ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ, ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Πειραματική Αξιολόγηση Εφαρμογών σε Kata Runtimes υπό Συνθήκες Παρεμβολών

ΣΤΑΥΡΟΥΛΑ ΑΛΙΚΑΚΟΥ

Επιβλέπων : Συμεών Παπαβασιλείου
Καθηγητής ΕΜΠ

Αθήνα, Οκτώβρης 2025

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ, ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Πειραματική Αξιολόγηση Εφαρμογών σε Kata Runtimes υπό Συνθήκες Παρεμβολών

ΣΤΑΥΡΟΥΛΑ ΑΛΙΚΑΚΟΥ

Επιβλέπων : Συμεών Παπαβασιλείου
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 17η Οκτωβρίου 2025.

(Υπογραφή)	(Υπογραφή)	(Υπογραφή)
.....	.....	.....
Συμεών Παπαβασιλείου Καθηγητής ΕΜΠ	Ιωάννα Ρουσσάκη Αναπληρώτρια Καθηγήτρια ΕΜΠ	Ελένη Στάη Επίκουρη Καθηγήτρια ΕΜΠ

Αθήνα, Οκτώβρης 2025

(Υπογραφή)

.....

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

Περίληψη

Η παρούσα διπλωματική εργασία διερευνά τη συγκριτική απόδοση διαφορετικών εκτελεστικών περιβαλλόντων containers στο πλαίσιο της υπολογιστικής των παρυφών (edge computing), εστιάζοντας στην αξιολόγηση των trade-offs μεταξύ ασφάλειας και απόδοσης. Συγκεκριμένα, εξετάζεται η επίδραση του overhead εικονικοποίησης στην απόδοση των Kata containers, τα οποία ενσωματώνουν την εικονικοποίηση υλικού μέσω Firecracker, σε σύγκριση με τα παραδοσιακά regular containers. Η μεθοδολογία περιλαμβάνει εκτεταμένη πειραματική αξιολόγηση σε περιβάλλον Kubernetes υπό διαφορετικά σενάρια παρεμβολής πόρων και δέσμευσης CPU. Μετρώνται κρίσιμες μετρικές απόδοσης σε πολλαπλά επίπεδα: επίπεδο εφαρμογής (latency, throughput), επίπεδο συστήματος (CPU utilization, IPC, context switches, cache miss rate), και επίπεδο δικτύου (overhead). Η ανάλυση εστιάζει στον εντοπισμό κρίσιμων σημείων απόδοσης και στην κατανόηση των αρχιτεκτονικών παραγόντων που επηρεάζουν την κλιμακωσιμότητα των δύο προσεγγίσεων. Η εργασία συμβάλλει στην κατανόηση του πώς η εικονικοποίηση επηρεάζει την απόδοση σε περιβάλλοντα περιορισμένων πόρων που συναντώνται στο edge computing. Αναλύονται οι μηχανισμοί I/O (virtio-block vs overlay filesystem), τα mode transitions, και η διαχείριση μνήμης ως πηγές overhead. Τα αποτελέσματα παρέχουν πρακτική καθοδήγηση για την επιλογή κατάλληλου container runtime ανάλογα με τις απαιτήσεις απόδοσης, ασφάλειας, και διαθέσιμων πόρων, καθώς και για τη βελτιστοποίηση της αξιοποίησης πόρων.

Λέξεις Κλειδιά— Υπολογιστική των παρυφών, Εκτελεστικά Περιβάλλοντα Containers, Kata Containers, Firecracker, Αξιολόγηση Απόδοσης, Kubernetes, Παρεμβολή Πόρων, Δέσμευση CPU, Απόδοση I/O, Εικονικοποίηση

Abstract

This thesis investigates the comparative performance of different container execution environments in the context of edge computing, focusing on evaluating the trade-offs between security and performance. Specifically, it examines the impact of virtualization overhead on the performance of Kata containers, which incorporate hardware-assisted virtualization through Firecracker, compared to traditional regular containers. The methodology includes extensive experimental evaluation in a Kubernetes environment under different resource contention and CPU allocation scenarios. Critical performance metrics are measured at multiple levels: application-level (latency, throughput), system-level (CPU utilization, IPC, context switches, cache miss rate), and network-level (overhead). The analysis focuses on identifying critical performance points and understanding the architectural factors that affect the scalability of both approaches. This work contributes to understanding how virtualization affects performance in resource-constrained environments typical of edge computing. It analyzes I/O mechanisms (virtio-block vs overlay filesystem), mode transitions, and memory management as sources of overhead. The findings provide practical guidance for selecting appropriate container runtimes based on performance requirements, security needs, and available resources, as well as for optimizing resource utilization.

Keywords— Edge Computing, Container Runtimes, Kata Containers, Firecracker, Performance Evaluation, Kubernetes, Resource Interference, CPU Pinning, I/O Performance, Virtualization

Ευχαριστίες

Αρχικά, θα ήθελα να εκφράσω τις ευχαριστίες μου στον κύριο Συμεών Παπαβασιλείου για επίβλεψή του και την ευκαιρία που μου προσέφερε να ασχοληθώ με το συγκεκριμένο επιστημονικό πεδίο.

Ιδιαίτερες ευχαριστίες οφείλω και στο διδάκτορα Δημήτρη Σπαθαράκη και υποψήφιο διδάκτορα Νικόλαο Φιλίνη για την πολύτιμη βοήθεια και καθοδήγησή τους. Η τεχνική τους εμπειρία, οι εποικοδομητικές συζητήσεις, και η διαθεσιμότητά τους να απαντήσουν σε ερωτήματα και απορίες ήταν ανεκτίμητες για την πορεία της έρευνας και την επίλυση προκλήσεων που προέκυψαν κατά την υλοποίηση.

Τέλος, θα ήθελα να ευχαριστήσω θερμά την οικογένειά μου, τους φίλους και συμφοιτητές μου για την υποστήριξη και ενθάρρυνση που μου παρείχαν όλα αυτά τα χρόνια των σπουδών μου.

Περιεχόμενα

Κατάλογος Σχημάτων

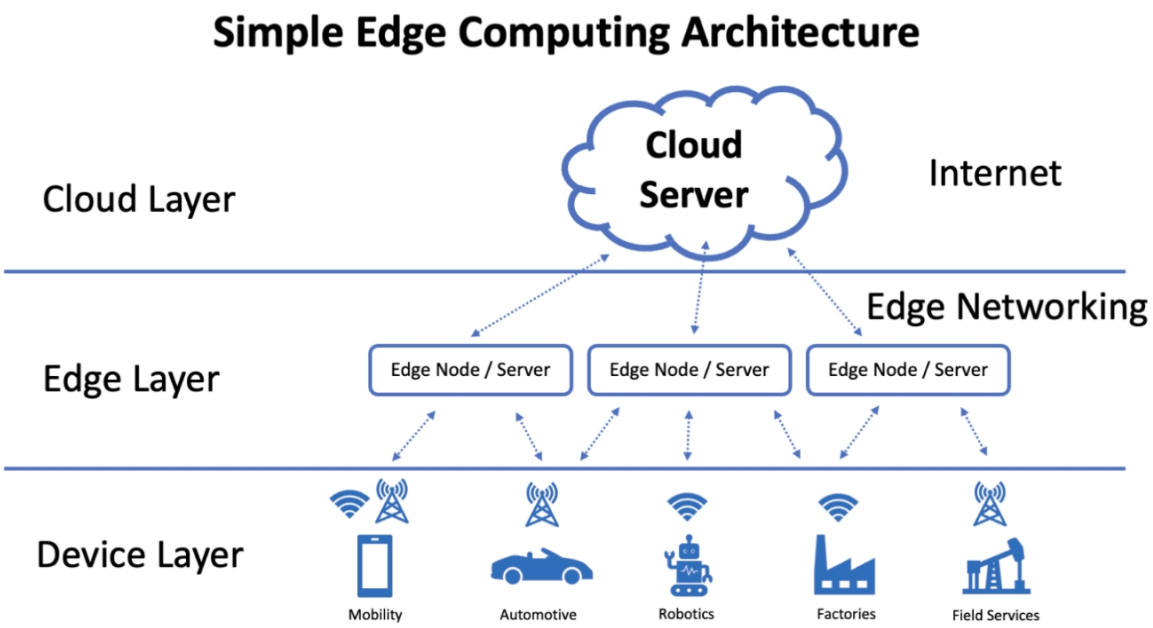
Κατάλογος Πινάκων

Εισαγωγή

Η διαχείριση και κατανομή πόρων στα περιβάλλοντα **edge computing** αποτελεί ένα κρίσιμο πεδίο έρευνας που είναι αποτέλεσμα της αυξανόμενης ζήτησης για υπολογιστικές υπηρεσίες χαμηλής καθυστέρησης και υψηλής απόδοσης. Σε αντίθεση με την παραδοσιακή cloud computing αρχιτεκτονική, όπου οι υπολογιστικοί πόροι συγκεντρώνονται σε κέντρα δεδομένων, το edge computing φέρνει την επεξεργασία δεδομένων πιο κοντά στους τελικούς χρήστες, χρησιμοποιώντας κατανεμημένους κόμβους. [1]

Η προσέγγιση αυτή επιτρέπει τη σημαντική μείωση της δικτυακής καθυστέρησης και βελτιώνει την ποιότητα υπηρεσίας (Quality of Service - QoS), καθιστώντας την ιδιαίτερα κατάλληλη για κρίσιμες εφαρμογές που απαιτούν επεξεργασία δεδομένων πραγματικού χρόνου. Ωστόσο, η κατανεμημένη φύση του edge computing εισάγει σημαντικές προκλήσεις στη διαχείριση των διαθέσιμων πόρων, καθώς οι edge κόμβοι συνήθως χαρακτηρίζονται από περιορισμένες υπολογιστικές δυνατότητες και ετερογένεια υλικού.

Η αποτελεσματική κατανομή πόρων στο edge computing προϋποθέτει τη βέλτιστη διαχείριση τόσο των υπολογιστικών πόρων, όπως επεξεργαστή (CPU), μνήμης, αποθηκευτικού χώρου, όσο και των δικτυακών πόρων, μεταξύ διαφορετικών εφαρμογών με ποικίλες απαιτήσεις απόδοσης. Η διαδικασία γίνεται ακόμη πιο σύνθετη λόγω της ανάγκης δυναμικής προσαρμογής σε μεταβαλλόμενα επίπεδα φόρτου και κινητικότητα των χρηστών. Αυτό δημιουργεί την απαίτηση για την εξισορρόπηση φορτίου, που αφορά στην ισομερή κατανομή του φόρτου εργασίας μεταξύ διαφορετικών κόμβων ώστε να αποφεύγεται η υπερφόρτωση συγκεκριμένων πόρων. Επιπλέον, απαιτείται προληπτική βελτιστοποίηση της απόδοσης που επιτρέπει την πρόβλεψη και αντιμετώπιση προβλημάτων απόδοσης πριν αυτά επηρεάσουν την ποιότητα των παρεχόμενων υπηρεσιών.



Σχήμα 1.1: Πολυεπίπεδη δομή edge computing

1.1 Προκλήσεις

Η πιο σημαντική πρόκληση στη διαχείριση πόρων edge computing αποτελεί η παρεμβολή απόδοσης (**performance interference**) μεταξύ συνεγκατεστημένων εφαρμογών. Ο ανταγωνισμός για την κοινή χρήση πόρων μεταξύ εικονικών μηχανών (VMs) αποτελεί την κύρια αιτία του performance interference. Όταν πολλά VMs ή containers πραγματοποιούν εντατική χρήση ενός συγκεκριμένου τύπου πόρου, όπως CPU, μνήμη και εύρος ζώνης δικτύου, προκύπτει ανταγωνισμός που οδηγεί σε σημαντική υποβάθμιση της απόδοσης.

Εκτός από τα προβλήματα απόδοσης, το edge computing αντιμετωπίζει σημαντικές προκλήσεις ασφάλειας που προκύπτουν από τη κατανεμημένη αρχιτεκτονική και την εγγύτητα στους τελικούς χρήστες. Οι edge κόμβοι, είναι πιο ευάλωτοι σε κινδύνους ασφαλείας συγκριτικά με τα κέντρα δεδομένων.

Ένα βασικό δίλημμα στο edge computing αφορά την επιλογή μεταξύ ασφάλειας και απόδοσης. Τα VMs προσφέρουν υψηλή ασφάλεια, καθώς κάθε εφαρμογή εκτελείται σε πλήρως απομονωμένο περιβάλλον, ωστόσο χαρακτηρίζονται από υψηλή κατανάλωση πόρων και αργό χρόνο εκκίνησης. Αντίθετα, τα containers είναι ελαφριά, ταχύτερα και απαιτούν περιορισμένους πόρους, παρουσιάζουν όμως χαμηλότερη ασφάλεια καθώς μοιράζονται τον ίδιο πυρήνα του λειτουργικού συστήματος. Αυτό συνεπάγεται περιορισμένη απομόνωση μεταξύ των εφαρμογών και δυνατότητα επίδρασης σε γειτονικές εφαρμογές ή το host σύστημα σε περίπτωση παραβίασης ενός container.

Τη λύση σε αυτό το δίλημμα αποτελούν τα Kata Containers με Firecracker ως hypervisor, τα οποία και εξετάζονται στην παρούσα διπλωματική εργασία ως ένα

υποσχόμενο μέσο για την αντιμετώπιση των προκλήσεων του edge computing.

1.2 Συνεισφορά

Η παρούσα εργασία διερευνά την απόδοση δύο διαφορετικών τεχνολογιών containers σε edge computing συστήματα με στόχο την αξιολόγηση των trade-offs μεταξύ ασφάλειας και επίδοσης. Συγκρίνουμε τα παραδοσιακά containers, που χρησιμοποιούν runc runtime, με τα Kata Containers, που χρησιμοποιούν kata-runtime με Firecracker ως hypervisor, για να κατανοήσουμε τις επιπτώσεις στην απόδοση όταν προστίθενται επιπλέον μηχανισμοί ασφάλειας. Μέσω της σύγκρισης αυτών των δύο τεχνολογιών, η εργασία στοχεύει να προσδιοριστούν οι διαφορές στην απόδοση, την κατανάλωση πόρων και τα χαρακτηριστικά overhead σε περιβάλλον Kubernetes. Τα Kata Containers με Firecracker ως λύση προσφέρουν ενισχυμένη ασφάλεια παρέχοντας απομόνωση σε επίπεδο υλικού μεταξύ εφαρμογών που εκτελούνται στον ίδιο κόμβο.

Η πειραματική αξιολόγηση πραγματοποιείται μέσω μιας εφαρμογής Flask που υλοποιεί πολλαπλές λειτουργίες επεξεργασίας εικόνας (αναστροφή, περιστροφή, φιλτράρισμα, μετατροπή σε grayscale, και αλλαγή μεγέθους). Η εφαρμογή αυτή χρησιμοποιείται ως τυποποιημένο φορτίο εργασίας (benchmark) για τη μέτρηση της απόδοσης υπό διαφορετικά επίπεδα φόρτου. Η μεθοδολογία περιλαμβάνει δοκιμές φόρτου με χρήση του εργαλείου Vegeta για τη δημιουργία κίνησης HTTP σε διαφορετικά επίπεδα requests per second , ενώ παράλληλα πραγματοποιείται monitoring σε επίπεδο συστήματος με το εργαλείο perf για τη συλλογή μετρητών απόδοσης υλικού. Για τη συγκέντρωση και αποθήκευση μετρικών χρησιμοποιείται το Prometheus, ενώ το Grafana παρέχει την οπτικοποίηση και ανάλυση των δεδομένων απόδοσης σε πραγματικό χρόνο. Η εφαρμογή μετρά εσωτερικά τους χρόνους φόρτωσης και επεξεργασίας εικόνας, επιτρέποντας την ακριβή ανάλυση της απόδοσης σε επίπεδο εφαρμογής σε σχέση με την καθυστέρηση σε επίπεδο δικτύου.

1.3 Δομή της εργασίας

Η εργασία οργανώνεται σε έξι κύρια μέρη, τα οποία καλύπτουν τη θεωρητική μελέτη, την τεχνολογική υλοποίηση και την πειραματική αξιολόγηση του θέματος. Αρχικά, στο Κεφάλαιο 2 (Βιβλιογραφία), παρουσιάζονται η έννοια της εικονικοποίησης και σχετικές μελέτες που εξετάζουν τα φαινόμενα παρεμβολής απόδοσης μεταξύ συνεγκατεστημένων εφαρμογών. Στη συνέχεια, στο Κεφάλαιο 3 (Kata Containers and Kubernetes), αναλύονται οι τεχνολογίες Kata Containers, το περιβάλλον Kubernetes και ο Firecracker hypervisor. Στο Κεφάλαιο 4 (Προτεινόμενη Μεθοδολογία) παρουσιάζεται η μεθοδολογία που ακολουθήθηκε για την

πειραματική αξιολόγηση, συμπεριλαμβανομένων των μεθόδων διαχείρισης πόρων, όπως οι τεχνικές pinning ,η ανάλυση disk I/O και network I/O και περιγράφεται η πειραματική διάταξη. Ακολουθεί το Κεφάλαιο 5 (Πειραματική Αξιολόγηση), όπου και παρουσιάζονται η μεθοδολογία μέτρησης και τα αποτελέσματα της συγκριτικής μελέτης των δύο τεχνολογιών containers υπό διαφορετικές συνθήκες φόρτου. Τέλος, στο Κεφάλαιο 6, η εργασία ολοκληρώνεται με τα συμπεράσματα και τις προτάσεις για μελλοντική έρευνα.

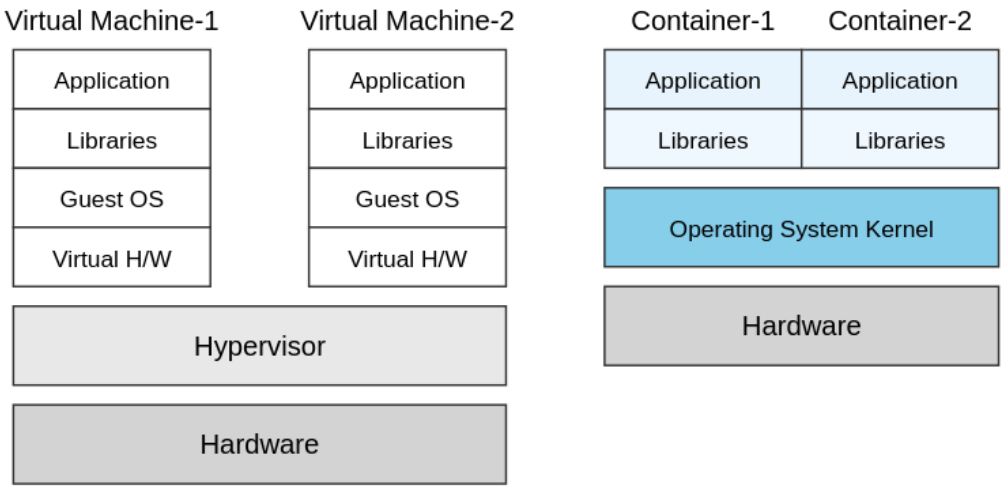
Βιβλιογραφία

2.1 Εικονικοποίηση

Η εικονικοποίηση (**virtualization**) σε edge computing περιβάλλοντα έχει γίνει αντικείμενο εκτενούς έρευνας λόγω των προκλήσεων που σχετίζονται με το performance interference και την απομόνωση εφαρμογών. Η παράλληλη εκτέλεση πολλαπλών φόρτων εργασίας στον ίδιο φυσικό κόμβο δημιουργεί σύνθετα προβλήματα που απαιτούν εξειδικευμένες λύσεις για τη διασφάλιση της απόδοσης. Η εικονικοποίηση επιτρέπει τη φιλοξενία πολλαπλών VMs στον ίδιο φυσικό εξυπηρετητή, παρέχοντας τη δυνατότητα βέλτιστης αξιοποίησης των διαθέσιμων πόρων, μείωσης του λειτουργικού κόστους και ευελιξίας στη διαχείριση εφαρμογών. Στο πλαίσιο αυτών των προκλήσεων απομόνωσης και διαχείρισης πόρων, ιδιαίτερη σημασία αποκτά η ανάλυση των δύο βασικών τεχνολογιών εικονικοποίησης: των VMs και των containers. Οι δύο τεχνολογίες εφαρμόζουν διαφορετικούς μηχανισμούς για την απομόνωση της απόδοσης.

Η εικονικοποίηση υλικού, που αξιοποιείται από τα VMs, βασίζεται σε έναν hypervisor, ο οποίος διαχωρίζει και εικονικοποιεί τους φυσικούς πόρους του διακομιστή σε πολλαπλά VMs. Κάθε VM διαθέτει μια πλήρη στοίβα: φυσικό υλικό, εικονικό υλικό, hypervisor, guest λειτουργικό σύστημα, βιβλιοθήκες και εφαρμογές. Με αυτόν τον τρόπο, κάθε VM διαθέτει το δικό του ανεξάρτητο λειτουργικό σύστημα και τις αντίστοιχες εφαρμογές, με πλήρη απομόνωση σε επίπεδο υλικού αλλά με μεγαλύτερο overhead.

Αντίθετα, η εικονικοποίηση λειτουργικού συστήματος, που αποτελεί τη βάση για τα containers, πραγματοποιείται σε επίπεδο πυρήνα και χαρακτηρίζεται από μια πιο απλή αρχιτεκτονική. Τα containers περιλαμβάνουν μόνο το φυσικό υλικό, τον πυρήνα του λειτουργικού συστήματος και τις εφαρμογές με τις βιβλιοθήκες τους. Επειδή όλα τα containers μοιράζονται τον ίδιο πυρήνα με το host, δεν απαιτείται guest λειτουργικό σύστημα. Έτσι, επιτυγχάνονται πολύ ταχύτεροι χρόνοι εκκίνησης και χαμηλότερη κατανάλωση πόρων, καθώς τα containers εκκινούν απλώς δημιουργώντας νέες διεργασίες χωρίς να χρειάζεται πλήρης εκκίνηση συστήματος. [2]



Σχήμα 2.1: Σύγκριση αρχιτεκτονικών μεταξύ VMs και Containers

Ένα επιπλέον πλεονέκτημα των containers είναι η δυνατότητα χρήσης soft resource limits, που τους επιτρέπει να αξιοποιούν ανεκμετάλλευτους πόρους άλλων containers. Σε περιβάλλοντα με περιορισμένους πόρους, αυτό μπορεί να προσφέρει έως και 40% βελτίωση απόδοσης σε σχέση με VMs με αυστηρά όρια πόρων. Παράλληλα, τα container images είναι έως και τρεις φορές μικρότερα και κατασκευάζονται ταχύτερα, αφού δεν περιλαμβάνουν πλήρες λειτουργικό σύστημα. [3]

Από πλευράς διαχείρισης, τα containers παρέχουν περισσότερες δυνατότητες ελέγχου μέσω των μηχανισμών του Linux (cgroups, namespaces), γεγονός που όμως αυξάνει την πολυπλοκότητα στη διαχείριση πόρων. Τα VMs, αντίθετα, διαθέτουν πιο αξιόπιστους μηχανισμούς μεταφοράς (migration), ενώ η μετακίνηση containers παραμένει περιορισμένη και λιγότερο σταθερή.

Παρά τα πλεονεκτήματά τους, τόσο τα VMs όσο και τα containers έχουν περιορισμούς στην απομόνωση απόδοσης (**performance isolation**). Η ανάγκη για performance isolation, δηλαδή η ικανότητα ενός συστήματος να διασφαλίζει ότι η απόδοση μιας εφαρμογής ή ενός VM δεν επηρεάζεται από άλλες εφαρμογές που τρέχουν στον ίδιο φυσικό εξυπηρετητή, έχει οδηγήσει στην ανάπτυξη διαφόρων προσεγγίσεων. Μία από αυτές είναι τα Kata Containers, τα οποία συνδυάζουν τα πλεονεκτήματα των ελαφρών containers (lightweight containers), με την ισχυρή απομόνωση που προσφέρουν τα VMs. Τα lightweight containers αποτελούν τεχνολογίες εικονικοποίησης που επιτρέπουν την απομονωμένη εκτέλεση εφαρμογών, αξιοποιώντας κοινό λειτουργικό σύστημα και καταναλώνοντας λιγότερους πόρους σε σύγκριση με τα παραδοσιακά VMs. Χάρη στα χαρακτηριστικά αυτά, προσφέρουν ταχύτερους χρόνους εκκίνησης, χαμηλότερη κατανάλωση μνήμης και τη δυνατότητα προσαρμογής σε αυξημένες ή μειωμένες απαιτήσεις φόρτου εργασίας (scalability). Ειδικά όταν ενσωματώνονται σε πλατφόρμες Kubernetes, τα Kata Containers στοχεύουν να πετύχουν μια ισορροπία μεταξύ ασφάλειας,

απόδοσης και σωστής διαχείρισης πόρων.

