. https://github.com/opcm/pcm. Accessed date: 31-07-2022.

[39] *Intel Performance Counter Monitor content*. https://www.intel.com/content/www/us/en/developer/articles/technical/performance-counter-monitor.html. Accessed date: 31-07-2022.

[40] *Serverless Workflow Specification*. https://github.com/serverlessworkflow/specification. Accessed data: 19-02-2022.

[41] *MinIO Object storage*. https://en.wikipedia.org/wiki/MinIO, https://min.io/. Accessed date: 18-01-2022.

[42] Kaiming He, Xiangyu Zhang, Shaoqing Ren και Jian Sun. *Deep Residual Learning for Image Recognition*, 2015.

[43] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter και Keith Winstein. *Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads*. *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, σελίδες 363–376, Boston, MA, 2017. USENIX Association.

[44] Tao Zheng, Jian Wan, Jilin Zhang και Congfeng Jiang. *Deep Reinforcement Learning-Based Workload Scheduling for Edge Computing*. *J. Cloud Comput.*, 11(1), 2022.

[45] *Influx Database*. https://www.influxdata.com/. Accessed date: 23-10-2022.

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

[47] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang και Wojciech Zaremba. *OpenAI Gym*, 2016.