2.2 Παρεμβολές (Interference)

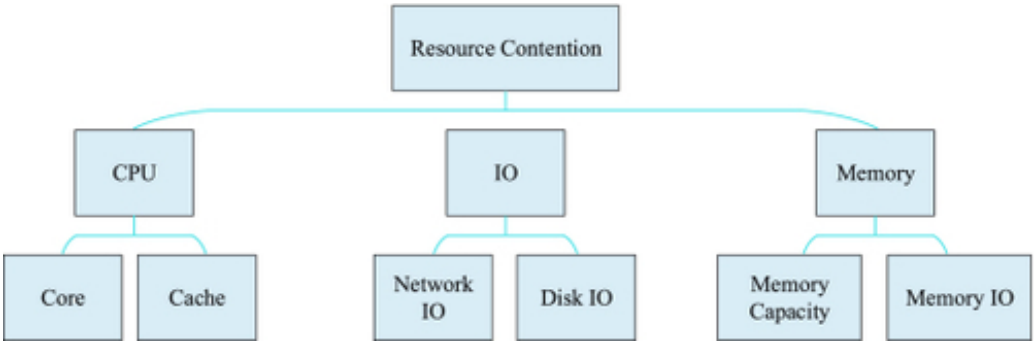
2.2.1 Μέθοδοι Μοντελοποίησης και Μέτρησης Παρεμβολών

Σύμφωνα με τους Lin κ.ά (2023) [4], η μέτρηση της απόδοσης εφαρμογών βασίζεται σε κρίσιμες παραμέτρους όπως η διάρκεια εκτέλεσης εργασιών και οι χρόνοι απόκρισης σε αιτήματα. Σε ιδανικές συνθήκες απομόνωσης, αυτοί οι δείκτες απόδοσης διατηρούν σταθερότητα, ωστόσο η παρουσία πολλών συνεντοπισμένων VMs εισάγει ανταγωνισμό για πόρους που περιορίζει την αναμενόμενη απόδοση, ειδικά όταν οι απαιτήσεις είναι παρόμοιες. Η ημιτελής απομόνωση πόρων αποτελεί τον κύριο παράγοντα πρόκλησης παρεμβολών. Αν και τεχνολογίες εικονικοποίησης, όπως οι hypervisors XenServer και vSphere, επιχειρούν να απομονώσουν CPU, μνήμη και δίσκο ανά VM, παραμένει δύσκολο να διαχωριστούν πόροι όπως η cache και το εύρος ζώνης δικτύου.

Όσον αφορά την CPU, η παράλληλη εκτέλεση πολλών εφαρμογών δημιουργεί ανταγωνισμό για τους πυρήνες της CPU και τη μνήμη cache. Κάθε VM, αν και διαθέτει δικό της εικονικό επεξεργαστή (vCPU), εξαρτάται από την πρόσβαση στην φυσική CPU για την πραγματική επεξεργασία. Το πρόβλημα επιδεινώνεται από την κοινή χρήση της Last-Level Cache (LLC), η οποία οδηγεί σε αυξημένα ποσοστά cache miss λόγω της απουσίας αποκλειστικού χώρου cache για κάθε VM.

Στους πόρους εισόδου-εξόδου (I/O), ο ανταγωνισμός εκδηλώνεται σε δύο επίπεδα: τοπικά, μέσω της πρόσβασης στον δίσκο για αποθήκευση και ανάκτηση δεδομένων, και απομακρυσμένα, μέσω των δικτυακών συνδέσεων για επικοινωνία με εξωτερικές συσκευές. Όταν πολλαπλές εφαρμογές εκτελούνται στον ίδιο server, αναπόφευκτα ανταγωνίζονται για το διαθέσιμο εύρος ζώνης τόσο του δίσκου όσο και του δικτύου.

Τέλος, για τη μνήμη, αν και συνήθως κάθε εφαρμογή διαθέτει τη δική της μνήμη, όταν δεν υπάρχει αρκετή διαθέσιμη μνήμη στο μηχάνημα, οι εφαρμογές θα αρχίσουν να ανταγωνίζονται και για αυτόν τον πόρο.



Σχήμα 2.2: Κατηγορίες ανταγωνισμού υπολογιστικών πόρων

Δεδομένων των παραπάνω αιτιών παρεμβολών, η ακριβής μέτρηση και ανίχνευση των παρεμβολών απόδοσης μεταξύ VMs απαιτεί την επιλογή κατάλληλων μετρικών που αντιστοιχούν στους συγκεκριμένους πόρους που ανταγωνίζονται. Υπάρχουν δύο τύποι μετρικών που μπορούν να χρησιμοποιηθούν για την ανίχνευση και μέτρηση παρεμβολών απόδοσης μεταξύ VMs: ανεξάρτητες μετρικές και παράγωγες μετρικές. Οι ανεξάρτητες μετρικές αναφέρονται σε μεταβλητές που λαμβάνονται άμεσα ως παρατηρήσεις και λειτουργούν ως έμμεσοι δείκτες παρεμβολής απόδοσης. Οι παράγωγες μετρικές είναι μεταβλητές που μπορούν να μετρήσουν άμεσα την ένταση της παρεμβολής ή το επίπεδο απώλειας απόδοσης που προκαλείται από την παρεμβολή.

1. Ανεξάρτητες Μετρικές: Η επιλογή ανεξάρτητων μετρικών συνδέεται συνήθως με τους πόρους του συστήματος. Για CPU-intensive εφαρμογές, επιλέγονται μετρικές όπως η χρησιμοποίηση CPU, ο κύκλος CPU και το ποσοστό cache/miss. Για εφαρμογές memory-intensive, οι μετρικές περιλαμβάνουν τη χρησιμοποίηση μνήμης, την ποσότητα αδρανούς μνήμης και το εύρος ζώνης μνήμης. Για I/O-intensive, εξετάζονται η χρησιμοποίηση δίσκου και τα δεδομένα I/O δίσκου.

2. Παράγωγες Μετρικές: Οι παράγωγες μετρικές προκύπτουν από υπολογισμούς βασισμένους σε κάποιες μετρικές απόδοσης. Ορισμένες εργασίες χρησιμοποιούν την απόδοση εφαρμογών ως μετρική απόδοσης, όπως ο χρόνος εκτέλεσης, ο ρυθμός επεξεργασίας εφαρμογών, ο χρόνος απόκρισης αιτημάτων και το QoS, επειδή αυτές οι μετρικές εμφανίζουν άμεσα τον αντίκτυπο της παρεμβολής στην απόδοση ενός cloud συστήματος.

Συγκεκριμένα για cpu-intensive εφαρμογές οι ερευνητές έχουν μελετήσει το πρόβλημα παρεμβολών σε μέσω διαφόρων πειραματικών προσεγγίσεων.

Οι Sun κ.ά (2014) [5] πρότειναν το μοντέλο Multivariable Exponent Interference (MVEI) για την πρόβλεψη παρεμβολών σε CPU-intensive εφαρμογές, το οποίο καταγράφει την εκθετική σχέση μεταξύ χρησιμοποίησης CPU και cache miss rate με την παρεμβολή απόδοσης. Το μοντέλο εξετάζει αυτές τις δύο μετρικές για να προβλέψει το βαθμό παρεμβολής.

Οι Nathuji κ.ά (2010) [6] ανέπτυξαν ένα MIMO (Multiple-Input Multiple-Output) feedback control μοντέλο που χρησιμοποιεί Q-states για να κατηγοριοποιήσει τα διαφορετικά επίπεδα ποιότητας υπηρεσιών. Το σύστημα παρακολουθεί συνεχώς τις μετρικές απόδοσης και προσαρμόζει δυναμικά την κατανομή πόρων για να διατηρήσει τα επιθυμητά επίπεδα QoS. Πρώτα διασφαλίζει το βασικό QoS όλων των εφαρμογών (Q0) και στη συνέχεια αξιοποιεί τους περισσευούμενους πόρους μέσω ελέγχου της αποδοτικότητας πόρων για να παρέχει υψηλότερα επίπεδα απόδοσης (Q1, Q2, Q3) όταν είναι δυνατό.

Οι Zhu κ.ά (2012) [7] στη μελέτη τους επικεντρώθηκαν στη χρονική μεταβλητότητα της χρήσης πόρων, αναπτύσσοντας στοχαστικό μοντέλο που προβλέπει την υποβάθμιση QoS λόγω παρεμβολών. Το μοντέλο υπολογίζει τους επιπλέον πόρους που απαιτούνται για την αντιμετώπιση παρεμβολών από όλους τους τύπους πόρων μέσω ενός μαθηματικού εργαλείου, τον πίνακα επιρροής (influence matrix) που εκτιμά τις αλληλεπιδράσεις μεταξύ διαφορετικών επιπέδων πόρων σε χρονικά μεταβαλλόμενα περιβάλλοντα. Ο πίνακας αυτός παράγει έναν συντελεστή διόγκωσης (dilation factor) για κάθε εφαρμογή, ο οποίος προσδιορίζει την αναγκαία αύξηση των πόρων ώστε να διατηρηθεί η επιθυμητή απόδοση παρά την παρουσία συνεγκατεστημένων εφαρμογών.

Οι Govindan κ.ά (2011) [8] διερεύνησαν την παρεμβολή από ανταγωνισμό για last level cache (LLC) μεταξύ συνεντοπισμένων VMs. Η έρευνά τους εισήγαγε το εργαλείο Cuanta που μετρά και ποσοτικοποιεί τις επιπτώσεις της κοινής χρήσης on-chip πόρων, το οποίο χρησιμοποιεί synthetic cache loader (scl), δηλαδή ένα τεχνητό φορτίο εργασίας που προσαρμόζεται για να προσπελάσει συγκεκριμένες περιοχές της last-level cache, επιτρέποντας τη δημιουργία κλόνων cache που μιμούνται την πίεση cache κάθε εφαρμογής. Μέσω αυτής της μεθόδου, επιτυγχάνεται η ακριβής πρόβλεψη των επιπτώσεων παρεμβολής από κοινόχρηστους on-chip πόρους.

Οι Pu κ.ά (2010) [9] μελέτησαν τις I/O workload παρεμβολές στο Xen VMM διαπιστώνοντας ότι τα CPU-bound και network-bound φορτία εργασίας συνδυάζονται καλύτερα από ομοίου τύπου. Η τοποθέτηση CPU-intensive εφαρμογών (μικρά αρχεία 1KB που απαιτούν γρήγορη επεξεργασία) με network-intensive εφαρμογές (μεγάλα αρχεία 100KB που καταναλώνουν bandwidth) μειώνει τον ανταγωνισμό πόρων καθώς οι εφαρμογές χρησιμοποιούν διαφορετικούς τύπους πόρων.

Οι Chiang και Huang (2011) [10] εισήγαγαν το TRACON πλαίσιο με τρία διακριτά μαθηματικά μοντέλα για την πρόβλεψη I/O παρεμβολών σε data-intensive εφαρμογές. Το Weighted Mean Method (WMM) βρίσκει παρόμοια παλιά δεδομένα βάσει Ευκλίδειας απόστασης και υπολογίζει μέσο όρο για την πρόβλεψη. Το Linear Model (LM) υποθέτει ότι η σχέση μεταξύ πόρων και απόδοσης είναι γραμμική και χρησιμοποιεί τεχνικές παλινδρόμησης (regression techniques) για

βελτιστοποίηση παραμέτρων. Το Nonlinear Model (NLM) χρησιμοποιεί πολυωνυμικές εξισώσεις για να αντιμετωπίσει τα περίπλοκα μοτίβα I/O των data-intensive εφαρμογών.

2.2.2 Στρατηγικές Βελτιστοποίησης Παρεμβολών

2.2.2.1 Σχεδιασμός Προτεραιοτήτων Εργασιών

Ο σχεδιασμός προτεραιοτήτων εργασιών αποτελεί κρίσιμο παράγοντα για τη διαχείριση παρεμβολών απόδοσης στα εικονικοποιημένα περιβάλλοντα, καθώς ορίζει τη σειρά με την οποία οι εφαρμογές κατανέμονται στους διαθέσιμους υπολογιστικούς πόρους.

Προτεραιότητα Βάσει Τύπου Πόρων

Οι Wei Zhang et al. [11] υιοθετούν μια προσέγγιση που δίνει υψηλότερη προτεραιότητα στις **CPU-intensive** καθώς οι CPU-intensive εφαρμογές τείνουν να καταλαμβάνουν τον επεξεργαστή για μεγάλα χρονικά διαστήματα. Αυτή η συμπεριφορά μπορεί να δημιουργήσει σημαντικές καθυστερήσεις στις εφαρμογές, οι οποίες απαιτούν γρήγορους χρόνους απόκρισης. Τοποθετώντας έτσι τις CPU-intensive εφαρμογές, το σύστημα διασφαλίζει ότι οι βαριές υπολογιστικές εργασίες δεν θα διακόψουν τις λειτουργίες σε πραγματικό χρόνο και ότι οι εφαρμογές θα έχουν πρόσβαση σε καθαρούς πόρους. Αυτό οδηγεί σε πιο προβλέψιμη συνολική ανταπόκριση του συστήματος.

Οι Sampaio et al. [12] προτείνουν μια αντίθετη προσέγγιση, δίνοντας προτεραιότητα στις **network-intensive** εφαρμογές αναγωρίζοντας ότι οι δικτυακές λειτουργίες έχουν συχνά αυστηρότερες απαιτήσεις σε χρόνο καθυστέρηση και εύρος ζώνης. Οι network-intensive εφαρμογές συχνά εξυπηρετούν πολλούς χρήστες ταυτόχρονα, και η υποβάθμιση της δικτυακής τους απόδοσης μπορεί να επηρεάσει σημαντικά την εμπειρία πολλαπλών χρηστών. Μια αργή δικτυακή εφαρμογή μπορεί να δημιουργήσει bottlenecks που επηρεάζουν πολλές άλλες εφαρμογές που εξαρτώνται από δικτυακές υπηρεσίες. Τέλος, οι network-intensive εφαρμογές συχνά συνδέονται με εξωτερικούς πόρους και υπηρεσίες, καθιστώντας την αποδοτική διαχείριση του δικτύου κρίσιμη για τη συνολική λειτουργία του συστήματος.

Προτεραιότητα Βάσει Χαρακτηριστικών Παρεμβολής

Οι Kim et al. [13] αναπτύσσουν ένα εξελιγμένο σύστημα που βασίζεται σε δύο μετρικές για τον χαρακτηρισμό των εφαρμογών. Η ένταση παρεμβολής μετράει πόσο επιθετική είναι μια εφαρμογή στην κατάληψη πόρων και υπολογίζεται βάσει του LLC miss rate και της συχνότητας πρόσβασης σε κοινόχρηστους πόρους. Η ευαισθησία παρεμβολής αντίστοιχα, αξιολογεί πόσο ευάλωτη είναι μια εφαρμογή σε παρεμβολές από άλλες εφαρμογές, λαμβάνοντας υπόψη την εξάρτησή της από

την απόδοση της cache και το εύρος ζώνης μνήμης. Το σύστημα προγραμματίζει τις εφαρμογές σε δύο στάδια. Στην πρώτη φάση, οι εφαρμογές τοποθετούνται σε φθίνουσα σειρά έντασης παρεμβολής, διασφαλίζοντας ότι οι επιθετικές εφαρμογές τοποθετούνται πρώτες. Στη δεύτερη φάση, οι υπόλοιπες εφαρμογές προγραμματίζονται σε αύξουσα σειρά ευαισθησίας, τοποθετώντας τις ευαίσθητες εφαρμογές σε ασφαλέστερα περιβάλλοντα. Αυτή η προσέγγιση προσφέρει πιο ακριβή χαρακτηρισμό και δυναμική προσαρμογή, καθιστώντας την ιδανική για ετερογενή περιβάλλοντα.

2.2.2.2 Πολιτικές Τοποθέτησης

Οι στρατηγικές τοποθέτησης εικονικών μηχανών καθορίζουν τον τρόπο με τον οποίο κατανέμονται τα φορτία εργασίας στους φυσικούς πόρους. Ο σχεδιασμός προτεραιότητας εργασιών επιτρέπει την κατηγοριοποίησή τους βάσει των χαρακτηριστικών παρεμβολής τους, ενώ οι πολιτικές τοποθέτησης βασίζονται σε αλγορίθμους που στοχεύουν στη μείωση του ανταγωνισμού για κοινόχρηστους πόρους.

Φιλτράρισμα Υποψήφιων Κόμβων

Πριν την έναρξη του προγραμματισμού, ορισμένοι κόμβοι αποκλείονται από τους υποψηφίους. Οι Alves et al. [14] χρησιμοποίησαν μαθηματικούς περιορισμούς για να μειώσουν τον αριθμό των υποψήφιων κόμβων, αποκλείοντας αυτούς που δεν διαθέτουν επαρκείς πόρους.

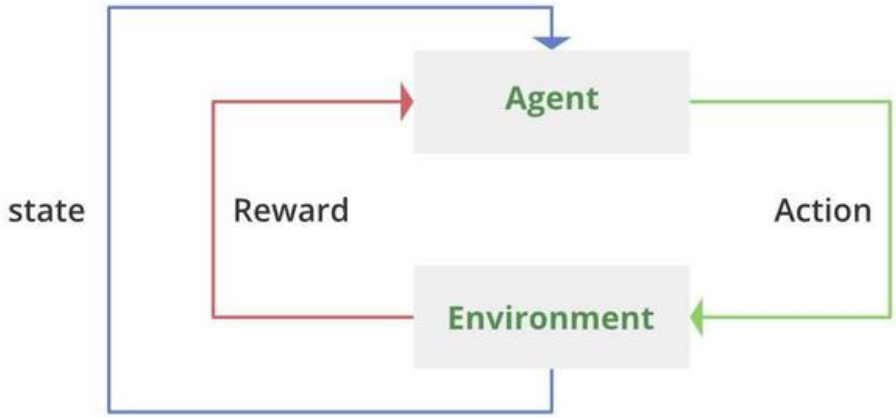
Μέθοδοι Αναζήτησης Κόμβων

Η πλειονότητα των προσεγγίσεων βασίζεται σε τοπικές μεθόδους βελτιστοποίησης, όπως ευριστικούς αλγορίθμους και μετα-ευριστικές τεχνικές, για την επιλογή κατάλληλων κόμβων. Οι ευριστικοί αλγόριθμοι είναι ιδιαίτερα δημοφιλείς λόγω του χαμηλού υπολογιστικού κόστους και της απλότητάς τους. Χαρακτηριστικά παραδείγματα αποτελούν οι αλγόριθμοι First-Fit Decreasing, Best-Fit Decreasing και Worst-Fit Decreasing, οι οποίοι κατανέμουν τα VMs ανάλογα με τις ανάγκες σε πόρους και τη διαθεσιμότητα.

2.2.2.3 Σχήματα Βελτιστοποίησης Χωρίς Προηγούμενη Γνώση

Μια σημαντική εξέλιξη στη διαχείριση παρεμβολών είναι τα αυτόνομα συστήματα που χρησιμοποιούν ενισχυτική μάθηση (reinforcement learning) [76] για να προσαρμόζονται χωρίς ανθρώπινη παρέμβαση. Η ενισχυτική μάθηση αποτελεί τεχνική τεχνητής νοημοσύνης όπου το σύστημα μαθαίνει μέσω δοκιμής και λάθους, λαμβάνοντας ανταμοιβές για σωστές αποφάσεις και ποινές για λανθασμένες. Αυτά τα συστήματα δεν απαιτούν εκ των προτέρων γνώση των χαρακτηριστικών εφαρμογών. Αντίθετα, παρακολουθούν συνεχώς τη συμπεριφορά των εφαρμογών και μαθαίνουν να εντοπίζουν καταστάσεις όπου μια εφαρμογή μπορεί να προκαλέσει

παρεμβολή σε άλλες, λαμβάνοντας αυτόματα προληπτικά μέτρα.



Σχήμα 2.3: Αλληλεπίδραση Agent-Environment στο Reinforcement Learning

Το σύστημα Stay-Away των Rameshan et al. [15] είναι ένα τέτοιο παράδειγμα το οποίο παρακολουθεί πολλές εφαρμογές στο ίδιο μηχάνημα και μαθαίνει να αναγνωρίζει επικίνδυνες καταστάσεις παρεμβολής. Το σύστημα λαμβάνει προληπτικά μέτρα προστασίας όταν εντοπίζει ότι μια εφαρμογή κινδυνεύει να επηρεαστεί από άλλες. Αυτή η προσέγγιση προσφέρει ευελιξία καθώς μπορεί να χειριστεί νέες εφαρμογές ή αλλαγές συμπεριφοράς χωρίς επαναπρογραμματισμό, καθιστώντας το σύστημα πιο αποδοτικό σε δυναμικά cloud περιβάλλοντα.

2.3 Προκλήσεις Kata Containers

Τα Kata Containers αποτελούν μια καινοτόμο τεχνολογική προσέγγιση που στοχεύει να συνδυάσει τα πλεονεκτήματα των VMs και των containers σε μία ενιαία λύση η οποία λύνει το προβλήματα μεταξύ ασφάλειας και απόδοσης. Τα συμβατικά containers εξασφαλίζουν εξαιρετικές επιδόσεις αλλά εμφανίζουν περιορισμούς στην απομόνωση λόγω της κοινής χρήσης του πυρήνα του λειτουργικού συστήματος. Αντιστρόφως, τα VMs προσφέρουν ισχυρή απομόνωση με το αντίστοιχο κόστος σε επιδόσεις και κατανάλωση πόρων. Προσπαθούν να περιορίσουν τα κενά ασφαλείας που χαρακτηρίζουν τα παραδοσιακά container runtimes, ενσωματώνοντας κάθε container μέσα σε ένα ξεχωριστό VM με το δικό της kernel. Ωστόσο, η εφαρμογή τους συνδιαστικά και με το περιβάλλον του Kubernetes, συνοδεύεται από σημαντικές προκλήσεις που σχετίζονται με την απόδοση, την κατανάλωση πόρων, την κλιμάκωση(scaling), δηλαδή δυνατότητα προσαρμογής σε αυξημένες ή μειωμένες απαιτήσεις φόρτου εργασίας (scalability). και τέλος την ενσωμάτωση. [16]

2.3.1 Χρόνος εκκίνησης

Μία από τις σημαντικότερες προκλήσεις είναι ο υψηλός χρόνος εκκίνησης. Σε αντίθεση με το προεπιλεγμένο runtime του Kubernetes, το runc, τα Kata Containers απαιτούν την εκκίνηση ενός VM για κάθε container, γεγονός που αυξάνει αισθητά το συνολικό χρόνο εκκίνησης. Οι Vincent van Rijn κ.ά διαπίστωσαν ότι τα Kata Containers εμφανίζουν χρόνους εκκίνησης περίπου 600 χιλιοστά του δευτερολέπτου, σημαντικά υψηλότερους από τα παραδοσιακά containers που ξεκινούν σε λιγότερο από 200 χιλιοστά του δευτερολέπτου. Αυτή η καθυστέρηση οφείλεται στην πολυπλοκότητα της αρχιτεκτονικής που απαιτεί τη δημιουργία τόσο namespaces όσο και την εκκίνηση ενός hypervisor με τον guest kernel του [17]. Στα πειράματα που πραγματοποιεί ο a.Sklavos [18] παρατηρεί ότι η υλοποίηση με QEMU παρουσιάζει καλύτερη συμπεριφορά σε σχέση με αυτή που βασίζεται στο Firecracker, παρόλο που έχει σχεδιαστεί για βελτιστοποιημένα boot times. Η απόδοση του Firecracker επηρεάζεται από τον περιορισμό του να υποστηρίζει αποκλειστικά block-based storage drivers, κάτι που επιβάλλει τη χρήση του devmapper αντί για το ταχύτερο overlayfs που χρησιμοποιεί το QEMU. Η εξάρτηση από το devmapper οδηγεί σε σημαντική καθυστέρηση κατά την εκκίνηση των containers, καθιστώντας την επιλογή hypervisor κρίσιμη παράμετρο για την επίτευξη αποδεκτής απόδοσης.

2.3.2 Κατανάλωση μνήμης

Ένα ακόμη ζήτημα αφορά την αυξημένη κατανάλωση μνήμης. Καθώς κάθε container εκτελείται σε ξεχωριστή εικονική μηχανή, το αποτύπωμα μνήμης είναι πολλαπλάσιο σε σχέση με τις παραδοσιακές λύσεις containerization. Σύμφωνα με τα αποτελέσματα των πειραμάτων [18], το μέσο αποτύπωμα μνήμης για ένα pod ήταν περίπου 10 MiB για το runc, ενώ για τα Kata Containers η μέση κατανάλωση έφτανε τα 152 MiB στην περίπτωση του QEMU και τα 135 MiB στην περίπτωση του Firecracker. Η αύξηση αυτή περιορίζει τη δυνατότητα φιλοξενίας μεγάλου αριθμού pods ανά worker node και επιβαρύνει την κλιμάκωση των συστημάτων.

2.3.3 CPU

Η επιβάρυνση της CPU αποτελεί επίσης σημαντική πρόκληση. Σενάρια που περιλαμβάνουν υπολογιστικά εντατικά φορτία εργασίας έδειξαν ότι τα Kata Containers εμφανίζουν απόδοση κατά 4% έως 15% χαμηλότερη σε σχέση με το runc [18]. Το επιπλέον επίπεδο εικονικοποίησης εισάγει καθυστερήσεις στην εκτέλεση των system calls, με αποτέλεσμα τη μείωση της συνολικής υπολογιστικής αποδοτικότητας. Η υποβάθμιση αυτή δεν είναι κρίσιμη σε περιβάλλοντα όπου η ασφάλεια είναι προτεραιότητα, αλλά μπορεί να αποτελέσει εμπόδιο σε σενάρια που απαιτούν υψηλή απόδοση και ταχύτητα απόκρισης.

2.3.4 Scaling

Επιπλέον, τα Kata Containers παρουσιάζουν προκλήσεις όσον αφορά την κλιμάκωση. Ο αυξημένος χρόνος εκκίνησης, σε συνδυασμό με το μεγαλύτερο αποτύπωμα μνήμης, δυσχεραίνουν την γρήγορη δημιουργία μεγάλου αριθμού pods. Παράλληλα, το Kubernetes επιβάλλει ανώτατο όριο 110 pods ανά worker node, γεγονός που περιορίζει την αξιοποίηση των διαθέσιμων πόρων όταν χρησιμοποιούνται runtimes, όπως τα Kata.

2.3.5 Ενσωμάτωση

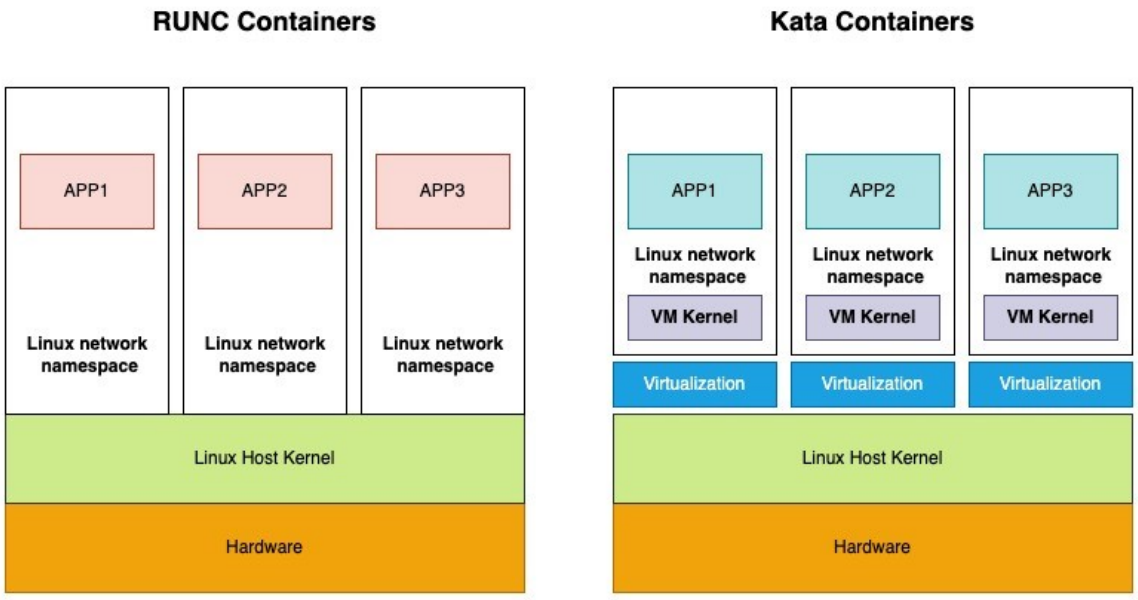
Η πολυπλοκότητα ενσωμάτωσης αποτελεί επίσης ένα ζήτημα. Η ενσωμάτωση των Kata Containers στο Kubernetes απαιτεί πρόσθετη παραμετροποίηση, ιδιαίτερα ως προς την επιλογή hypervisor, το υποστηριζόμενο filesystem και τον snapshotter. Σε πειράματα που πραγματοποιήθηκαν [18], η χρήση του devmapper με το QEMU παρουσίασε αστάθειες και απέτυχε σε ορισμένες περιπτώσεις, γεγονός που υποδεικνύει ότι η αξιοπιστία του συστήματος μπορεί να επηρεαστεί από πειραματικές ή μη πλήρως υποστηριζόμενες διαμορφώσεις.

- Τεχνικές απαιτήσεις: Η εγκατάσταση των Kata Containers με Firecracker απαιτεί εκτεταμένη διαμόρφωση του devmapper snapshotter, συμπεριλαμβανομένης της δημιουργίας block devices για metadata και αποθήκευση δεδομένων. Η διαδικασία περιλαμβάνει τη ρύθμιση thin-pool device mapper targets και την τροποποίηση του αρχείου ρυθμίσεων του containerd για να υποστηρίξει τον devmapper snapshotter αντί του προεπιλεγμένου overlayfs.
- RuntimeClass configuration: Η ενσωμάτωση στο Kubernetes απαιτεί τη δημιουργία συγκεκριμένων πόρων τύπου RuntimeClass, οι οποίοι ορίζουν τους kata-qemu και kata-fc handlers. Αυτό επιτρέπει στους χρήστες να επιλέξουν το κατάλληλο runtime μέσω του πεδίου runtimeClassName στις προδιαγραφές των pods.
- Περιορισμοί συμβατότητας: Η επιλογή συστήματος αρχείων επηρεάζει σημαντικά τη σταθερότητα του συστήματος. Το overlayfs, αν και προσφέρει καλύτερες επιδόσεις, δεν υποστηρίζεται από προεπιλογή από το Firecracker και απαιτεί ειδική διαμόρφωση για να λειτουργήσει σωστά. Από την άλλη, η χρήση του devmapper με QEMU παρουσιάζει προβλήματα αξιοπιστίας.

Kata Containers and Kubernetes

3.1 Kata Containers

Τα Kata Containers, όπως αναφέρθηκε στο Κεφάλαιο 2, εστιάζουν στη δημιουργία ασφαλούς περιβάλλοντος με ισχυρή απομόνωση φόρτου εργασίας και αναπτύσσουν ένα container runtime παρέχοντας ελαφριά VMs. Προκειμένου να το πετύχουν αυτό, χρησιμοποιούν χαρακτηριστικά του kernel όπως τα namespaces καθώς και την τεχνολογία εικονικοποίησης του υλικού [19]. Σε αντίθεση με τα απλά Linux containers όπως είναι τα runc, κάθε Kata Container εκτελείται εντός ενός αποκλειστικού VM, διαθέτοντας ανεξάρτητο λειτουργικό σύστημα και πυρήνα από αυτό του συστήματος του host. Έτσι βελτιστοποιείται ο χρόνος εκκίνησης του πυρήνα και ελαχιστοποιείται το αποτύπωμα μνήμης. Η τεχνολογία αυτή υποστηρίζει την ενσωμάτωση με πλατφόρμες ενορχήστρωσης containers, όπως το Kubernetes.



Σχήμα 3.1: Αρχιτεκτονική σύγκριση runc και Kata Containers

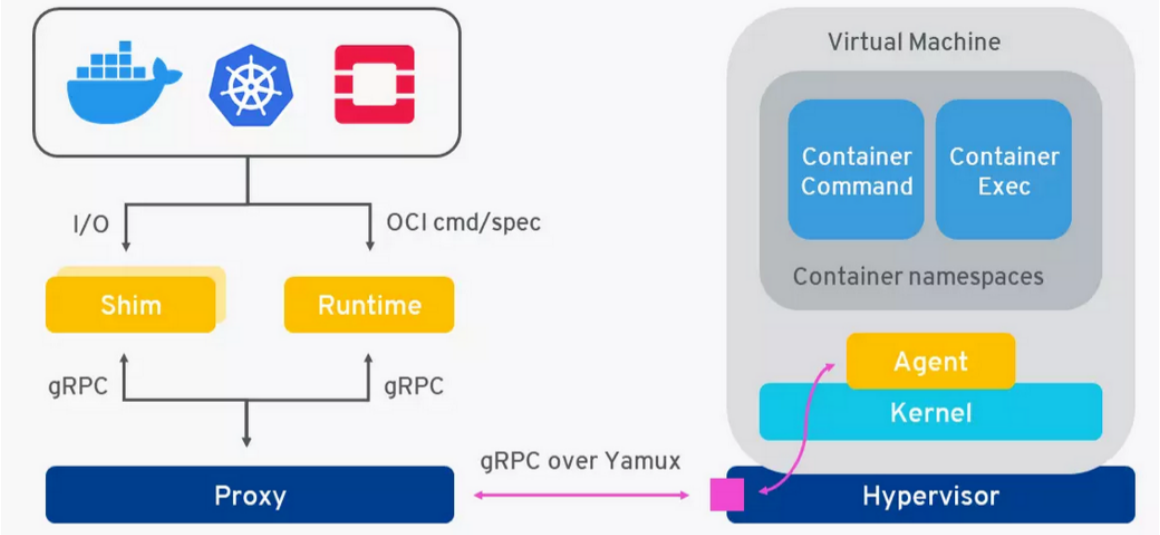
Η αρχιτεκτονική του λογισμικού των Kata Containers περιλαμβάνει τέσσερα βασικά συστατικά:

Kata-runtime: Λειτουργεί ως διεπαφή μεταξύ του περιβάλλοντος των containers όπως Docker και Kubernetes και του hypervisor. Πρόκειται για ένα container runtime συμβατό με τις προδιαγραφές OCI (Open Container Initiative), το οποίο είναι υπεύθυνο για την εκκίνηση του hypervisor, την εκτέλεση όλων των εντολών για τη δημιουργία των ελαφριών εικονικών μηχανών καθώς και την εκκίνηση των διεργασιών kata-shim.

Kata-shim: Είναι μια διεργασία που εκτελείται στον host και διαχειρίζεται όλες τις ροές I/O των containers. Εμφανίζεται σαν να είναι η ίδια η διεργασία του container η οποία στην πραγματικότητα τρέχει στην εικονική μηχανή, ώστε να μπορεί το σύστημα διαχείρισης διεργασιών του host να την αναγνωρίσει και να την παρακολουθεί.

Kata-proxy: Πρόκειται για μια διεργασία που επιτρέπει σε πολλούς πελάτες Kata-shim και Kata-runtime να έχουν πρόσβαση στον Kata-agent μέσα στο VM. Ο βασικός της ρόλος είναι η δρομολόγηση των αιτημάτων και σημάτων ανάμεσα σε κάθε Kata-shim και τον Kata-agent.

Kata-agent: Αποτελεί μια διεργασία που εκτελείται στο guest μηχάνημα εντός του VM και αποτελεί τον συντονιστή για τη διαχείριση των containers και των διεργασιών που τρέχουν μέσω του hypervisor. Διευκολύνει την επικοινωνία μεταξύ του Kata Runtime στον host και του φόρτου εργασίας του container μέσα στο VM, και εκτελεί εντολές όπως εκκίνηση, διακοπή και παρακολούθηση του container. [20]



Σχήμα 3.2: Αρχιτεκτονική λειτουργίας Kata Containers

3.2 VMM

Οι Virtual Machine Monitors (VMM) ή αλλιώς Hypervisors αποτελούν λογισμικό το οποίο μέσω ενός στρώματος εικονικοποίησης επιτρέπουν την διανομή πόρων υλικού του host, στο guest μηχάνημα, καθώς και την λειτουργία των VMs [21]. Μέσω του στρώματος εικονικοποίησης, οι hypervisors επιτυγχάνουν τη δημιουργία ανεξάρτητων και απομονωμένων υπολογιστικών περιβάλλοντων εντός του ίδιου φυσικού μηχανήματος, προσφέροντας σε κάθε VM την ψευδαίσθηση ότι διαθέτει αποκλειστική πρόσβαση στο σύνολο των διαθέσιμων πόρων. Τα θετικά αυτής της προσέγγισης είναι η μείωση του υπολογιστικού κόστους και η βελτιστοποίηση της ταχύτητας εκκίνησης των VMs. Ωστόσο η εικονοποίηση του υλικού, επιφέρει αρνητικά αποτελέσματα στην απόδοση καθώς οι λειτουργίες του hypervisor απαιτούν επιπλέον απαιτήσεις σε επεξεργαστικούς κύκλους, διαχείριση μνήμης και λειτουργίες εισόδου-εξόδου σε σύγκριση με την άμεση εκτέλεση εφαρμογών. Οι εντολές που δεν είναι εφικτό να εκτελεστούν άμεσα από τα guest περιβάλλοντα πραγματοποιούνται μέσω hypercall μηχανισμών προς τον hypervisor που είναι μια διαδικασία που προσθέτει και αυτή υπολογιστικό overhead [17].

Τα hypercalls αποτελούν ειδικούς μηχανισμούς κλήσης που επιτρέπουν στις εικονικές μηχανές να επικοινωνούν με τον hypervisor και να αιτούνται υπηρεσίες που απαιτούν προνομιούχα πρόσβαση στο υλικό [22]. Κάθε hypercall συνεπάγεται context switch από το guest περιβάλλον στον hypervisor και επιστροφή, διαδικασία που περιλαμβάνει αποθήκευση και επαναφορά καταστάσεων του επεξεργαστή, διαχείριση μνήμης και εκκαθάριση των κρυφών μνημών (cache) της CPU, εισάγοντας σημαντικό υπολογιστικό κόστος [23].

3.3 Firecracker

Το Firecracker αποτελεί ένα εξειδικευμένο Virtual Machine Monitor που βασίζεται στην τεχνολογία Kernel-based Virtual Machine (KVM) του Linux για τη δημιουργία και διαχείριση ελαφριών VM, γνωστών ως microVMs [24]. Το Firecracker αναπτύχθηκε από την Amazon για τις υπηρεσίες serverless computing της AWS, με στόχο να παρέχει μεγαλύτερη ασφάλεια και απομόνωση φόρτου εργασίας σε σύγκριση με τις παραδοσιακές εικονικές μηχανές, ενώ ταυτόχρονα να εξασφαλίζει την ταχύτητα εκκίνησης και την αποδοτικότητα πόρων των containers. Κάθε microVM του Firecracker απομονώνεται περαιτέρω με τη χρήση μηχανισμών ασφαλείας του χώρου χρήστη του Linux, μέσω ενός πολυεπίπεδου συστήματος ασφαλείας που ονομάζεται jailer.

Ο jailer λειτουργεί ως ένα επιπλέον επίπεδο άμυνας, προσφέροντας προστασία σε περίπτωση που ο μηχανισμός απομόνωσης της εικονοποίησης παραβιαστεί [25]. Δημιουργεί ένα περίβλημα γύρω από το Firecracker που το τοποθετεί σε ένα

περιοριστικό sandbox περιβάλλον πριν από την εκκίνηση του guest συστήματος. Αυτός ο μηχανισμός περιλαμβάνει την εκτέλεση του Firecracker σε chroot περιβάλλον, την απομόνωσή του σε ξεχωριστά pid και network namespaces, καθώς και την κατάργηση προνομιακών δικαιωμάτων. Με αυτόν τον τρόπο, ακόμη και αν υπάρξει παραβίαση της εικονικοποίησης, ο επιτιθέμενος θα αντιμετωπίσει πρόσθετα εμπόδια ασφαλείας που περιορίζουν σημαντικά τις δυνατότητές του να επηρεάσει το host σύστημα.

Ο έλεγχος της διεργασίας του Firecracker πραγματοποιείται μέσω ενός RESTful API, το οποίο επιτρέπει ενέργειες όπως η ρύθμιση του αριθμού των εικονικών CPUs ή η εκκίνηση της μηχανής. Οι συσκευές αποθήκευσης και δικτύου του Firecracker διαθέτουν ενσωματωμένους περιοριστές ρυθμού που διαμορφώνονται επίσης μέσω του API. Αυτοί οι περιοριστές επιτρέπουν τον καθορισμό ορίων στις λειτουργίες ανά δευτερόλεπτο και στο εύρος ζώνης για κάθε συσκευή που συνδέεται σε κάθε microVM. Διασφαλίζουν ότι οι συσκευές αποθήκευσης και δικτύου διατηρούν επαρκές εύρος ζώνης για λειτουργίες του control plane και αποτρέπουν έναν μικρό αριθμό MicroVMs υψηλού φόρτου από το να επηρεάσουν την απόδοση άλλων MicroVMs [26].

3.4 Kubernetes

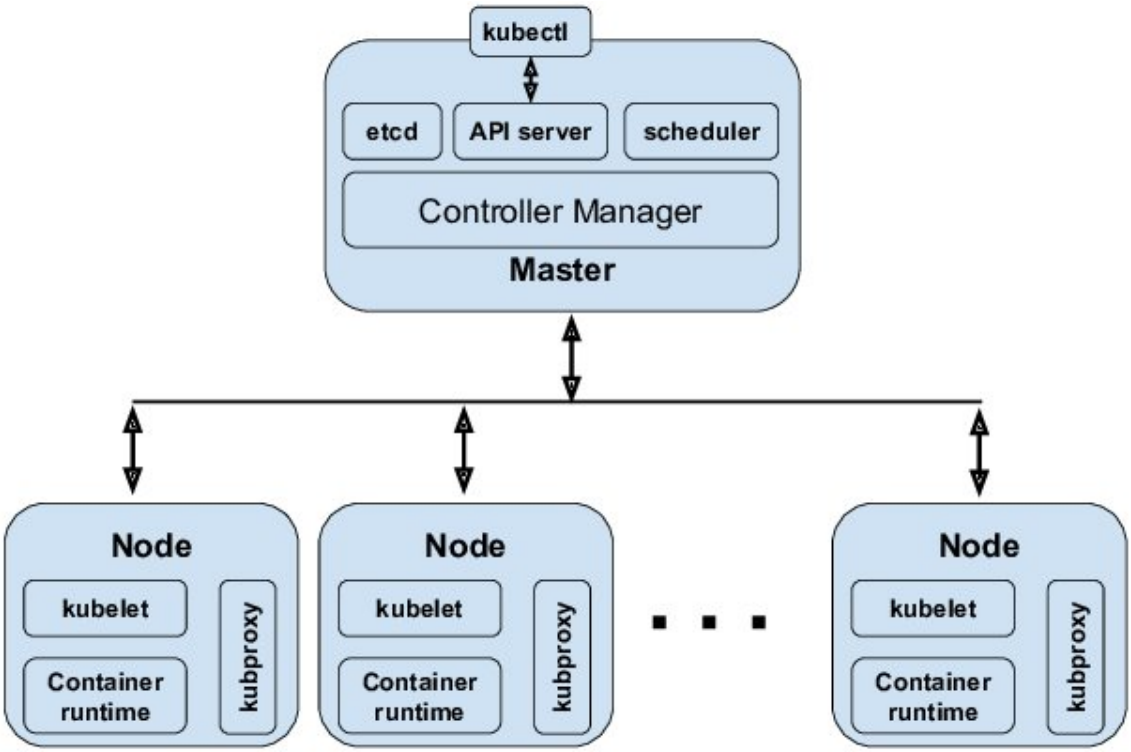
Το Kubernetes , γνωστό επίσης ως K8s ή Kube, είναι μια πλατφόρμα ανοιχτού κώδικα η οποία αποτελεί το πιο διαδεδομένο εργαλείο ενορχήστρωσης containers στις μέρες μας [27]. Ήταν το τρίτο container-management σύστημα της Google, μετά το Borg και το Omega [28].

Η πρόσβαση στην κατάσταση του συστήματος γίνεται αποκλειστικά μέσω ενός ειδικού REST API, που εφαρμόζει versions ανώτερου επιπέδου, μηχανισμούς επικύρωσης και πολιτικές, προστατεύοντας το σύστημα ενώ παράλληλα επιτρέπει σε διαφορετικούς τύπους εφαρμογών να το χρησιμοποιούν αποτελεσματικά. Το Kubernetes έχει σχεδιαστεί με έμφαση στην ευκολία των προγραμματιστών για εφαρμογές που αφορούν ένα cluster, ώστε να απλοποιεί την ανάπτυξη και διαχείριση κατανεμημένων εφαρμογών και να αξιοποιεί τα πλεονεκτήματα της αποδοτικότερης χρήσης πόρων που προσφέρουν τα containers.

3.4.1 Cluster

Ένα Kubernetes cluster αποτελείται από ένα σύνολο φυσικών ή εικονικών μηχανημάτων και άλλων υποδομών που χρησιμοποιούνται για την εκτέλεση εφαρμογών. Βασίζεται στην αρχιτεκτονική master-slave και περιλαμβάνει ένα Control Plane, το οποίο αντιστοιχεί στον master node και ένα σύνολο από Slave Nodes ή αλλιώς worker nodes, που ονομάζουμε απλούστερα ως Nodes, οι οποίοι εκτελούν containerized εφαρμογές. Κάθε cluster χρειάζεται τουλάχιστον ένα Node για να

εκτελέσει pods.



Σχήμα 3.3: Δομή cluster Kubernetes με Master και Worker Nodes

3.4.2 Pods

Τα pods είναι οι μικρότερες εκτελέσιμες μονάδες που μπορούν να δημιουργηθούν και να διαχειριστούν στο Kubernetes. Ένα pod αποτελείται από ένα ή περισσότερα containers, τα οποία μοιράζονται τους ίδιους πόρους αποθήκευσης και δικτύου, καθώς και τις προδιαγραφές για τον τρόπο εκτέλεσής τους. Προγραμματίζεται ώστε να εκτελείται σε έναν συγκεκριμένο κόμβο, γεγονός που συνεπάγεται ότι όλα τα containers που περιλαμβάνει τρέχουν στο ίδιο φυσικό μηχάνημα. Κάθε pod αποκτά μια μοναδική τοπική διεύθυνση IP στο εσωτερικό του cluster, ενώ όλα τα containers του pod μοιράζονται τον ίδιο χώρο θυρών. Ως αποτέλεσμα, δύο υπηρεσίες που απαιτούν την ίδια θύρα δεν μπορούν να συνυπάρχουν στο ίδιο pod [29].

3.4.3 Deployments

Το deployment είναι υπεύθυνο για τη δημιουργία, τη διαχείριση και το scaling των pods. Μέσω του deployment, το Kubernetes διασφαλίζει ότι ο επιθυμητός αριθμός pods είναι πάντα διαθέσιμος και αν κάποιο pod αποτύχει, το deployment αναλαμβάνει να το αντικαταστήσει αυτόματα, προσφέροντας ανθεκτικότητα στο σύστημα. Τα deployments μπορούν να δημιουργούν νέα ReplicaSets ή να διαγράφουν υπάρχοντα deployments, αναλαμβάνοντας ταυτόχρονα τη διαχείριση όλων

των σχετικών πόρων μέσω νέων deployments. Το ReplicaSet αποτελεί έναν από τους μηχανισμούς του Kubernetes που παρακολουθεί συνεχώς την κατάσταση των pods και παίρνει αυτόματα μέτρα για να διατηρήσει τον καθορισμένο αριθμό τους, λειτουργώντας ως ενδιάμεσο επίπεδο μεταξύ του Deployment και των pods. Η χρήση των deployments προσφέρει αποδοτικό scaling του συστήματος, καθώς η αύξηση ή η μείωση των απαιτούμενων πόρων μπορεί να επιτευχθεί απλώς με την αλλαγή του αριθμού των replicas των pods που δημιουργούνται [30].

3.4.4 Service

Ένα service δημιουργεί ένα σταθερό σημείο πρόσβασης προς τα pods, παρέχοντας μια σταθερή IP ή DNS διεύθυνση. Με αυτόν τον τρόπο, το service κατευθύνει τη δικτυακή κίνηση προς τα διαθέσιμα pods, ενώ όταν υπάρχουν πολλαπλά pods, λειτουργεί και ως εξισορροπητής φόρτου (load balancer), διασφαλίζοντας την ομαλή και σταθερή λειτουργία των εφαρμογών [31].

3.4.5 Control Plane Components

Το control plane του Kubernetes αποτελείται από τέσσερα βασικά συστατικά που συνεργάζονται για τη διαχείριση του cluster [32].

Kube-apiserver: Αποτελεί την κύρια διεπαφή επικοινωνίας με το cluster και λειτουργεί ως το σημείο εισόδου για όλες τις λειτουργίες. Αυτή η διεπαφή υλοποιείται ως REST API που εκθέτει όλες τις λειτουργίες του Kubernetes μέσω HTTP/-HTTPS endpoints. Όλα τα συστατικά του cluster, συμπεριλαμβανομένων του scheduler, controller manager και kubelet, καθώς και εξωτερικά εργαλεία όπως το kubectl, επικοινωνούν αποκλειστικά μέσω αυτού του API.

Etcd: Λειτουργεί ως ο κεντρικός αποθηκευτικός χώρος του cluster, όπου αποθηκεύονται και δημιουργούνται αντίγραφα ασφαλείας για όλα τα δεδομένα του. Η πρόσβαση στο etcd γίνεται αποκλειστικά μέσω του Kubernetes API server, προκειμένου να αποτραπεί η μη ασφαλής πρόσβαση στο cluster [33].

Kube-scheduler: Αποτελεί τη συσκευή λήψης αποφάσεων για την τοποθέτηση των pods. Χρησιμοποιεί εξελιγμένους αλγορίθμους που λαμβάνουν υπόψη διάφορους παράγοντες όπως τις απαιτήσεις πόρων, και την τοπολογία. Όταν δεν υπάρχει κατάλληλος node διαθέσιμος, το pod παραμένει σε κατάσταση "unscheduled" μέχρι να βρεθεί κατάλληλος κόμβος.

Kube-controller-manager: Διαχειρίζεται τους βασικούς controllers του Kubernetes οι οποίοι παρακολουθούν συνεχώς την κατάσταση του cluster μέσω του API server και πραγματοποιούν ενέργειες ώστε η τρέχουσα κατάσταση να είναι η επιθυμητή. Κάθε controller αποτελεί ξεχωριστή διεργασία αλλά για τη μείωση της

πολυπλοκότητας, όλοι οι controllers έχουν μεταγλωττιστεί σε ένα ενιαίο binary και εκτελούνται ως μία μόνο διεργασία. Παραδείγματα controllers που παρέχονται με το Kubernetes είναι ο endpoints controller, namespace controller, deployment controller, replica set controller και horizontal pod autoscaler controller.

3.4.6 Nodes

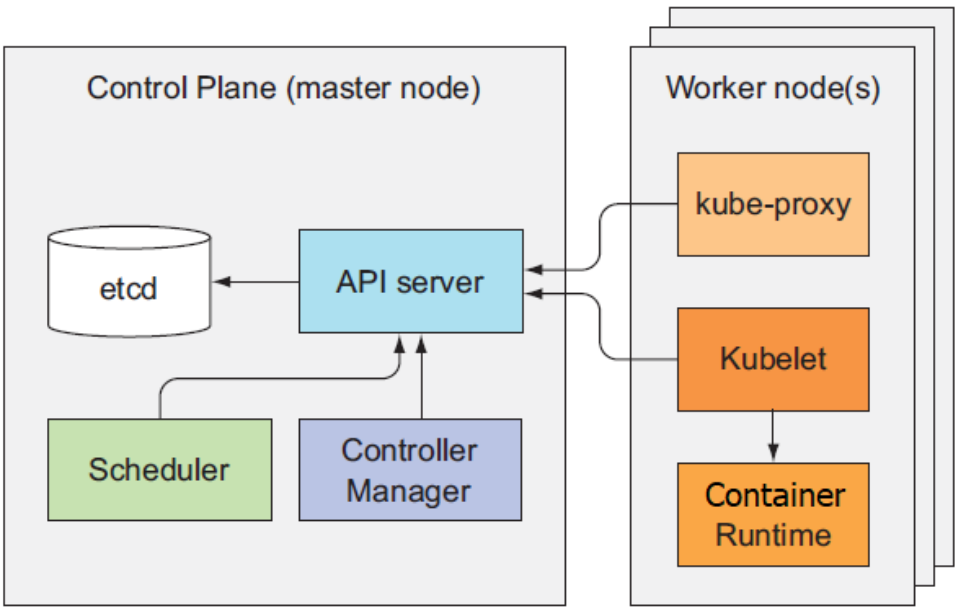
Οι worker nodes είναι φυσικά ή εικονικά μηχανήματα εξοπλισμένα με τις απαραίτητες υπηρεσίες για την εκτέλεση containers και την εκτέλεση των πραγματικών application workloads.

Kubelet: Αποτελεί τον κύριο agent κάθε κόμβου του cluster και διασφαλίζει τη σωστή λειτουργία των containers. Λαμβάνει οδηγίες για την εκτέλεσή τους και συνεργάζεται με το container runtime (όπως containerd ή CRI-O) προκειμένου να τα εκτελέσει. Το kubelet έχει τη δυνατότητα να καταχωρίσει τον κόμβο στον apiserver μέσω εναλλακτικών μηχανισμών αναγνώρισης, συμπεριλαμβανομένου του hostname του συστήματος. Το kubelet λειτουργεί με βάση τα PodSpec, τα οποία είναι αντικείμενα YAML ή JSON που περιγράφουν ένα pod. Συγκεκριμένα, το kubelet λαμβάνει ένα σύνολο PodSpecs που παρέχονται μέσω διαφόρων μηχανισμών (κυρίως μέσω του apiserver) και εξασφαλίζει ότι τα containers που περιγράφονται σε αυτά τα PodSpecs εκτελούνται και παραμένουν σε υγιή κατάσταση. Το kubelet επίσης λειτουργεί ως κανάλι επικοινωνίας μέσω του οποίου ο master επικοινωνεί με τους nodes, αποτελώντας έτσι κρίσιμο κρίκο στη λειτουργία του cluster [34].

Kube-proxy: Διαχειρίζεται τους κανόνες δικτύωσης ώστε οι συνδέσεις προς τις διευθύνσεις IP των υπηρεσιών να δρομολογούνται σωστά προς τα pods. Κάθε κόμβος του cluster διαθέτει αυτόν τον δικτυακό proxy για τη διασφάλιση της συνδεσιμότητας και την κατανομή της κίνησης προς τα διαθέσιμα pods. Μπορεί να πραγματοποιεί απλή προώθηση ροών TCP, UDP και SCTP ή round-robin κατανομή αυτών των πρωτοκόλλων προς ένα σύνολο backends. Για να ρυθμιστεί ο proxy, ο χρήστης πρέπει να δημιουργήσει μια υπηρεσία μέσω του API server [35].

Container runtime: Είναι υπεύθυνα για τη διαχείριση του κύκλου ζωής των containers, περιλαμβάνοντας τη δημιουργία, εκκίνηση, παύση και τερματισμό τους. Διαχειρίζονται τα container images και συνεργάζονται με storage drivers για τη διαχείριση των filesystem layers. Η ρύθμιση πολιτικών ασφαλείας και η εφαρμογή namespace isolation αποτελούν επίσης βασικές λειτουργίες των container runtimes. Το Kubernetes υποστηρίζει πολλαπλά container runtimes μέσω του Container Runtime Interface (CRI), το οποίο είναι ένα plugin API που επιτρέπει στο Kubernetes να χρησιμοποιεί διαφορετικά container runtimes όπως το con-

tainerd, το CRI-O και άλλα CRI-compliant runtimes [36].



Σχήμα 3.4: Αλληλεπίδραση μεταξύ master-worker node

Επιπλέον, το Kubernetes υποστηρίζει κρίσιμα πρότυπα όπως το Container Network Interface (CNI), το οποίο επιτρέπει την ενσωμάτωση διαφόρων plugins (όπως Calico, Flannel, Weave) για τη διαχείριση της επικοινωνίας μεταξύ των pods, και το Container Storage Interface (CSI), το οποίο παρέχει τη δυνατότητα υποστήριξης διαφόρων αποθηκευτικών συστημάτων μέσω ενός ενοποιημένου driver model.

3.4.7 Resource Management

Στο Kubernetes, οι βασικοί υπολογιστικοί πόροι είναι η CPU και η μνήμη, οι οποίοι αντιπροσωπεύουν μετρήσιμες ποσότητες που μπορούν να ζητηθούν και να καταναλωθούν από τα containers. Κατά τη δημιουργία ενός pod, ο χρήστης μπορεί να καθορίσει αιτήματα πόρων (resource requests) και όρια πόρων (resource limits), ώστε ο scheduler να επιλέγει τον κατάλληλο κόμβο για την εκτέλεση, διασφαλίζοντας την αποδοτική κατανομή και αποφυγή υπερφόρτωσης.

- Το **Resource Request** ορίζεται ως η ελάχιστη ποσότητα υπολογιστικών πόρων που απαιτεί ένα container για τη σωστή λειτουργία του. Ο Kubernetes scheduler αξιοποιεί αυτές τις παραμέτρους κατά τη διαδικασία δρομολόγησης για τον καθορισμό του κατάλληλου node όπου θα εκτελεστεί το pod. Τα resource requests λειτουργούν ως εγγύηση διαθεσιμότητας πόρων για το container, χωρίς ωστόσο να αποτελούν ανώτατο όριο κατανάλωσης εφόσον υπάρχουν επιπλέον διαθέσιμοι πόροι στον node.

- Το **Resource Limit** αντιστοιχεί στη μέγιστη επιτρεπόμενη ποσότητα πόρων που μπορεί να καταναλώσει ένας container. Στην περίπτωση υπέρβασης των ορίων μνήμης, το σύστημα τερματίζει το pod προκειμένου να διαφυλάξει τη σταθερότητα του node. Αντιθέτως, για υπέρβαση CPU limit, εφαρμόζεται περιορισμός (throttling) χρήσης της CPU, όπου θα αναλυθεί στο επόμενο κεφάλαιο, χωρίς τερματισμό του pod. Τα resource limits αποτελούν κρίσιμο μηχανισμό προστασίας του συστήματος από containers που ενδεχομένως να καταναλώνουν υπερβολικούς πόρους και να επηρεάζουν αρνητικά την απόδοση άλλων εφαρμογών στο ίδιο node [31].

Τα requests και limits μεταβιβάζονται στο container runtime όταν το kubelet εκκινεί το container, εξασφαλίζοντας τον έλεγχο της κατανάλωσης πόρων. Ο scheduler τοποθετεί τα pods σε κόμβους λαμβάνοντας υπόψη τη διαθέσιμη χωρητικότητα, αποφεύγοντας περιπτώσεις υπερδέσμευσης πόρων. Έτσι, ακόμη και αν η τρέχουσα χρήση είναι χαμηλή, ένα pod δεν εκτελείται σε κόμβο όπου τα ζητούμενα resources υπερβαίνουν τα επιτρεπτά όρια, προστατεύοντας το σύστημα από μελλοντικές ελλείψεις.

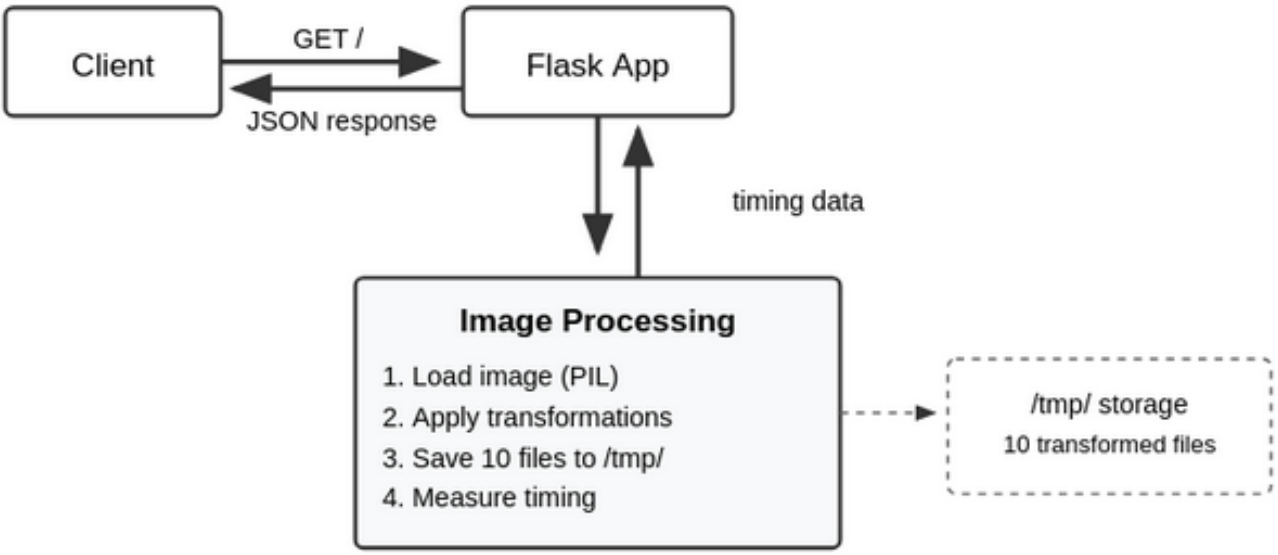
Συγκεκριμένα για τη CPU, αυτή εκφράζεται σε μονάδες πυρήνων, όπου μία μονάδα αντιστοιχεί σε έναν εικονικό επεξεργαστή vCPU ή ένα hyperthread, ενώ υποστηρίζονται και κλασματικά αιτήματα. Η μνήμη μετράται σε bytes και μπορεί να δηλωθεί είτε ως ακέραιος αριθμός είτε με δυαδικές μονάδες (Ki, Mi, Gi) ή δεκαδικές μονάδες (K, M, G).

Προτεινόμενη Μεθοδολογία

4.1 Πειραματική Διάταξη και Υλοποίηση

4.1.1 Αρχιτεκτονική Εφαρμογής

Η εφαρμογή της παρούσας εργασίας αποτελείται από τρία κύρια στοιχεία: το κεντρικό module επεξεργασίας εικόνας **function_app.py**, την Flask εφαρμογή **app.py** και το **Dockerfile**. Το module function_app.py αποτελεί τον πυρήνα της λειτουργικότητας, εκτελώντας τους μετασχηματισμούς σε κάθε εικόνα εισόδου και δημιουργώντας συνολικά 10 νέα αρχεία εικόνας στον κατάλογο /tmp/ του container. Αυτή η διαδικασία στρεσάρει σημαντικά τις I/O λειτουργίες και τη διαχείριση μνήμης των container runtimes, καθιστώντας την εφαρμογή ιδανική για την αξιολόγηση της απόδοσης. Η επικοινωνία με την εφαρμογή επεξεργασίας πραγματοποιείται μέσω του module app.py, το οποίο υλοποιεί μια Flask εφαρμογή που παρέχει RESTful API endpoint στη διεύθυνση "/". Το endpoint δέχεται GET αιτήματα και ενσωματώνει το σύστημα Prometheus metrics collection για τη συλλογή HTTP-level μετρικών απόδοσης. Συγκεκριμένα, χρησιμοποιείται ο decorator @metrics.histogram για την καταγραφή του χρόνου απόκρισης κάθε αιτήματος, επιτρέποντας τη λεπτομερή παρακολούθηση της συμπεριφοράς του συστήματος. Η μετατροπή της εφαρμογής σε container επιτυγχάνεται μέσω του Dockerfile, το οποίο βασίζεται σε Python 3.8-slim base image για ελαχιστοποίηση του μεγέθους. Το κατάλογος εργασίας ορίζεται ως /app, ενώ δημιουργούνται οι κατάλογοι /app/images και /tmp για τις ανάγκες της εφαρμογής. Εγκαθίστανται οι απαραίτητες βιβλιοθήκες Flask, Gunicorn και prometheus-flask-exporter. Το container εκθέτει την πόρτα 8080 και εκκινεί με Gunicorn ως production WSGI server, διαμορφωμένο με 2 workers και 2 threads ανά worker. Ακόμη, ορίζεται το timeout σε "0" για να αποφευχθούν χρονικοί περιορισμοί κατά τη διάρκεια της επεξεργασίας εικόνας.



Σχήμα 4.1: Διάγραμμα ροής εφαρμογής επεξεργασίας εικόνας

4.1.2 Διαμόρφωση IBench

Αναπτύχθηκε ένα ειδικό interference workload χρησιμοποιώντας το IBench tool. Η διαμόρφωση του IBench deployment περιλαμβάνει ένα container που εκτελεί CPU stress test, με resource limits 1 CPU core για το interference level 1.0, και 1.5 πυρήνες CPU για το επίπεδο interference 1.5. Το IBench container αναπτύσσεται παράλληλα με τα testing containers στο ίδιο Kubernetes node, δημιουργώντας ελεγχόμενη πίεση στους διαθέσιμους πόρους.

4.1.3 Container Deployments

Η ανάπτυξη των δύο container runtimes στο Kubernetes πραγματοποιείται μέσω δύο ξεχωριστών αρχείων YAML. Το αρχείο **container-deployment.yaml** ορίζει την ανάπτυξη των regular containers που χρησιμοποιούν το προεπιλεγμένο runc container runtime, δημιουργώντας ένα pod με το όνομα image-processing-container που εκτελεί την εφαρμογή επεξεργασίας εικόνας. Για την εξωτερική πρόσβαση στην εφαρμογή, το Service object δημιουργεί ένα NodePort service στην πόρτα 31397, χρησιμοποιώντας selector για να κατευθύνει την κίνηση στα pods με το σωστό label app διασφαλίζοντας έτσι την σωστή δρομολόγηση των αιτημάτων. Παράλληλα, το αρχείο **kata-deployment.yaml** περιέχει ουσιαστικά ίδια διαμόρφωση με το regular container deployment, με την κρίσιμη διαφορά της προσθήκης του πεδίου runtimeClassName: kata-fc. Με αυτό τον τρόπο το Kubernetes καθοηγείται να χρησιμοποιήσει το Kata Container runtime αντί του προεπιλεγμένου runc runtime, με το kata-fc runtime class να αντιστοιχεί στη Firecracker-based υλοποίηση του Kata Containers που χρησιμοποιεί lightweight virtual machines για την απομόνωση των containers. Το αντίστοιχο service εκθέτει την Kata-based εφαρμογή στην πόρτα 30885, διαχωρίζοντας την κίνηση

μεταξύ των δύο διαφορετικών container runtime υλοποιήσεων και επιτρέποντας την ταυτόχρονη αξιολόγησή τους.

RuntimeClass Configuration

Το κρίσιμο στοιχείο για την ενεργοποίηση των Kata Containers είναι ο πόρος RuntimeClass που ορίζεται στο αρχείο **kata-runtime-class.yaml**. Ο χειριστής kata-fc καθοδηγεί το containerd να χρησιμοποιήσει το Kata runtime αντί του προεπιλεγμένου runc, ενεργοποιώντας έτσι την υλοποίηση βασισμένη στο Firecracker.

4.1.4 Υποδομή Παρακολούθησης

Για την παρακολούθηση της απόδοσης των δύο container runtimes, ορίζονται πόροι ServiceMonitor μέσω των αρχείων **servicemonitor.yaml** και **servicemonitorkata.yaml**, οι οποίοι ενσωματώνονται με το σύστημα παρακολούθησης Prometheus. Αυτές οι ρυθμίσεις επιτρέπουν την αυτόματη συλλογή μετρικών από τα endpoints /metrics των εφαρμογών, διασφαλίζοντας τη συνεχή καταγραφή δεδομένων απόδοσης κατά τη διάρκεια των πειραμάτων.

Το πειραματικό περιβάλλον εκτελείται σε εικονική μηχανή του πανεπιστημίου με διεύθυνση IP 147.102.13.62, στην οποία πραγματοποιείται απομακρυσμένη πρόσβαση μέσω SSH. Για την πρόσβαση στο Grafana dashboard του Prometheus, δημιουργείται SSH tunnel που προωθεί την πόρτα 3000 του τοπικού συστήματος στην πόρτα 3000 της εικονικής μηχανής, σε συνδυασμό με kubectl port-forward για την προώθηση της πόρτας του Prometheus-Grafana service, επιτρέποντας την απομακρυσμένη παρακολούθηση μέσω του τοπικού browser.

Η δημιουργία φορτίου για την αξιολόγηση των runtimes πραγματοποιείται μέσω του εργαλείου Vegeta load testing, το οποίο χρησιμοποιεί προκαθορισμένους στόχους HTTP που ορίζονται στα αρχεία **container_targets.txt** και **kata_targets.txt**. Και τα δύο αρχεία κατευθύνουν προς την ίδια εφαρμογή επεξεργασίας εικόνας, αλλά μέσω διαφορετικών NodePort services: το container target χρησιμοποιεί την πόρτα 31397 για το regular container deployment, ενώ το kata target χρησιμοποιεί την πόρτα 30885 για το Kata container deployment. Τα αρχεία περιλαμβάνουν τη διεύθυνση IP 147.102.13.62, η οποία αντιστοιχεί στον κόμβο Kubernetes όπου εκτελούνται τα deployments και επιτρέπει την εξωτερική πρόσβαση από το Vegeta tool που εκτελείται εκτός του cluster και διασφαλίζοντας ότι οι μετρήσεις δεν επηρεάζονται από την εσωτερικό δίκτυο του Kubernetes.

4.2 Μεθοδολογία Πειραματικής Αξιολόγησης

4.2.1 Disk and Network I/O

Η απόδοση των εφαρμογών σε containerized περιβάλλοντα επηρεάζεται σημαντικά από τον τρόπο πρόσβασης στα δεδομένα. Δύο κύρια patterns I/O που χαρακτηρίζουν τις σύγχρονες εφαρμογές είναι το **Disk I/O** και το **Network I/O**.

Το υπολογιστικό φορτίο της εφαρμογής μας συνδυάζει CPU processing για τις μετατροπές εικόνων (flip, rotate, filter, grayscale conversion και resize) με εκτεταμένες I/O operations. Η εφαρμογή δέχεται ως input μια εικόνα από το local filesystem path (GET endpoint) και από την επεξεργασία προκύπτουν 10 παράγωγες εικόνες που αποθηκεύονται στο /tmp/ directory του container, καθιστώντας την εφαρμογή I/O intensive λόγω του μεγάλου αριθμού λειτουργιών disk read/write.

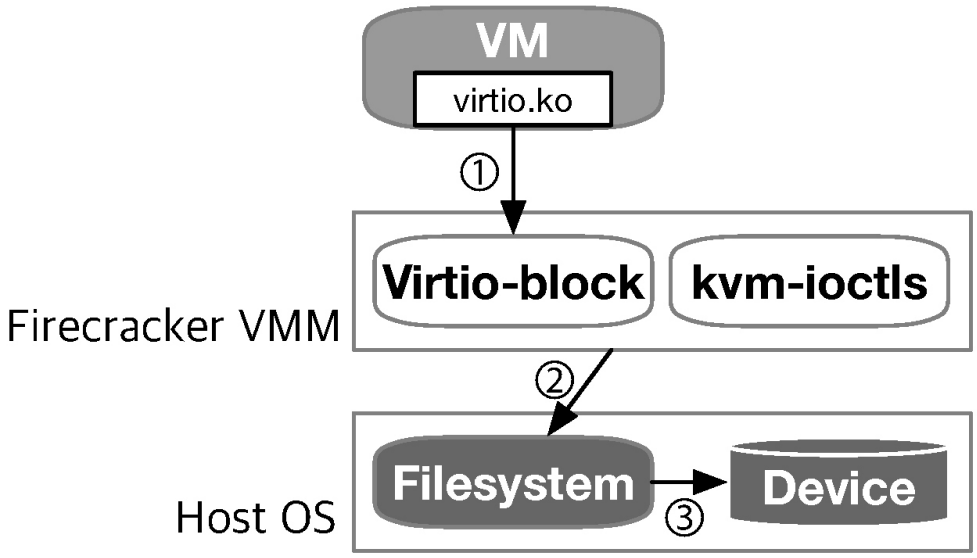
Disk I/O Characteristics

Το Disk I/O περιλαμβάνει εργασίες που απαιτούν άμεση αλληλεπίδραση με το storage subsystem του host όπως read, write και input/output operations σε δίσκους, όπως έχουμε και στην παρούσα εργασία.

Η αρχιτεκτονική του filesystem διαφέρει αρκετά μεταξύ Standard και Kata containers. Τα κανονικά containers χρησιμοποιούν την τεχνολογία overlay2 filesystem, η οποία λειτουργεί σε επίπεδο αρχείων και όχι σε επίπεδο blocks, χρησιμοποιώντας τη μνήμη πιο αποδοτικά. Αυτό επιτρέπει άμεση πρόσβαση στο κεντρικό ext4 filesystem του host συστήματος. Τα Kata Containers με Firecracker χρησιμοποιούν μια ειδική αρχιτεκτονική I/O που βασίζεται στις virtio block συσκευές.

Οι virtio block συσκευές λειτουργούν ως ένα ζεύγος drivers που επικοινωνούν μεταξύ του VM και του host OS. Το virtio front-end driver βρίσκεται μέσα στο VM και είναι υπεύθυνο για την αποστολή και λήψη I/O αιτημάτων προς και από το host OS. Το virtio back-end driver βρίσκεται στο host OS και εκτελεί τις πραγματικές read/write operations στις αποθηκευτικές συσκευές. Το Firecracker ανέπτυξε έναν Virtual Machine Monitor (Firecracker VMM) που χρησιμοποιεί το virtio back-end driver για να επεξεργάζεται τα I/O αιτήματα από τα VMs. Αυτή η προσέγγιση διαφέρει από τις παραδοσιακές hypervisor λύσεις και στοχεύει στη βελτιστοποίηση της απόδοσης για serverless και container workloads.

Όταν μια εφαρμογή που εκτελείται μέσα σε ένα Firecracker VM υποβάλλει I/O αιτήματα, το virtio block στον VMM λαμβάνει τα αιτήματα ασύγχρονα μέσω του virtio front-end driver (virtio.ko) του VM και στη συνέχεια, για να χειριστεί αυτά τα αιτήματα, το virtio block εκτελεί τις file I/O functions του host OS [37].



Σχήμα 4.2: Αρχιτεκτονική virtio-block για Disk I/O στο Firecracker

Στην περίπτωσή μας κάθε φορά που η εφαρμογή επεξεργασίας εικόνας αποθηκεύει ένα από τα 10 μετασχηματισμένα αρχεία στο /tmp/ filesystem, το αίτημα πρέπει να περάσει από όλα αυτά τα στάδια virtualization. Σε αντίθεση με τα regular containers που γράφουν απευθείας στο host filesystem, τα Kata containers με Firecracker πρέπει να διαχειριστούν την επιπλέον πολυπλοκότητα του virtio stack.

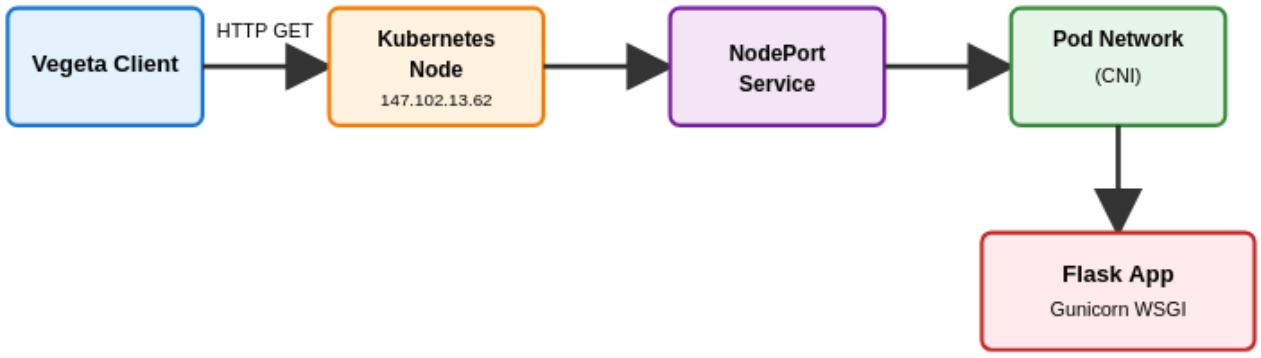
Σε I/O intensive workloads, τα Kata containers παρουσιάζουν σημαντική μείωση του disk I/O bandwidth και αύξηση του commit latency σε σχέση με τα standard containers, όπου το commit latency αναφέρεται στον χρόνο που απαιτείται για την ολοκλήρωση μιας write operation και τη διασφάλιση ότι τα δεδομένα έχουν αποθηκευτεί με ασφάλεια. Η αιτία βρίσκεται στο γεγονός ότι οι οι λειτουργίες αρχείων στα Kata containers διέρχονται από πρόσθετα επίπεδα virtualization δημιουργώντας επιπλέον υπολογιστική επιβάρυνση [38].

Χαρακτηριστικά Network I/O

Το Network I/O αποτελεί το δεύτερο κρίσιμο στοιχείο της I/O απόδοσης στην εφαρμογή μας, καθώς κάθε HTTP request που λαμβάνει η Flask εφαρμογή περιλαμβάνει λειτουργίες δικτύου για τη λήψη του αιτήματος και την αποστολή της JSON απόκρισης που περιέχει τους χρόνους απόδοσης.

Κάθε HTTP GET request που στέλνεται από το Vegeta load testing tool προς τα Kubernetes NodePort services περιλαμβάνει την ακόλουθη διαδρομή: Vegeta client, Kubernetes Node IP (147.102.13.62), Kubernetes Service (NodePort routing), Pod network interface μέσω του Container Network Interface (CNI), Flask application εντός του container. Η JSON απόκριση ακολουθεί την αντίστροφη πορεία. Για τα Kata containers, κάθε ένα από αυτά τα βήματα

περιλαμβάνει επιπλέον VM virtualization overhead.



Σχήμα 4.3: Ροή HTTP Αιτημάτων

4.2.2 Pinning

Η δέσμευση CPU (**CPU Pinning**) αποτελεί μια τεχνική διαχείρισης υπολογιστικών πόρων που επιτρέπει την στατική ανάθεση συγκεκριμένων CPU σε VMs ή containers, αγνοώντας τους δυναμικούς μηχανισμούς scheduling [39].

Το Kubernetes υλοποιεί το CPU pinning μέσω του CPU Manager Policy. Το CPU Manager Policy αποτελεί ένα component του kubelet που είναι υπεύθυνο για την διαχείριση και ανάθεση των CPU resources στα containers που εκτελούνται σε έναν node. Όταν ρυθμίζεται σε "static" mode, το kubelet μπορεί να αποδίδει αποκλειστικούς CPU cores σε pods που ικανοποιούν συγκεκριμένα κριτήρια. Συγκεκριμένα, τα pods πρέπει να ανήκουν στην "Guaranteed" Quality of Service κλάση που σημαίνει ότι τα CPU requests και limits πρέπει να είναι ίσα και να αντιπροσωπεύουν ακέραιες τιμές cores [40].

Ενεργοποιώντας το CPU Manager static policy στο Kubernetes μπορούμε να έχουμε τα παρακάτω οφέλη [41]:

Αποκλειστική Κατανομή Πόρων

Τα workload containers λαμβάνουν αποκλειστική πρόσβαση σε συγκεκριμένους CPU cores χωρίς να τους διαμοιράζονται με άλλα containers. Αυτή η απομόνωση εξασφαλίζει ότι οι εφαρμογές δεν υποβαθμίζονται από συνυπάρχοντα workloads τα οποία ενδέχεται να καταναλώνουν δυσανάλογα μεγάλο όγκο πόρων.

Μείωση του Interference πόρων

Η στατική ανάθεση CPUs επιτρέπει το partitioning των πόρων μεταξύ των workloads, μειώνοντας την interference όχι μόνο στο επίπεδο των CPU cores αλλά και στις ιεραρχίες cache και το εύρος ζώνης της μνήμης. Αυτό οδηγεί σε πιο προβλέψιμη απόδοση καθώς κάθε workload έχει εγγυημένη πρόσβαση σε συγκεκριμένους

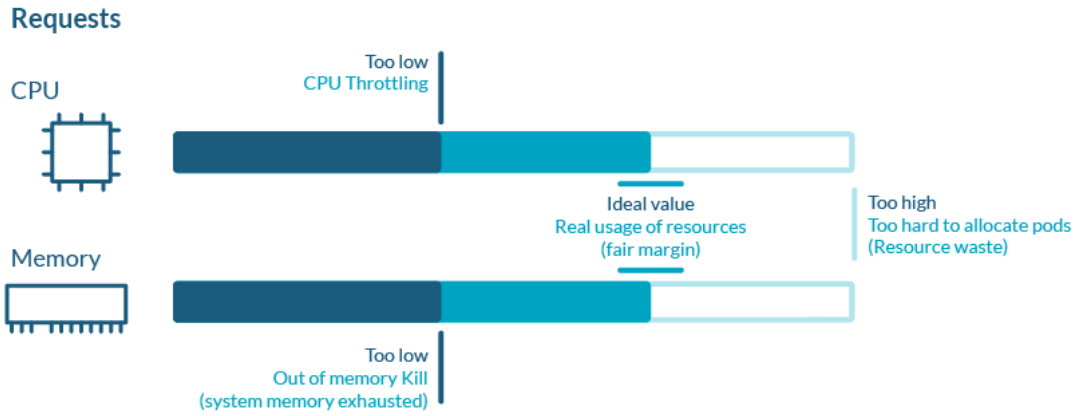
πόρους.

Βελτιστοποιημένη Τοπολογική Ανάθεση

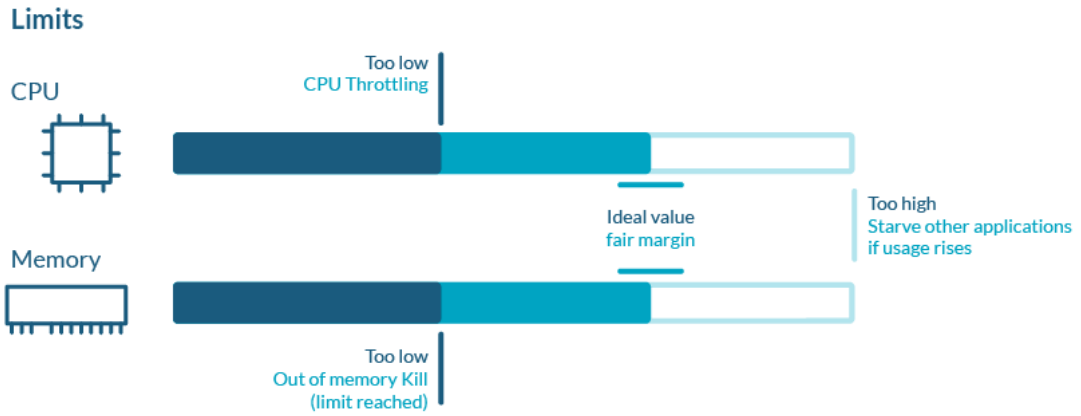
Ο CPU Manager εφαρμόζει στρατηγικές ανάθεσης που λαμβάνουν υπόψη την τοπολογία του hardware. Αναθέτει όλες τις CPUs από ένα ολόκληρο socket στο φόρτο εργασίας, αποφεύγοντας την επικοινωνία μεταξύ διαφόρων sockets που επιβάλλει επιπλέον χρονική καθυστέρηση.

Εξάλειψη του CFS Throttling

Τα containers σε Guaranteed QoS pods αποφεύγουν το throttling που μπορεί να επηρεάσει τις εφαρμογές με υψηλό φόρτο εργασίας. Το CPU throttling είναι ένας μηχανισμός του Linux CFS(Completely Fair Scheduler) scheduler που παγώνει προσωρινά την εκτέλεση ενός process όταν έχει καταναλώσει το allocated CPU quota του εντός μιας χρονικής περιόδου, αναγκάζοντάς το να περιμένει μέχρι την επόμενη περίοδο scheduling [42].



Σχήμα 4.4: CPU Requests



Σχήμα 4.5: CPU Limits

Η τεχνική του CPU pinning μπορεί να μειώσει και το overhead που προκαλείται από τη μετανάστευση των διεργασιών μεταξύ των πυρήνων CPU σε κάθε πράξη scheduling. Συγκεκριμένα, η μετακίνηση μιας διεργασίας επιβαρύνει το σύστημα με περιττές προσβάσεις στη μνήμη λόγω cache misses και με την ανάγκη επαναδημιουργίας των interrupts για τις I/O λειτουργίες.

Στην εργασία μας εφαρμόστηκε CPU pinning στα benchmark tests. Συγκεκριμένα, τα containers διαμορφώθηκαν με Guaranteed QoS class, ορίζοντας requests και limits στα 2 πυρήνων CPU και 2Gi μνήμης. Αυτή η διαμόρφωση, σε συνδυασμό με το CPU Manager static policy του Kubernetes, εξασφάλισε την αποκλειστική ανάθεση συγκεκριμένων πυρήνων CPU σε κάθε pod, εξαλείφοντας το overhead του dynamic scheduling και το CFS throttling που παρατηρείται σε non-guaranteed workloads.

4.2.3 Μετρικές Απόδοσης

Η ανάλυση της απόδοσης των container runtimes βασίστηκε σε μια πολυεπίπεδη μεθοδολογία συλλογής μετρήσεων, η οποία καταγράφει δεδομένα σε διαφορετικά επίπεδα του συστήματος. Στο επίπεδο της εφαρμογής, η Flask εφαρμογή καταγράφει αναλυτικές χρονικές μετρήσεις για κάθε στάδιο της επεξεργασίας εικόνας. Η μετρική *Load Time* μετρά τον χρόνο που απαιτείται για το άνοιγμα και το parsing του αρχείου εικόνας, η μετρική *Processing Time* καλύπτει όλες τις μετατροπές εικόνων και την αποθήκευση των 10 παράγωγων αρχείων στο filesystem, ενώ η μετρική "Total Time" αντιπροσωπεύει τη συνολική διάρκεια από την έναρξη έως την ολοκλήρωση της διαδικασίας. Οι μετρήσεις αυτές επιστρέφονται σε κάθε HTTP response ως JSON object.

Στο επίπεδο HTTP, το εργαλείο φόρτωσης Vegeta συλλέγει μετρικές όπως το *Response Time*, που μετρά τη συνολική καθυστέρηση από την αποστολή του αιτήματος έως τη λήψη της απάντησης, το *Success Rate*, που αποτυπώνει το ποσοστό των επιτυχημένων αιτημάτων, και το Throughput, που καταγράφει τον αριθμό αιτημάτων που ολοκληρώνονται ανά δευτερόλεπτο. Η διαφορά μεταξύ του HTTP response time και του application processing time ορίζει τη μετρική *Infrastructure Overhead*, η οποία αποκαλύπτει το επιπλέον κόστος που εισάγουν το container runtime και η υποδομή του Kubernetes.

Στο επίπεδο συστήματος, το εργαλείο Linux perf συλλέγει μετρικές κατά την εκτέλεση των load tests. Οι μετρικές *CPU Cycles* και *Instructions Executed* καταγράφουν την υπολογιστική δραστηριότητα, ενώ το *Instructions Per Cycle (IPC)* υποδεικνύει την αποδοτικότητα χρήσης της CPU. Οι μετρικές Context Switches και CPU Migrations αποτυπώνουν τη συχνότητα αλλαγών στον χρονοπρογραμματισμό διεργασιών, ενώ οι Cache References και Cache Misses παρέχουν πληροφορίες για την αποδοτικότητα της αξιοποίησης της cache και τον αντίκτυπο που έχει η μετακίνηση διεργασιών στη χωρική τοπικότητα της cache.

Πειραματική Αξιολόγιση

5.1 Σχεδιασμός Πειραμάτων

Τα πειράματα σχεδιάστηκαν για να αξιολογήσουν την απόδοση των container runtimes υπό διαφορετικές συνθήκες διάθεσης πόρων και CPU pinning. Συγκεκριμένα, εκτελέστηκαν τρία σετ πειραμάτων:

Πειράματα χωρίς CPU Pinning: Στο πρώτο σετ πειραμάτων, τα containers δεν είχαν εφαρμοσμένο CPU pinning και δοκιμάστηκαν με τρία επίπεδα πόρων. Στη διαμόρφωση περιορισμένων πόρων, τα containers δέσμευαν πυρήνα 0.5 πυρήνα CPU και 512Mi μνήμης, στη διαμόρφωση μεσαίων πόρων, οι πόροι αυξήθηκαν σε 1 πυρήνα CPU και 1Gi μνήμης, και τέλος στη διαμόρφωση αυξημένων πόρων, τα containers χρησιμοποίησαν 1.5 πυρήνες CPU και 1.5Gi μνήμης.

Πειράματα με CPU Pinning για Αμφότερα τα Runtimes: Στο δεύτερο σετ πειραμάτων, εφαρμόστηκε CPU pinning και στα δύο runtimes χρησιμοποιώντας το static policy του Kubernetes CPU Manager. Τα containers διαμορφώθηκαν με Guaranteed QoS class, ορίζοντας requests και limits στους 2 πυρήνες CPU και 2Gi μνήμης. Για την αξιολόγηση της συμπεριφοράς υπό συνθήκες διεκδίκησης πόρων, χρησιμοποιήθηκε τεχνητό φορτίο χρησιμοποιώντας το εργαλείο iBench, το οποίο δημιουργούσε παρεμβολή CPU. Εκτελέστηκαν δύο σενάρια: ένα με 1 και με 1.5 πυρήνες CPU για παρεμβολή, προσομοιώνοντας την παρουσία άλλων workloads στο σύστημα.

Πειράματα με CPU Pinning Μόνο για Kata: Στο τρίτο σετ πειραμάτων, εφαρμόστηκε CPU pinning αποκλειστικά στα Kata containers, ενώ τα runc containers λειτουργούσαν χωρίς pinning. Τα Kata containers διαμορφώθηκαν με 2 πυρήνες CPU και 2Gi μνήμης, και δοκιμάστηκαν επίσης με 1 και 1.5 πυρήνες CPU για παρεμβολή από το iBench. Αυτή η διαμόρφωση στόχευε στην αξιολόγηση του κατά πόσο το CPU pinning μπορεί να λειτουργήσει ως τεχνική βελτιστοποίησης της απόδοσης σε περιβάλλοντα βασισμένα σε containers, λαμβάνοντας υπόψη τις ιδιαιτερότητες του Kata runtime και τα επιπλέον επίπεδα εικονικοποίησης.

5.2 Μεθοδολογία Μέτρησης

Το σύστημα παρακολούθησης βασίζεται στο Prometheus και το Grafana, τα οποία αποτελούν πρότυπα εργαλεία για την παρακολούθηση εφαρμογών σε Kubernetes.

Η **Flask** εφαρμογή χρησιμοποιεί τη βιβλιοθήκη prometheus-flask-exporter για την αυτόματη εξαγωγή μετρικών HTTP requests. Η βιβλιοθήκη καταγράφει αυτόματα τον αριθμό των requests ανά HTTP method καθώς και τους χρόνους απόκρισης σε μορφή ιστογράμματος. Επιπλέον, προστέθηκαν προστέθηκαν προσαρμοσμένα αντικείμενα ιστογράμματος για την καταγραφή των χρόνων φόρτωσης και επεξεργασίας.

Τα **δοκιμές φόρτου** εκτελέστηκαν με το εργαλείο Vegeta, το οποίο δημιουργεί κίνηση HTTP συγκεκριμένο ρυθμό αιτημάτων και διάρκεια. Τα τεστ έγιναν σε 4 επίπεδα φορτίου, σε όλες τις περιπτώσεις, με διάρκεια πέντε λεπτά το καθένα. Κάθε τεστ εκτελέστηκε ξεχωριστά για κάθε container runtime με επαρκή χρονική απόσταση μεταξύ τους. Η μεθοδολογία περιλάμβανε τη δυναμική δημιουργία και διαγραφή των pods για κάθε τεστ, αποφεύγοντας την παράλληλη εκτέλεση των δύο runtimes που θα δημιουργούσε σύγκρουση πόρων. Μετά τη δημιουργία κάθε pod, εφαρμοζόταν περίοδος αναμονής 30 δευτερολέπτων για την πλήρη αρχικοποίηση του container και την ετοιμότητα του service endpoint. Μετά την ολοκλήρωση κάθε τεστ, το pod διαγράφονταν και το σύστημα αφήνονταν να σταθεροποιηθεί για 120 δευτερόλεπτα πριν από το επόμενο test. Κατά τη διάρκεια κάθε τεστ, το Linux perf εκτελούνταν παράλληλα για τη συλλογή μετρητών απόδοσης υλικού, καταγράφοντας μετρικές όπως CPU cycles, instructions executed, context switches και cache misses. Τα πειράματα με παρεμβολές από το iBench, εκτελούνταν παράλληλα με τις δοκιμές φόρτου, με το iBench να ξεκινά πριν την έναρξη του Vegeta test και να τερματίζεται μετά την ολοκλήρωσή του. Το iBench διαμορφώθηκε να καταναλώνει τον καθορισμένο αριθμό CPU cores συνεχόμενα, δημιουργώντας ρεαλιστικά σενάρια ανταγωνισμού για πόρους.

Το **Grafana** χρησιμοποιήθηκε για τη δημιουργία διαδραστικών dashboards που παρέχουν οπτικοποίηση ,πραγματικού χρόνου, των μετρικών κατά τη διάρκεια των τεστ.

Για τον υπολογισμό της μέσης καθυστέρησης, η συνάρτηση rate εφαρμόζεται τόσο στο άθροισμα όσο και στο πλήθος των requests σε ένα κυλιόμενο χρονικό παράθυρο ενός λεπτού, και στη συνέχεια υπολογίζεται ο λόγος των δύο ρυθμών μέσω του query:

rate(image_processing_duration_seconds_sum[1m]) /
rate(image_processing_duration_seconds_count[1m])

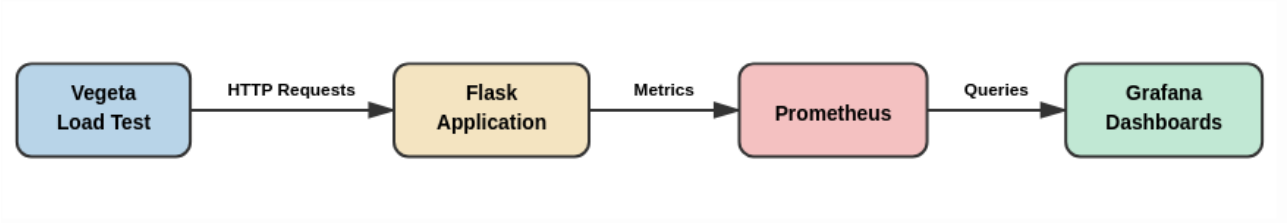
Η μετρική CPU utilization υπολογίζεται ως το ποσοστό της πραγματικής χρήσης CPU σε σχέση με τα ορισμένα resource limits του pod. Για τα runc containers, το αντίστοιχο query είναι:

*round(100 * sum(rate(container_cpu_usage_seconds_totalpod= "image-processing-container.*",container=""[40s])) by (pod) /*
sum(kube_pod_container_resource_limitspod= "image-processing-container.", resource="cpu") by (pod))*

ενώ για τα Kata containers χρησιμοποιείται το query:

*round(100*sum(rate(container_cpu_usage_seconds_totalpod= "image-processing-kata.*", container=""[40s]))by (pod) /*
sum(kube_pod_container_resource_limitspod= "image-processing-kata.", resource="cpu")by (pod)).*

Το χρονικό παράθυρο των 40s επιλέχθηκε για να παρέχει επαρκή εξομάλυνση των στιγμιαίων διακυμάνσεων χωρίς να εισάγει υπερβολική καθυστέρηση στην αποτύπωση των αλλαγών στη χρήση CPU.



Σχήμα 5.1: Διάγραμμα παρακολούθησης απόδοσης

5.3 Πείραμα Α: Απόδοση Εφαρμογών σε Διαφορετικά Επίπεδα Πόρων

Περιορισμένοι πόροι: 500m CPU, 512Mi μνήμη

Rate (RPS)	Runtime	Mean Latency (ms)	Throughput (req/s)	Success Rate
2	Regular	41.256	2.00	100%
2	Kata	48.836	2.00	100%
4	Regular	41.592	4.00	100%
4	Kata	59.865	4.00	100%
6	Regular	40.945	6.00	100%
6	Kata	61.971	6.00	100%
8	Regular	39.944	8.00	100%
8	Kata	117.794	8.00	100%

Πίνακας 5.1: Μετρικές απόδοσης για διαφορετικούς ρυθμούς αιτημάτων (Περιορισμένοι πόροι)

Στον Πίνακα 5.1 όπου φαίνονται τα αποτελέσματα του Vegeta load testing tool, παρατηρείται ότι τα regular containers διατηρούν σταθερή μέση καθυστέρηση 39.9-41.6 ms. Αντίθετα, τα Kata containers παρουσιάζουν τριπλάσια καθυστέρηση από 48.8 ms στα 2 RPS έως 117.8 ms στα 8 RPS. Αυτή η απότομη αύξηση υποδηλώνει ότι το overhead εικονικοποίησης των Kata containers δημιουργεί bottleneck που επιδεινώνεται, παρά το 100% ποσοστού επιτυχίας. Η σταθερή διατήρηση της ρυθμαπόδοσης και στα δύο runtimes επιβεβαιώνει ότι όλα τα αιτήματα εξυπηρετούνται.

Rate (RPS)	Runtime	Avg Load Time (s)	Avg Processing Time (s)	Avg Total Time (s)
2	Regular	0.0003	0.0382	0.0387
2	Kata	0.0004	0.0438	0.0444
4	Regular	0.0003	0.0385	0.0390
4	Kata	0.0007	0.0515	0.0531
6	Regular	0.0002	0.0379	0.0384
6	Kata	0.0005	0.0526	0.0535
8	Regular	0.0002	0.0373	0.0377
8	Kata	0.0007	0.0782	0.0798

Πίνακας 5.2: Ανάλυση απόδοσης σε επίπεδο εφαρμογής (Περιορισμένοι πόροι)

Rate (RPS)	Container	Mean Latency (ms)	Total Processing (ms)	Overhead (ms)
2	Regular	41.256	38.7	2.6
2	Kata	48.836	44.4	4.4
4	Regular	41.592	39.0	2.6
4	Kata	59.865	53.1	6.8
6	Regular	40.945	38.4	2.5
6	Kata	61.971	53.5	8.5
8	Regular	39.944	37.7	2.2
8	Kata	117.794	79.8	38.0

Πίνακας 5.3: Ανάλυση επιβάρυνσης δικτύου (Περιορισμένοι πόροι)

Στον Πίνακα 5.2 παρατηρείται ότι ο χρόνος φόρτωσης στα regular containers έχει τιμές 0.0002-0.0003 s ενώ τα kata containers παρουσιάζουν μεγαλύτερο εύρος τιμών, 0.0004-0.0007 s. Παρόμοια, όσον αφορά το χρόνο επεξεργασίας, ενώ τα regular containers διατηρούν σταθερή τιμή γύρω στα 0.038 s ανεξάρτητα από τον ρυθμό αιτημάτων, τα Kata containers παρουσιάζουν προοδευτική αύξηση από 0.044 s στα 2 RPS έως 0.078 s στα 8 RPS. Το overhead στον πίνακα 5.3, φαίνεται να παραμένει σχετικό σταθερό για την περίπτωση των regular containers, ενώ στην περίπτωση των kata containers παρουσιάζει σταδιακή αύξηση η οποία κορυφώνεται στα 8 RPS.

Rate (RPS)	Runtime	CPU Cycles ($\times 10^7$)	Instructions ($\times 10^7$)	IPC	Context Switches ($\times 10^6$)	CPU Migrations ($\times 10^3$)	Cache Miss Rate
2	Regular	2570.7	5032.4	1.96	4.25	444.5	26.78%
2	Kata	2493.2	5054.2	2.03	4.32	464.1	26.10%
4	Regular	2608.0	5250.4	2.01	4.32	446.2	25.96%
4	Kata	2664.5	5224.9	1.96	4.42	473.4	26.88%
6	Regular	2703.4	5447.8	2.02	4.44	481.1	26.31%
6	Kata	2707.9	5439.1	2.01	4.48	478.2	26.94%
8	Regular	2695.7	5712.8	2.12	4.47	485.0	25.70%
8	Kata	2913.1	5701.4	1.96	4.61	498.6	27.34%

Πίνακας 5.4: Μετρικές απόδοσης σε επίπεδο συστήματος (Περιορισμένοι πόροι)

Από τον πίνακα των μετρικών απόδοσης σε επίπεδο συστήματος, διαπιστώνεται ότι οι κύκλοι επεξεργαστή έχουν παραπλήσιες τιμές και για τα δύο περιβάλλοντα.

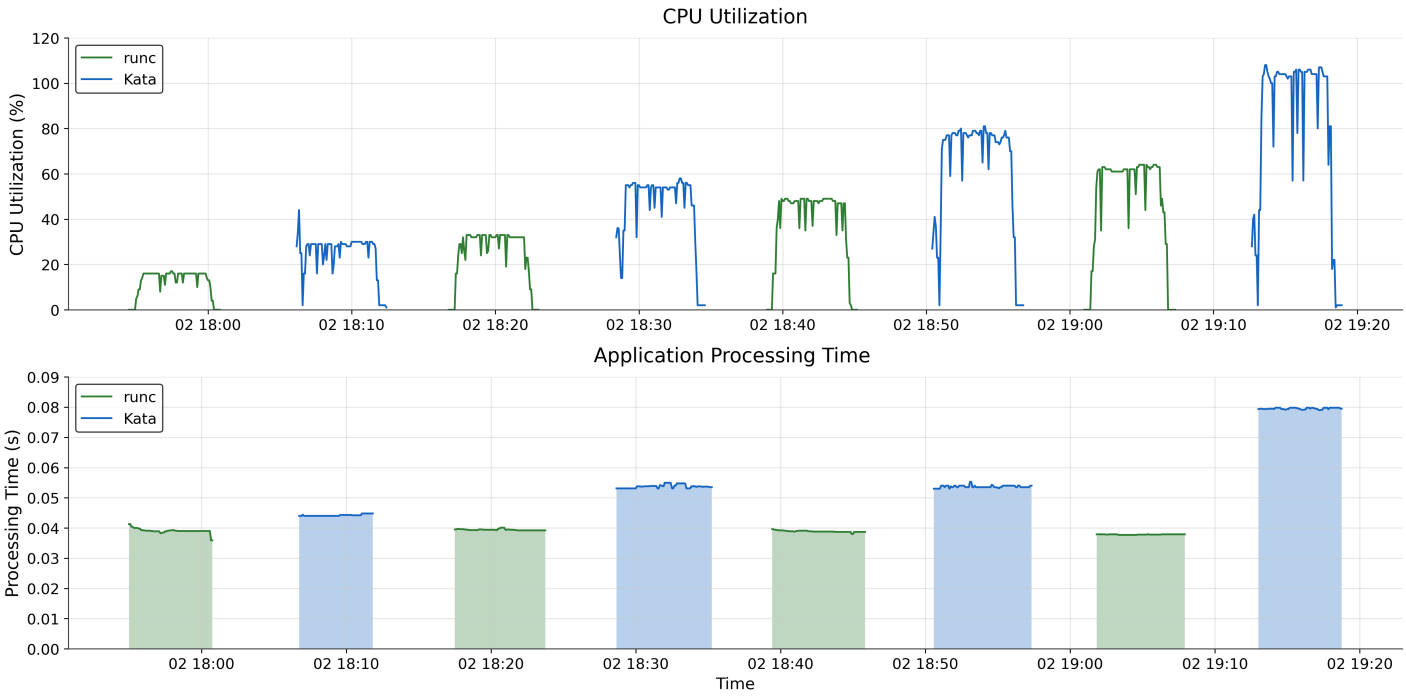
Οι τιμές context switches αυξάνονται σταδιακά και για τα δύο περιβάλλοντα καθώς αυξάνει το φορτίο, με τα Kata containers να παρουσιάζουν ελαφρώς υψηλότερες τιμές. Οι τιμές CPU Migrations παραμένουν υψηλές και για τα δύο περιβάλλοντα λόγω της απουσίας CPU pinning. Η ελαφρώς υψηλότερη συχνότητα μεταναστεύσεων στα Kata containers, υποδηλώνει μεγαλύτερη αστάθεια στην τοποθέτηση των διεργασιών, λόγω της πολυπλοκότερης αλληλεπίδρασης με τον scheduler του hypervisor.

Το ποσοστό αστοχίας κρυφής μνήμης παραμένει συγκρίσιμο μεταξύ των δύο περιβαλλόντων, με τα regular containers να εμφανίζουν τιμές από 25.70-26.78% και τα Kata containers 26.10-27.34%, με τα Kata containers να διατηρούν σταθερά ελαφρώς υψηλότερα ποσοστά αστοχίας, γεγονός που αντικατοπτρίζει την πολυπλοκότητα της διαχείρισης της ιεραρχίας μνήμης στο πλαίσιο της εικονικοποίησης.

Στο παρακάτω γράφημα χρήσης CPU φαίνεται ότι τα Kata containers καταναλώνουν σταθερά περισσότερους πόρους σε σύγκριση με τα regular containers για όλα τα επίπεδα φορτίου. Συγκεκριμένα, κατά τη διάρκεια των πειραμάτων με χαμηλό ρυθμό αιτημάτων (2 RPS), τα regular containers παρουσιάζουν χρήση CPU περίπου 15-20%, ενώ τα Kata containers φτάνουν το 30-35%. Αυτή η διαφορά γίνεται ακόμα πιο έντονη στα 8 RPS, όπου τα regular containers διατηρούν τη χρήση CPU στο 55-60%, ενώ τα Kata containers φτάνουν το 100% .Το γράφημα χρήσης CPU των Kata containers παρουσιάζει μεγαλύτερη μεταβλητότητα όπως φαίνεται από τις απότομες κορυφές στο γράφημα. Αυτό δείχνει λιγότερο προβλέψιμη και λιγότερο αποδοτική διαχείριση των πόρων, που επιβεβαιώνεται και από τις μετρικές context switches και CPU migrations που είναι υψηλότερες για τα Kata containers.

Η αυξημένη κατανάλωση CPU των Kata containers σχετίζεται άμεσα με τους υψηλότερους χρόνους επεξεργασίας που παρατηρούνται στο γράφημα Application

Processing Time.



Σχήμα 5.2: Χρήση CPU (%) και χρόνος επεξεργασίας εφαρμογής (s) για περιορισμένους πόρους

Μεσαίοι πόροι: 1000m CPU, 1Gi μνήμη

Rate (RPS)	Runtime	Mean Latency (ms)	Throughput (req/s)	Success Rate
4	Regular	41.173	4.00	100%
4	Kata	50.213	4.00	100%
8	Regular	40.399	8.00	100%
8	Kata	53.195	8.00	100%
12	Regular	40.616	12.00	100%
12	Kata	54.397	12.00	100%
16	Regular	40.640	16.00	100%
16	Kata	67.389	16.00	100%

Πίνακας 5.5: Μετρικές απόδοσης για διαφορετικούς ρυθμούς αιτημάτων (Μεσαίοι πόροι)

Η αύξηση των διαθέσιμων πόρων επιτρέπει τη δοκιμή υψηλότερων ρυθμών αιτημάτων (4-16 RPS) σε σύγκριση με τις συνθήκες περιορισμένων πόρων. Τα regular containers διατηρούν σταθερότητα με μέση καθυστέρηση στο εύρος 40.4-41.2 ms. Τα Kata containers εξακολουθούν να παρουσιάζουν αυξημένη καθυστέρηση που κυμαίνεται από 50.2 ms στα 4 RPS έως 67.4 ms στα 16 RPS.

Rate (RPS)	Runtime	Avg Load Time (s)	Avg Processing Time (s)	Avg Total Time (s)
4	Regular	0.0002	0.0382	0.0388
4	Kata	0.0004	0.0450	0.0455
8	Regular	0.0003	0.0374	0.0379
8	Kata	0.0005	0.0473	0.0480
12	Regular	0.0003	0.0377	0.0381
12	Kata	0.0005	0.0484	0.0491
16	Regular	0.0003	0.0378	0.0382
16	Kata	0.0005	0.0541	0.0549

Πίνακας 5.6: Ανάλυση απόδοσης σε επίπεδο εφαρμογής (Μεσαίοι πόροι)

Rate (RPS)	Container	Mean Latency (ms)	Total Processing (ms)	Overhead (ms)
4	Regular	41.173	38.8	2.4
4	Kata	50.213	45.5	4.7
8	Regular	40.399	37.9	2.5
8	Kata	53.195	48.0	5.2
12	Regular	40.616	38.1	2.5
12	Kata	54.397	49.1	5.3
16	Regular	40.640	38.2	2.4
16	Kata	67.389	54.9	12.5

Πίνακας 5.7: Ανάλυση επιβάρυνσης δικτύου (Μεσαίοι πόροι)

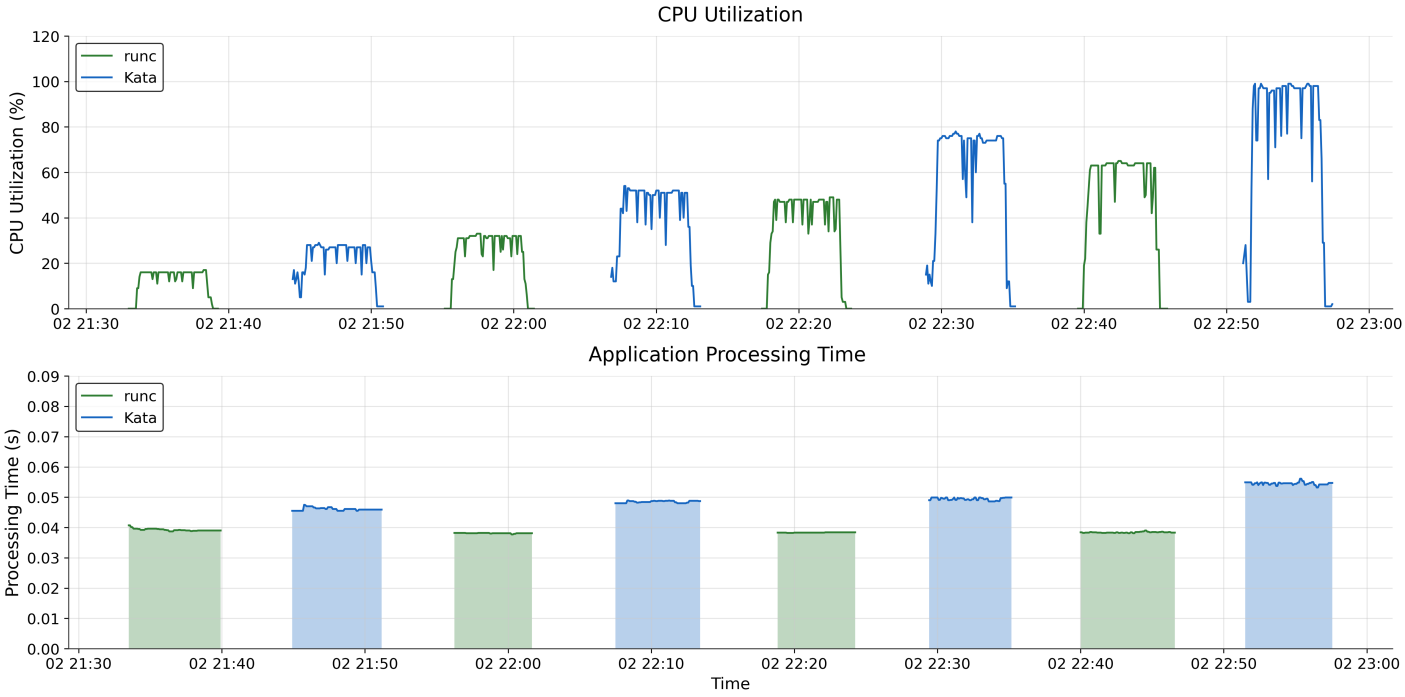
Η διαθεσιμότητα ενός πυρήνα CPU επιτρέπει στα regular containers να διατηρούν σταθερούς χρόνους επεξεργασίας (0.037-0.038 s) σε όλα τα επίπεδα φορτίου. Τα Kata containers, παρόλο που βελτιώνονται σε σχέση με τη περίπτωση περιορισμένων πόρων, εξακολουθούν να παρουσιάζουν αυξημένους χρόνους (0.045-0.054 s)

Rate (RPS)	Runtime	CPU Cycles ($\times 10^7$)	Instructions ($\times 10^7$)	IPC	Context Switches ($\times 10^6$)	CPU Migrations ($\times 10^3$)	Cache Miss Rate
4	Regular	2618.4	5244.9	2.00	4.32	458.5	26.38%
4	Kata	2680.2	5226.2	1.95	4.42	475.9	27.37%
8	Regular	2819.7	5703.9	2.02	4.51	490.1	26.67%
8	Kata	2838.2	5606.9	1.98	4.60	507.9	27.44%
12	Regular	2967.8	6023.1	2.03	4.62	499.7	26.64%
12	Kata	3021.6	6001.9	1.99	4.73	508.9	28.20%
16	Regular	3149.0	6396.9	2.03	4.79	527.9	26.95%
16	Kata	3223.5	6376.6	1.98	4.86	519.4	28.73%

Πίνακας 5.8: Μετρικές απόδοσης σε επίπεδο συστήματος (Μεσαίοι πόροι)

Στην περίπτωση με μεσαίους πόρους, η ανάλυση της χρήσης CPU, στο σχήμα 5.3, παρουσιάζει παρόμοια συμπεριφορά με τη περίπτωση περιορισμένων πόρων. Στα χαμηλά επίπεδα φορτίου (4 RPS), τα regular containers διατηρούν τη χρήση της CPU στο 15-20%, ενώ τα Kata containers φτάνουν το 25-30%. Καθώς το

φορτίο αυξάνεται στα 8 RPS, τα regular containers ανεβαίνουν στο 30-35%, ενώ τα Kata containers φτάνουν το 50-55%. Στα 12 RPS, παρατηρείται χρήση CPU περίπου 45-50% για τα regular containers και 75-80% για τα Kata containers. Τέλος, στα 16 RPS, τα regular containers φτάνουν το 60%, ενώ τα Kata containers πλησιάζουν το 100%.



Σχήμα 5.3: Χρήση CPU (%) και χρόνος επεξεργασίας εφαρμογής (s) για μέτριους πόρους

Αυξημένοι πόροι: 1500m CPU, 1536Mi μνήμη

Rate (RPS)	Runtime	Mean Latency (ms)	Throughput (req/s)	Success Rate
10	Regular	40.071	10.00	100%
10	Kata	54.107	10.00	100%
15	Regular	40.531	15.00	100%
15	Kata	53.727	15.00	100%
20	Regular	40.261	20.00	100%
20	Kata	60.130	20.00	100%
25	Regular	39.886	25.00	100%
25	Kata	249.312	25.00	100%

Πίνακας 5.9: Μετρικές απόδοσης για διαφορετικούς ρυθμούς αιτημάτων (Αυξημένοι πόροι)

Στον πίνακα 5.9, φαίνεται ότι τα regular containers αξιοποιούν αποδοτικά τους επιπλέον διαθέσιμους πόρους χωρίς να εμφανίζουν bottlenecks. Τα Kata containers παρουσιάζουν αύξηση της καθυστέρησης από 54.1 ms στα 10 RPS έως 60.1 ms στα 20 RPS, ωστόσο, στα 25 RPS παρατηρείται ότι η καθυστέρηση αυξάνεται απότομα φτάνοντας στα 249.3 ms.

Rate (RPS)	Runtime	Avg Load Time (s)	Avg Processing Time (s)	Avg Total Time (s)
10	Regular	0.0002	0.0374	0.0378
10	Kata	0.0005	0.0482	0.0489
15	Regular	0.0002	0.0378	0.0382
15	Kata	0.0004	0.0481	0.0487
20	Regular	0.0002	0.0375	0.0379
20	Kata	0.0005	0.0520	0.0528
25	Regular	0.0002	0.0373	0.0376
25	Kata	0.0006	0.0846	0.0856

Πίνακας 5.10: Ανάλυση απόδοσης σε επίπεδο εφαρμογής (Αυξημένοι πόροι)

Rate (RPS)	Container	Mean Latency (ms)	Total Processing (ms)	Overhead (ms)
10	Regular	40.071	37.8	2.3
10	Kata	54.107	48.9	5.2
15	Regular	40.531	38.2	2.3
15	Kata	53.727	48.7	5.0
20	Regular	40.261	37.9	2.4
20	Kata	60.130	52.8	7.3
25	Regular	39.886	37.6	2.3
25	Kata	249.312	85.6	163.7

Πίνακας 5.11: Ανάλυση επιβάρυνσης δικτύου (Αυξημένοι πόροι)

Στον Πίνακα 5.10, παρατηρείται ότι οι χρόνοι επεξεργασίας σε επίπεδο εφαρμογής για τα regular containers διατηρούν εξαιρετική σταθερότητα με χρόνο φόρτωσης εικόνας σταθερό στα 0.0002 s και ο χρόνος επεξεργασίας παραμένει επίσης σταθερός στα 0.0373-0.0378 s. Τα Kata containers παρουσιάζουν αυξημένους χρόνους σε όλες τις μετρικές. Ο χρόνος φόρτωσης κυμαίνεται από 0.0004 s έως 0.0006 s, περίπου τριπλάσιος από τα regular containers και ο χρόνος επεξεργασίας αυξάνεται από 0.0482 s στα 10 RPS έως 0.0846 s στα 25 RPS.

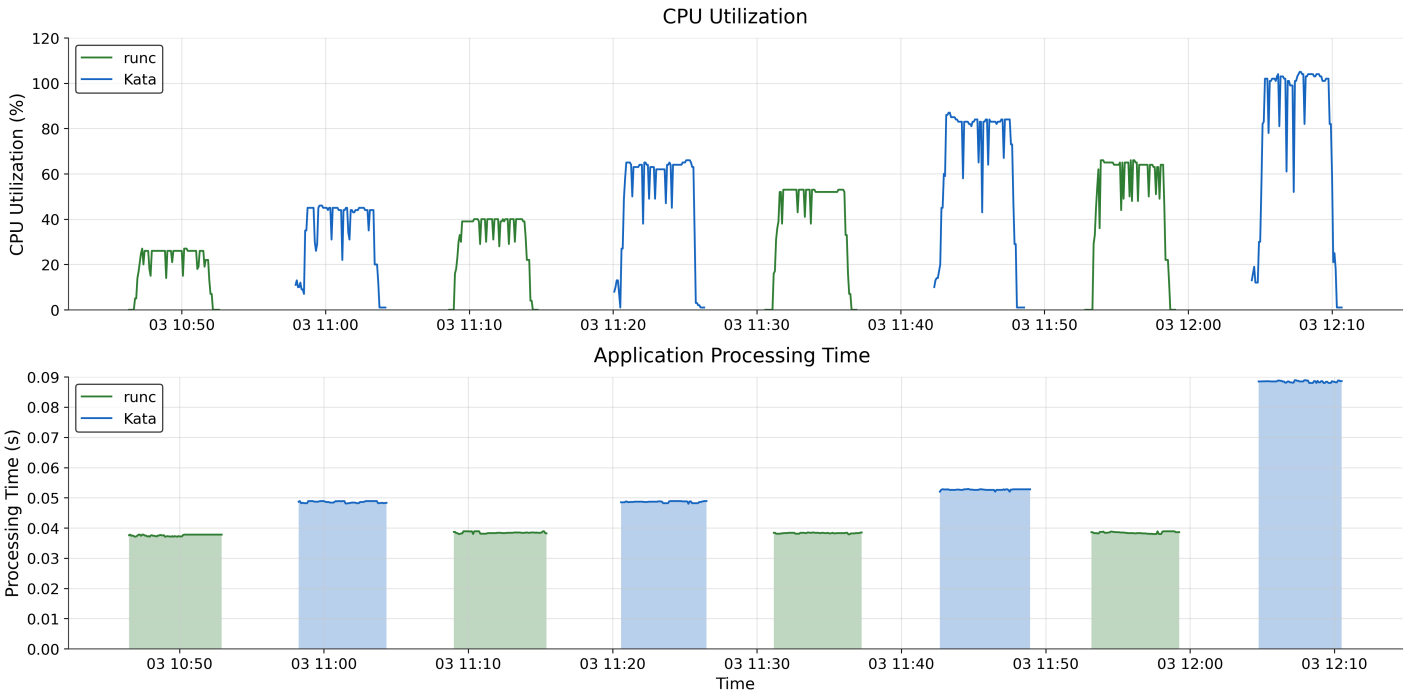
Ο Πίνακας 5.11 δείχνει ότι για τα regular containers, το overhead παραμένει εξαιρετικά χαμηλό και σταθερό στο εύρος 2.3-2.4 ms σε όλους τους ρυθμούς αιτημάτων ενώ για τα Kata containers αυξάνεται από 5.2 ms στα 10 RPS έως 163.7 ms στα 25 RPS. Αυτή η σημαντική αύξηση του overhead αποτελεί το 66% της συνολικής μέσης καθυστέρησης (249.3 ms), υποδεικνύοντας ότι το κύριο bottleneck στα 25 RPS δεν είναι η επεξεργασία της εφαρμογής (85.6 ms) αλλά η καθυστέρηση που εισάγεται από το στρώμα εικονικοποίησης και το δίκτυο.

Rate (RPS)	Runtime	CPU Cycles ($\times 10^7$)	Instructions ($\times 10^7$)	IPC	Context Switches ($\times 10^6$)	CPU Migrations ($\times 10^3$)	Cache Miss Rate
10	Regular	2832.7	5793.5	2.05	4.62	510.0	26.12%
10	Kata	2930.3	5760.1	1.97	4.77	540.7	27.98%
15	Regular	3105.4	6248.4	2.01	4.80	529.9	26.79%
15	Kata	3127.0	6246.7	2.00	4.99	562.3	28.65%
20	Regular	3296.2	6737.4	2.04	5.00	563.8	27.13%
20	Kata	3399.2	6712.0	1.97	5.16	564.9	29.60%
25	Regular	3492.5	7219.5	2.07	5.13	583.0	27.54%
25	Kata	3588.4	7219.7	2.01	5.26	511.1	29.45%

Πίνακας 5.12: Μετρικές απόδοσης σε επίπεδο συστήματος (Αυξημένοι πόροι)

Το IPC, όπως φαίνεται στον πίνακα 5.12, παραμένει χαμηλότερο για τα Kata containers (1.97-2.01) σε σύγκριση με τα regular containers (2.04-2.07), φανερώνοντας ότι ακόμα και με περισσότερους διαθέσιμους πόρους, η αποδοτικότητα χρήσης της CPU είναι μειωμένη.

Στο σχήμα 5.4, παρατηρείται ότι στα 10 RPS, τα regular containers διατηρούν τη χρήση CPU στο 25-30%, ενώ τα Kata containers φτάνουν το 40-45%. Στα 15 RPS, η χρήση αυξάνεται στο 35-40% για τα regular containers και 60-65% για τα Kata containers. Στα 20 RPS, τα regular containers ανεβαίνουν στο 50-55%, ενώ τα Kata containers πλησιάζουν το 80-85%. Το κρίσιμο σημείο εμφανίζεται στα 25 RPS, όπου τα regular containers φτάνουν το 60-65% χρήση της CPU, ενώ τα Kata containers το 100%.



Σχήμα 5.4: Χρήση CPU (%) και χρόνος επεξεργασίας εφαρμογής (s) για αυξημένους πόρους

5.3.1 Ανάλυση αποτελεσμάτων

Τα Kata containers παρουσιάζουν μεταβλητή απόδοση που εξαρτάται σημαντικά από τη διάθεση πόρων. Στη περίπτωση περιορισμένων πόρων, το κρίσιμο σημείο εμφανίζεται στα 8 RPS, στη περίπτωση μέτριων πόρων στα 16 RPS, ενώ στη περίπτωση αυξημένων πόρων, το κρίσιμο σημείο μετατοπίζεται στα 25 RPS. Σε κάθε σενάριο όμως, η αύξηση των διαθέσιμων πόρων δεν εξαλείφει το πρόβλημα του overhead εικονικοποίησης, αλλά μετατοπίζει το κρίσιμο σημείο σε υψηλότερα επίπεδα φορτίου. Τα regular containers, αντίθετα, δεν παρουσιάζουν τέτοια κρίσιμα σημεία και κλιμακώνονται γραμμικά, διατηρώντας σταθερή απόδοση ανεξάρτητα από τον ρυθμό αιτημάτων ή τους διαθέσιμους πόρους.

Στα Kata containers με περιορισμένους πόρους, η μέση καθυστέρηση κυμαίνεται από 48.8 ms στα 2 RPS έως 117.8 ms στα 8 RPS. Με μέτριους πόρους, η μέση καθυστέρηση είναι 50.2-67.4 ms για ρυθμούς έως 16 RPS. Με αυξημένους πόρους, η καθυστέρηση φτάνει τα 249.3 ms. Τα regular containers αντίθετα, χαρακτηρίζονται από σταθερή συμπεριφορά, διατηρώντας μέση καθυστέρηση 39.9-41.6 ms.

Η ανάλυση των χρόνων επεξεργασίας σε επίπεδο εφαρμογής αποκαλύπτει ότι ο χρόνος φόρτωσης εικόνας είναι σταθερά υψηλότερος για τα Kata containers (0.0004-0.0007 s) σε σύγκριση με τα regular containers (0.0002-0.0003 s) και ο χρόνος επεξεργασίας είναι αυξημένος.

Όσον αφορά το overhead που εισάγεται από το στρώμα εικονικοποίησης, παρουσιάζει σταθερά μοτίβα σε όλα τα σενάρια. Για τα regular containers, η επιβάρυνση παραμένει σταθερά χαμηλή στο εύρος 2.2-2.6 ms ανεξάρτητα από τους διαθέσιμους πόρους ή τον ρυθμό αιτημάτων. Για τα Kata containers, κυμαίνεται από 4.4 ms στα χαμηλά φορτία έως 163.7 ms στα 25 RPS με αυξημένους πόρους, υποδεικνύοντας μη γραμμική αύξηση καθώς το σύστημα πλησιάζει τα όρια της χωρητικότητάς του. Το overhead των Kata containers αποδίδεται στην I/O επιβάρυνση που εισάγει η virtio-block αρχιτεκτονική των Kata containers όπου αναφέρθηκε και στο προηγούμενο κεφάλαιο. Κάθε λειτουργία disk I/O πρέπει να διέλθει μέσω του virtio front-end driver στο VM, του Firecracker VMM, και τελικά του host OS filesystem, δημιουργώντας σημαντικό overhead σε σύγκριση με την άμεση πρόσβαση στο overlay2 filesystem των regular containers. Η επιβάρυνση αυτή γίνεται ιδιαίτερα εμφανής υπό υψηλό φορτίο, όπου η συσσώρευση των VM operations δημιουργεί bottleneck στο σύστημα.

Στις μετρικές επιπέδου συστήματος, το cache miss rate είναι σταθερά υψηλότερο για τα Kata containers, 27-29% σε όλες τις περιπτώσεις, σε σύγκριση με το 25-27% των regular containers. Αυτό το αυξημένο cache miss rate οδηγεί σε περισσότερες προσβάσεις στην κύρια μνήμη, επιβραδύνοντας έτσι την εκτέλεση. Οι τιμές των CPU migrations είναι επίσης υψηλότερες για τα Kata containers, ιδιαίτερα υπό υψηλό φορτίο.

5.4 Πείραμα Β: Απόδοση Εφαρμογών σε συνθήκες Παρεμβολών

CPU Pinning: 2 CPU, 2Gi μνήμη με παρεμβολή iBench (1 CPU)

Στη περίπτωση αυτή, τόσο τα regular όσο και τα Kata containers έχουν αποκλειστική πρόσβαση σε 2 πυρήνες CPU μέσω CPU pinning, ενώ παράλληλα εκτελείται ένα pod παρεμβολής με το iBench benchmark που καταναλώνει 1 πυρήνα CPU.

Rate (RPS)	Runtime	Mean Latency (ms)	Throughput (req/s)	Success Rate
15	Regular	38.095	15.00	100%
15	Kata	50.611	15.00	100%
20	Regular	37.671	20.00	100%
20	Kata	52.956	20.00	100%
25	Regular	38.479	25.00	100%
25	Kata	56.216	25.00	100%
30	Regular	38.862	30.00	100%
30	Kata	57.925	30.00	100%

Πίνακας 5.13: Μετρικές απόδοσης για διαφορετικούς ρυθμούς αιτημάτων με παρεμβολή 1 CPU

Στον Πίνακα 5.13, παρατηρείται ότι και τα δύο είδη container εμφανίζουν 100% ποσοστό επιτυχίας στα αιτήματα που τους στάλθηκαν. Τα regular containers διατηρούν σταθερότητα με μέση καθυστέρηση 37.7-38.9 ms για ρυθμούς από 15 έως 30 RPS. Η απόκλειστική πρόσβαση σε 2 πυρήνες CPU μέσω pinning επιτρέπει προβλέψιμη απόδοση ακόμα και παρουσία του pod παρεμβολής. Τα Kata containers από την άλλη, παρουσιάζουν αύξηση της καθυστέρησης από 50.6 ms στα 15 RPS έως 57.9 ms στα 30 RPS.

Rate (RPS)	Runtime	Avg Load Time (s)	Avg Processing Time (s)	Avg Total Time (s)
15	Regular	0.0002	0.0362	0.0365
15	Kata	0.0004	0.0456	0.0462
20	Regular	0.0002	0.0359	0.0362
20	Kata	0.0003	0.0485	0.0488
25	Regular	0.0002	0.0365	0.0368
25	Kata	0.0004	0.0504	0.0509
30	Regular	0.0002	0.0369	0.0373
30	Kata	0.0003	0.0514	0.0519

Πίνακας 5.14: Ανάλυση απόδοσης σε επίπεδο εφαρμογής με παρεμβολή 1 CPU

Rate (RPS)	Container	Mean Latency (ms)	Total Processing (ms)	Overhead (ms)
15	Regular	38.095	36.5	1.6
15	Kata	50.611	46.2	4.4
20	Regular	37.671	36.2	1.5
20	Kata	52.956	48.8	4.1
25	Regular	38.479	36.8	1.7
25	Kata	56.216	50.9	5.3
30	Regular	38.862	37.3	1.6
30	Kata	57.925	51.9	6.0

Πίνακας 5.15: Ανάλυση επιβάρυνσης δικτύου με παρεμβολή 1 CPU

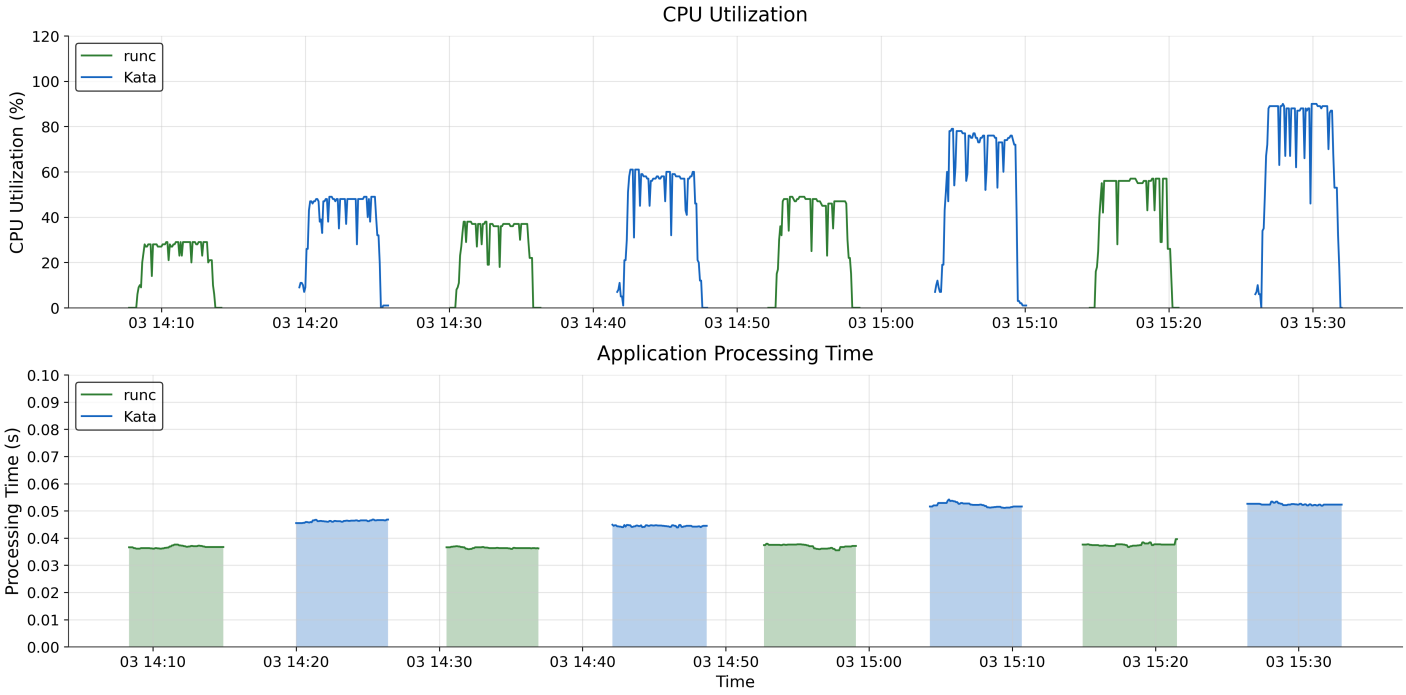
Ο Πίνακας 5.14 φανερώνει ότι ο χρόνος φόρτωσης εικόνας παραμένει σταθερός για τα regular containers (0.0002 s) αλλά υψηλότερος για τα Kata containers (0.0003-0.0004 s). Ο συνολικός χρόνος επεξεργασίας για τα regular containers κυμαίνεται στα 0.0362-0.0373 s, ενώ για τα Kata containers από 0.0462 s έως 0.0519 s στα 30 RPS.
Από τον Πίνακα 5.15 προκύπτει ότι η επιβάρυνση δικτύου για τα regular containers παραμένει εξαιρετικά χαμηλή και σταθερή (1.5-1.7 ms). Τα Kata containers παρουσιάζουν επιβάρυνση 4.1-6.0 ms, περίπου 3 φορές υψηλότερη. Το CPU pinning φαίνεται να περιορίζει την επιβάρυνση σε σχέση με τα προηγούμενα σενάρια χωρίς pinning.

Rate (RPS)	Runtime	CPU Cycles ($\times 10^7$)	Instructions ($\times 10^7$)	IPC	Context Switches ($\times 10^6$)	CPU Migrations ($\times 10^3$)	Cache Miss Rate
15	Regular	2835.7	4683.7	1.65	4.46	483.0	29.46%
15	Kata	3082.4	4941.2	1.60	4.64	456.0	31.34%
20	Regular	3241.7	5654.2	1.74	4.60	460.0	29.16%
20	Kata	3319.0	5872.3	1.77	4.79	424.6	31.51%
25	Regular	3627.3	6815.1	1.81	4.73	437.4	29.31%
25	Kata	3905.1	6801.3	1.74	4.99	361.6	32.01%
30	Regular	4069.1	7509.4	1.85	4.87	408.4	28.81%
30	Kata	4327.9	7716.1	1.78	5.18	307.5	31.53%

Πίνακας 5.16: Μετρικές απόδοσης σε επίπεδο συστήματος με παρεμβολή 1 CPU

Ο Πίνακας 5.16 αποκαλύπτει ότι το CPU pinning βελτιώνει την αποδοτικότητα και για τα δύο runtimes. Το IPC για τα regular containers κυμαίνεται στο 1.65-1.85, ενώ για τα Kata containers στο 1.60-1.78, δείχνοντας μικρότερη διαφορά σε σχέση με σενάρια χωρίς pinning. Οι τιμές των cache miss rate παραμένει υψηλότερο για τα Kata containers (31.3-32.0%) έναντι των regular containers (29.2-29.5%). Οι τιμές CPU migrations είναι σημαντικά μειωμένες λόγω του pinning, με τα regular containers να παρουσιάζουν 408-483 migrations ($\times 10^3$) και τα Kata containers 307-456 migrations ($\times 10^6$). Η μείωση των migrations συμβάλλει στη βελτιωμένη αποδοτικότητα cache.

Τα γραφήματα CPU Utilization δείχνουν ότι τα regular containers διατηρούν τη χρήση CPU στο 25-50% ανάλογα με τον ρυθμό αιτημάτων, ενώ τα Kata containers φτάνουν το 45-85% με πολύ πιο σταδιακή αύξηση συγκριτικά με τις περιπτώσεις του πρώτου πειράματος χωρίς το CPU pinning.



Σχήμα 5.5: Χρήση CPU (%) και χρόνος επεξεργασίας εφαρμογής (s) για παρεμβολή 1 CPU

CPU Pinning: 2 CPU, 2Gi μνήμη με παρεμβολή iBench (1.5 CPU)

Rate (RPS)	Runtime	Mean Latency (ms)	Throughput (req/s)	Success Rate
15	Regular	38.754	15.00	100%
15	Kata	50.932	15.00	100%
20	Regular	38.176	20.00	100%
20	Kata	51.839	20.00	100%
25	Regular	38.505	25.00	100%
25	Kata	58.041	25.00	100%
30	Regular	39.571	30.00	100%
30	Kata	1510.000	30.00	100%

Πίνακας 5.17: Μετρικές απόδοσης για διαφορετικούς ρυθμούς αιτημάτων με παρεμβολή 1.5 CPU

Από τον πίνακα 5.17 διαπιστώνεται ότι στα regular containers η μέση καθυστέρηση παραμένει σταθερή στα 38.2–39.6 ms, ενώ στα Kata containers εμφανίζεται αυξημένη, από 50.9-58 ms για 15–25 RPS. Στο φορτίο των 30 RPS, η καθυστέρηση στα Kata containers εκτοξεύεται στα 1510 ms, ενώ τα regular containers διατηρούν σταθερή απόδοση, δείχοντας την αδυναμία των Kata containers να πραγματοποιήσουν αποδοτική κλιμάκωση υπό υψηλό φόρτο.

Rate (RPS)	Runtime	Avg Load Time (s)	Avg Processing Time (s)	Avg Total Time (s)
15	Regular	0.0002	0.0367	0.0370
15	Kata	0.0003	0.0462	0.0466
20	Regular	0.0002	0.0362	0.0364
20	Kata	0.0003	0.0467	0.0471
25	Regular	0.0002	0.0363	0.0366
25	Kata	0.0003	0.0521	0.0526
30	Regular	0.0002	0.0374	0.0377
30	Kata	0.0005	0.0928	0.0937

Πίνακας 5.18: Ανάλυση απόδοσης σε επίπεδο εφαρμογής με παρεμβολή 1.5 CPU

Rate (RPS)	Container	Mean Latency (ms)	Total Processing (ms)	Overhead (ms)
15	Regular	38.754	36.7	2.1
15	Kata	50.932	46.2	4.7
20	Regular	38.176	36.2	2.0
20	Kata	51.8395	46.7	5.1
25	Regular	38.505	36.3	2.2
25	Kata	58.041	52.1	5.9
30	Regular	39.571	37.4	2.2
30	Kata	1510.000	92.8	1417.2

Πίνακας 5.19: Ανάλυση επιβάρυνσης δικτύου με παρεμβολή 1.5 CPU

Στον πίνακα 5.18, ο μέσος χρόνος φόρτωσης παραμένει αμελητέος σε όλα τα σενάρια, 0.0002-0,003 s, με τον χρόνο φότωσης των Kata containers στα 30 RPS να είναι λίγο μεγαλύτερος, 0,0005 s. Τα regular containers επιτυγχάνουν σταθερές τιμές για τον χρόνο επεξεργασίας, 36.2–37.4 ms σε όλα τα επίπεδα φορτίου, ενώ τα Kata containers εμφανίζουν προοδευτική αύξηση από 46.2 ms στα 15 RPS έως 52.1 ms στα 25 RPS. Στον ρυθμό των 30 RPS, ο χρόνος επεξεργασίας για τα Kata containers φτάνει στα 92.8 ms, έναντι 37.4 ms των regular containers, επιβεβαιώνοντας ότι η φάση επεξεργασίας αποτελεί το σημείο όπου εκδηλώνεται η υποβάθμιση απόδοσης υπό συνθήκες πίεσης.

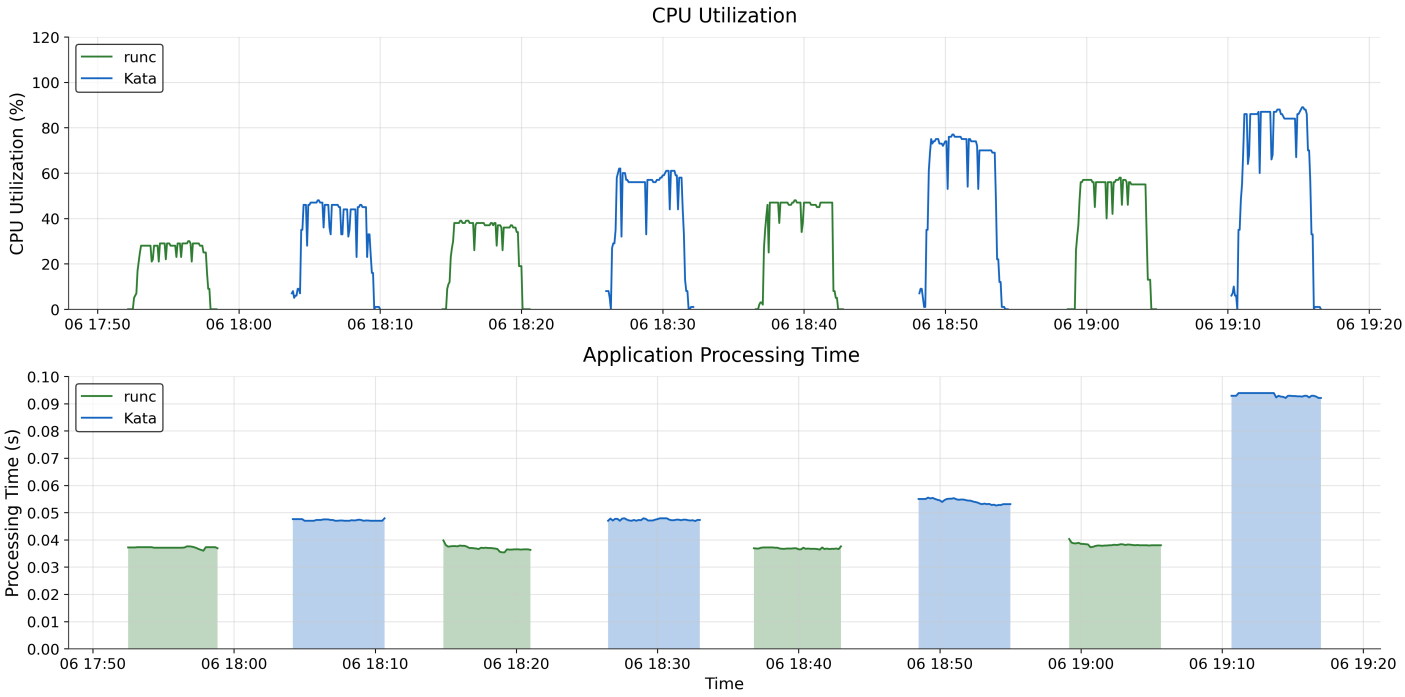
Στα Kata Containers, το overhead κυμαίνεται μεταξύ 4.7-5.9 ms, διπλάσιο σε σχέση με τα regular containers, 2.0–2.2 ms. Στην περίπτωση των 30 RPS, το overhead στα Kata containers εκτινάσσεται στα 1417.2 ms, αποτελώντας την κυρίαρχη αιτία της δραματικής αύξησης της συνολικής καθυστέρησης. Η απότομη αυτή αύξηση υποδηλώνει ότι η στοίβα δικτύου και οι μηχανισμοί επικοινωνίας μεταξύ guest και host συστημάτων φτάνουν σε σημείο κορεσμού.

Rate (RPS)	Runtime	CPU Cycles ($\times 10^7$)	Instructions ($\times 10^7$)	IPC	Context Switches ($\times 10^6$)	CPU Migrations ($\times 10^3$)	Cache Miss Rate
15	Regular	3928.1	6766.9	1.72	4.37	383.9	27.93%
15	Kata	4006.0	6783.1	1.69	4.63	379.6	29.92%
20	Regular	4096.3	7361.8	1.80	4.57	389.4	27.76%
20	Kata	4193.3	7364.9	1.76	4.85	371.0	29.99%
25	Regular	4303.7	7916.7	1.84	4.72	394.9	28.00%
25	Kata	4429.3	7892.3	1.78	5.01	346.0	30.46%
30	Regular	4579.0	8403.7	1.84	4.94	387.5	28.13%
30	Kata	4643.2	8403.8	1.81	5.21	312.6	30.26%

Πίνακας 5.20: Μετρικές απόδοσης σε επίπεδο συστήματος με παρεμβολή 1.5 CPU

Τα Kata containers απαιτούν περισσότερους κύκλους CPU σε όλες τις περιπτώσεις και εμφανίζουν μειωμένο IPC 1.69–1.81 έναντι στα regular containers όπου κυμαίνεται από 1.72 έως 1.84, υποδηλώνοντας μικρότερη αποδοτικότητα στην εκτέλεση εντολών ανά κύκλο. Επίσης, οι μετρήσεις δείχνουν ότι τα Kata containers εμφανίζουν σταθερά υψηλότερα ποσοστά cache misses 29.92% έως 30.46% έναντι 27.76–28.13% για τα regular containers, και αύξηση στον αριθμό των context switches, ειδικά υπό υψηλότερο φόρτο.

Στα γραφήματα χρήσης CPU φαίνεται ότι τα regular containers διατηρούν τη χρήση CPU στο 30-60%, ενώ τα Kata containers φτάνουν το 50-80%, ποσοστά λίγο αυξημένα συγκιτικά με τη περίπτωση όπου είχαμε παρεμβολή 1 πυρήνα CPU.



Σχήμα 5.6: Χρήση CPU (%) και χρόνος επεξεργασίας εφαρμογής (s) για παρεμβολή 1.5 CPU

5.4.1 Ανάλυση αποτελεσμάτων των πειραμάτων με βάση την παρεμβολή

Σύγκριση CPU Pinning: 2 CPU, 2Gi μνήμη με παρεμβολή iBench (1 CPU) με Σύγκριση CPU Pinning: 2 CPU, 2Gi μνήμη με παρεμβολή iBench (1.5 CPU)

Αναφορικά με τη μέση καθυστέρηση απόκρισης, παρατηρείται ότι για χαμηλά έως μέτρια επίπεδα φορτίου, η αύξηση της παρεμβολής από 1 σε 1.5 πυρήνες CPU έχει ελάχιστη επίδραση και στα δύο περιβάλλοντα. Το κρίσιμο σημείο διαφοροποίησης εμφανίζεται στον ρυθμό των 30 RPS, όπου τα regular containers διατηρούν σχεδόν ίδια απόδοση με μέση καθυστέρηση σε αντίθεση με τα Kata containers που παρουσιάζουν απότομη αύξηση στη καθυστέρηση φτάνοντας το σύστημα Kata containers πέραν του σημείου κορεσμού του.

Ο χρόνος φόρτωσης παραμένει αμελητέος και στα δύο σενάρια, υποδεικνύοντας ότι δεν αποτελεί παράγοντα που επηρεάζει τη διαφορά απόδοσης. Ο χρόνος επεξεργασίας για τα regular containers είναι σχεδόν ίδιος μεταξύ των δύο επιπέδων παρεμβολής σε όλους τους ρυθμούς φορτίου ενώ για τα Kata containers, ο χρόνος επεξεργασίας με παρεμβολή 1 πυρήνα CPU αυξάνεται γραμμικά. Με παρεμβολή 1.5 CPU, ο χρόνος επεξεργασίας ακολουθεί παρόμοια πορεία για τα χαμηλά και μέτρια φορτία, αλλά στα 30RPS εκτοξεύεται.

Το overhead δικτύου αποτελεί τον κρίσιμο παράγοντα που εξηγεί την κατάρρευση της απόδοσης. Για τα regular containers, το overhead παραμένει εξαιρετικά χαμηλό και σταθερό σε όλα τα σενάρια ενώ στα Kata containers το overhead αυξάνεται ελαφρώς. Ωστόσο, στα 30RPS με παρεμβολή 1.5 CPU, το overhead αυξάνεται απότομα. Αυτή η αύξηση κατά δύο τάξεις μεγέθους υποδηλώνει ότι η στοίβα δικτύου και οι μηχανισμοί επικοινωνίας μεταξύ guest και host συστημάτων αντιμετωπίζουν κορεσμό όταν οι διαθέσιμοι πόροι γίνονται ανεπαρκείς.

Οι τιμές για τα context switches παραμένουν συγκρίσιμες μεταξύ των δύο επιπέδων παρεμβολής και για τα δύο περιβάλλοντα, με τα regular containers να εμφανίζουν ελαφρώς χαμηλότερες τιμές με έναν πυρήνα παρεμβολής. Τα Kata containers εμφανίζουν παρόμοιο πρότυπο, με ελαφρώς χαμηλότερες τιμές με 1.5 πυρήνα CPU παρεμβολής σε σύγκριση με 1 πυρήνα CPU.

Όσον αφορά την χρήση CPU, με 1 CPU για παρεμβολή, τα regular containers διατηρούν τη χρήση CPU στο 25-50%, ενώ με 1.5 η χρήση αυξάνεται στο 30-60%, υποδηλώνοντας μεγαλύτερη πίεση στους διαθέσιμους πόρους. Τα Kata containers εμφανίζουν χρήση 40-80% με 1 CPU για παρεμβολή, η οποία αυξάνεται σε 50-80% με 1.5 CPU, πλησιάζοντας στα όρια της διαθέσιμης επεξεργαστικής ισχύος και εξηγώντας την κατάρρευση της απόδοσης στα υψηλά φορτία.

Συνοψίζοντας, η σύγκριση αποδεικνύει ότι τα regular containers παρουσιάζουν εξαιρετική ανθεκτικότητα στην αυξημένη παρεμβολή, διατηρώντας σχεδόν

ταυτόσημη απόδοση ανεξάρτητα από το επίπεδο παρεμβολής. Αντίθετα, τα Kata containers, παρά το γεγονός ότι διατηρούν αποδεκτή απόδοση για χαμηλά έως μέτρια φορτία ακόμα και με αυξημένη παρεμβολή, εμφανίζουν κρίσιμο σημείο αστοχίας όταν ο συνδυασμός υψηλού φορτίου και αυξημένης παρεμβολής ξεπερνάει τα όρια του συστήματος.

Μετρική	**Regular**		**Kata**		**ΜεταβολήKata**
	1 CPU	**1.5 CPU**	**1 CPU**	**1.5 CPU**	
Μέση Καθυστέρηση 15 RPS (ms)	38.1	38.8	50.6	50.9	+0.6%
Μέση Καθυστέρηση 30 RPS (ms)	38.9	39.6	57.9	1510.0	+2507%
Network Overhead 30 RPS (ms)	1.6	2.2	6.0	1417.2	+23520%
Χρόνος Επεξεργασίας 30 RPS (ms)	36.9	37.4	51.4	92.8	+81%
IPC 30 RPS	1.85	1.84	1.78	1.81	+1%
Cache Miss Rate 30 RPS (%)	28.8	28.1	31.5	30.3	-3.8%

Πίνακας 5.21: Συγκριτικά Αποτελέσματα Μέτρησης Απόδοσης για παρεμβολή 1 και 1.5 CPU

5.5 Πείραμα Γ: Απόδοση εφαρμογών σε Συνθήκες Παρεμβολών με Kata Container

CPU Pinning Kata Containers: 2 CPU, 2Gi μνήμη με παρεμβολή iBench (1 CPU)

Rate (RPS)	Runtime	Mean Latency (ms)	Throughput (req/s)	Success Rate
15	Regular	39.391	15.00	100%
15	Kata	50.097	15.00	100%
20	Regular	39.041	20.00	100%
20	Kata	52.748	20.00	100%
25	Regular	40.092	25.00	100%
25	Kata	55.318	25.00	100%
30	Regular	41.413	30.00	100%
30	Kata	57.233	30.00	100%

Πίνακας 5.22: Μετρικές απόδοσης για διαφορετικούς ρυθμούς αιτημάτων με παρεμβολή 1 CPU-kata pinned

Από τον πίνακα 5.21 διαπιστώνεται ότι η μέση καθυστέρηση στα regular containers παραμένει σχεδόν σταθερή στα 39.0–41.4 ms για όλα τα επίπεδα φορτίου, ενώ στα Kata containers εμφανίζεται αυξημένη, κυμαινόμενη από 50.1 έως 57.2 ms για ρυθμούς 15–30 RPS. Η ρυθμαπόδοση παραμένει ίση με τον στοχευόμενο ρυθμό αιτημάτων και για τα δύο περιβάλλοντα, ενώ το ποσοστό επιτυχίας διατηρείται στο 100% σε όλα τα σενάρια, επιβεβαιώνοντας την ικανότητα του συστήματος να εξυπηρετεί το φορτίο χωρίς απώλειες.

Rate (RPS)	Runtime	Avg Load Time (s)	Avg Processing Time (s)	Avg Total Time (s)
15	Regular	0.0002	0.0367	0.0371
15	Kata	0.0003	0.0454	0.0460
20	Regular	0.0002	0.0366	0.0369
20	Kata	0.0004	0.0473	0.0478
25	Regular	0.0003	0.0373	0.0377
25	Kata	0.0004	0.0496	0.0502
30	Regular	0.0002	0.0386	0.0390
30	Kata	0.0003	0.0511	0.0516

Πίνακας 5.23: Ανάλυση απόδοσης σε επίπεδο εφαρμογής με παρεμβολή 1 CPU-kata pinned

Rate (RPS)	Container	Mean Latency (ms)	Total Processing (ms)	Overhead (ms)
15	Regular	39.391	37.1	2.3
15	Kata	50.097	45.7	4.4
20	Regular	39.041	36.9	2.1
20	Kata	52.748	47.8	4.9
25	Regular	40.092	37.7	2.4
25	Kata	55.318	50.2	5.1
30	Regular	41.413	39.0	2.4
30	Kata	57.233	51.6	5.6

Πίνακας 5.24: Ανάλυση επιβάρυνσης δικτύου με παρεμβολή 1 CPU-kata pinned

Ο χρόνος επεξεργασίας, όπως φαίνεται στον πίνακα 5.22, είναι αυξημένος στα Kata containers, από 0.037–0.039 s στα regular και φτάνει από 0.045–0.051 s στα Kata. Η διαφορά αυτή εντοπίζεται σε όλο το φάσμα των RPS

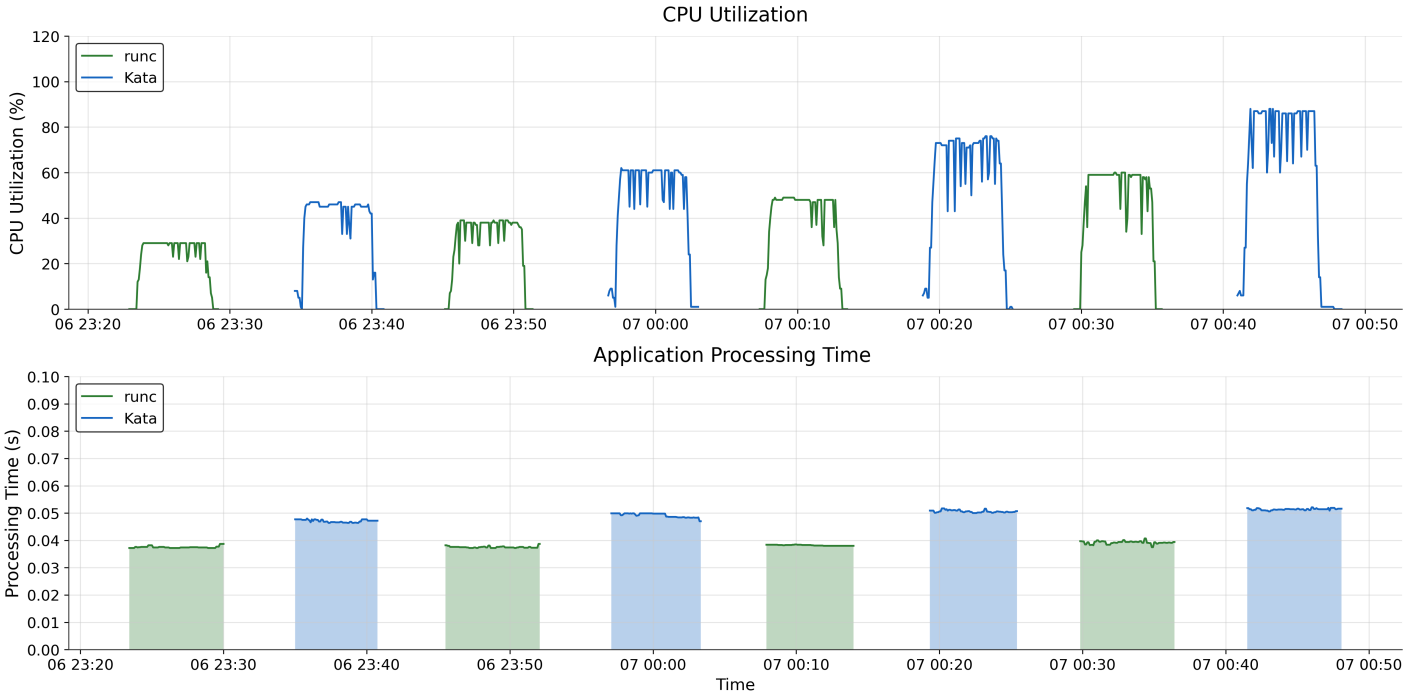
Στον πίνακα 5.23 φαίνεται ότι στα regular containers, η καθυστέρηση που αντιστοιχεί στο επιπλέον κόστος πέραν της καθαρής επεξεργασίας κυμαίνεται σταθερά στα 2.1–2.4 ms. Στα Kata containers το overhead είναι αυξημένο περίπου κατά 2–3 ms, φτάνοντας στα 5.6 ms στα 30 RPS. Αν και η διαφορά αυτή είναι αισθητή, παραμένει σταθερή και δεν παρουσιάζει εκρηκτική αύξηση.

Rate (RPS)	Runtime	CPU Cycles ($\times 10^7$)	Instructions ($\times 10^7$)	IPC	Context Switches ($\times 10^6$)	CPU Migrations ($\times 10^3$)	Cache Miss Rate
15	Regular	3972.4	7691.0	1.94	4.43	395.3	26.51%
15	Kata	3888.4	7437.4	1.91	4.61	328.7	28.82%
20	Regular	4200.5	8166.4	1.94	4.60	396.9	27.27%
20	Kata	4148.0	7969.6	1.92	4.83	306.5	28.94%
25	Regular	4407.2	8602.4	1.95	4.77	389.1	26.64%
25	Kata	4253.1	8439.4	1.98	4.98	290.3	29.14%
30	Regular	4621.1	9039.8	1.96	4.92	373.2	26.83%
30	Kata	4559.3	8763.2	1.92	5.24	264.8	29.56%

Πίνακας 5.25: Μετρικές απόδοσης σε επίπεδο συστήματος με παρεμβολή 1 CPU-kata pinned

Στα regular containers το IPC παραμένει γύρω στο 1.94–1.96, ενώ στα Kata containers κυμαίνεται από 1.91 έως 1.98, δείχνοντας ότι η αποδοτικότητα εκτέλεσης εντολών είναι συγκρίσιμη, με μικρές διακυμάνσεις. Οι τιμές των context switches είναι ελαφρώς αυξημένες στα Kata, ιδιαίτερα καθώς αυξάνεται το RPS, ενώ οι μετρήσεις CPU migrations είναι χαμηλότερες στα Kata containers, κάτι που μπορεί να σχετίζεται με την πιο αυστηρή δέσμευση CPU. Το ποσοστό cache miss παραμένει σταθερά υψηλότερο στα Kata (28–29% έναντι 26–27%)

Στο γράφημα χρήσης CPU φαίνεται ότι τα regular containers διατηρούν τη χρήση CPU στο 30-60%, σε υψηλότερες τιμές συγκριτικά με τη περίπτωση όπου είχαμε δεσμευμένη CPU, όπου οι τιμές ήταν 25-50%, γεγονός που φανερώνει τη πλεονέκτημα της δέσμευσης μνήμης σχετικά με την πιο αποδοτική κατανάλωση πόρων. Τα Kata containers κυμαίνονται από το 45-85%, όπως είχε παρατηρηθεί και στο προηγούμενο πείραμα. Η αυξηση της χρήσης CPU εξακολουθεί να είναι πιο σταδιακή καθώς αυξάνονται τα RPS στη περίπτωση των regular containers συγκρητικά με τα kata containers.



Σχήμα 5.7: Χρήση CPU (%) και χρόνος επεξεργασίας εφαρμογής (s) για παρεμβολή 1 CPU-kata pinned

CPU Pinning Kata Containers: 2 CPU, 2Gi μνήμη με παρεμβολή iBench (1.5 CPU)

Rate (RPS)	Runtime	Mean Latency (ms)	Throughput (req/s)	Success Rate
15	Regular	39.250	15.00	100%
15	Kata	50.120	15.00	100%
20	Regular	39.366	20.00	100%
20	Kata	50.744	20.00	100%
25	Regular	40.066	25.00	100%
25	Kata	58.415	25.00	100%
30	Regular	41.347	30.00	100%
30	Kata	62.471	30.00	100%

Πίνακας 5.26: Μετρικές απόδοσης για διαφορετικούς ρυθμούς αιτημάτων με παρεμβολή 1.5 CPU-kata pinned

Στον πίνακα 5.25, φαίνεται ότι ποσοστό επιτυχίας παραμένει σταθερά στο 100% και για τις δύο υλοποιήσεις η μέση καθυστέρηση στα Kata containers είναι συστηματικά αυξημένη σε σχέση με τα regular, φτάνοντας τα 62.471 ms έναντι 41.35 ms. Οι τιμές είναι ελαφρώς αυξημένες συγκριτικά με τη προηγούμενη περίπτωση όπου η παρεμβολή ήταν μικρότερη.

Rate (RPS)	Runtime	Avg Load Time (s)	Avg Processing Time (s)	Avg Total Time (s)
15	Regular	0.0002	0.0367	0.0371
15	Kata	0.0003	0.0453	0.0459
20	Regular	0.0002	0.0368	0.0371
20	Kata	0.0004	0.0457	0.0461
25	Regular	0.0003	0.0374	0.0377
25	Kata	0.0003	0.0524	0.0529
30	Regular	0.0002	0.0385	0.0388
30	Kata	0.0003	0.0541	0.0546

Πίνακας 5.27: Ανάλυση απόδοσης σε επίπεδο εφαρμογής με παρεμβολή 1.5 CPU-kata pinned

Rate (RPS)	Container	Mean Latency (ms)	Total Processing (ms)	Overhead (ms)
15	Regular	39.250	37.1	2.2
15	Kata	50.120	45.9	4.2
20	Regular	39.366	37.1	2.3
20	Kata	50.744	46.1	4.6
25	Regular	40.066	37.7	2.4
25	Kata	58.415	52.9	5.5
30	Regular	41.347	38.8	2.5
30	Kata	62.471	54.6	7.9

Πίνακας 5.28: Ανάλυση επιβάρυνσης δικτύου με παρεμβολή 1.5 CPU-kata pinned

Ο χρόνος φόρτωσης παραμένει πολύ χαμηλός και στα δύο περιβάλλοντα. Ωστόσο, ο χρόνος επεξεργασίας εμφανίζεται πάλι αυξημένος στα Kata containers, φτάνοντας τα 54.6 ms στα 30 RPS έναντι 38.8 ms στα regular. Η διαφορά είναι λιγότερο έντονη στα χαμηλά φορτία, για παράδειγμα περίπου 9 ms στα 15 RPS, αλλά αυξάνεται προοδευτικά με την αύξηση του RPS. Στα regular containers η

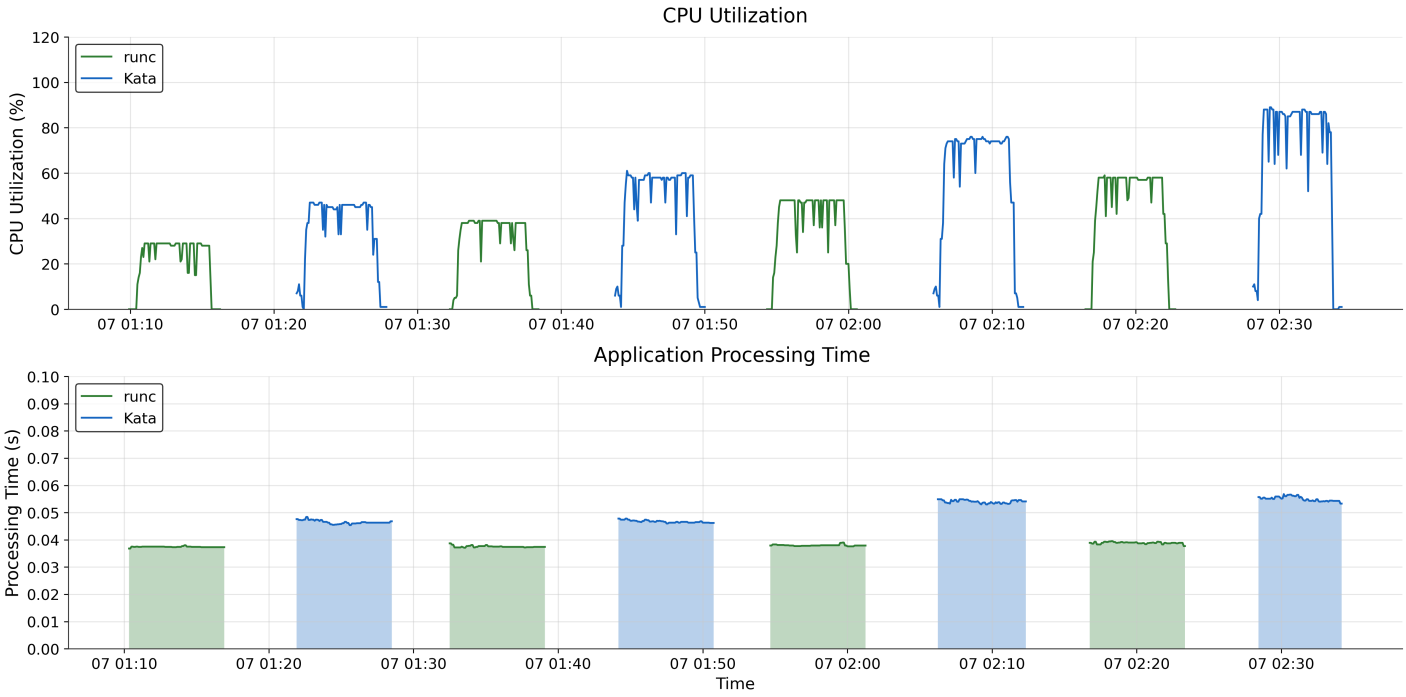
επιβάρυνση παραμένει σταθερή γύρω στα 2.2–2.5 ms, ενώ στα Kata containers αυξάνεται σημαντικά, φτάνοντας τα 7.9 ms στα 30 RPS.

Rate (RPS)	Runtime	CPU Cycles ($\times 10^7$)	Instructions ($\times 10^7$)	IPC	Context Switches ($\times 10^6$)	CPU Migrations ($\times 10^3$)	Cache Miss Rate
15	Regular	3892.0	7575.2	1.95	4.26	365.4	26.18%
15	Kata	3861.0	7361.6	1.91	4.49	301.7	28.35%
20	Regular	4170.3	8095.0	1.94	4.44	355.8	26.83%
20	Kata	4054.6	7849.6	1.94	4.73	282.3	28.93%
25	Regular	4729.0	8881.8	1.88	4.62	324.7	27.38%
25	Kata	4708.2	8687.9	1.85	4.93	250.8	28.68%
30	Regular	5013.6	9539.6	1.90	4.73	316.4	26.85%
30	Kata	4920.3	9098.6	1.85	5.16	237.2	29.83%

Πίνακας 5.29: Μετρικές απόδοσης σε επίπεδο συστήματος με παρεμβολή 1.5 CPU-kata pinned

Τα Kata containers εμφανίζουν χαμηλότερο IPC (1.85–1.94 έναντι 1.88–1.95), υψηλότερη τιμή context switches και χαμηλότερη τιμή CPU migrations.Το ποσοστό cache misses είναι συστηματικά υψηλότερο στα Kata 28–29.8% συγκριτικά με 26–27% για τα regular, στοιχείο που φανερώνει μεγαλύτερη επιβάρυνση στη διαχείριση μνήμης.

Στο γράφημα χρήσης CPU φαίνεται ότι τα regular containers διατηρούν τη χρήση CPU στο 30-60%, ενώ τα Kata containers κυμαίνονται απ 45-85%.



Σχήμα 5.8: Χρήση CPU (%) και χρόνος επεξεργασίας εφαρμογής (s) για παρεμβολή 1.5 CPU-kata pinned

5.5.1 Ανάλυση αποτελεσμάτων των πειραμάτων με βάση τη δέσμευση μνήμης

Σύγκριση CPU Pinning: 2 CPU, 2Gi μνήμη με παρεμβολή iBench (1 CPU) με CPU Pinning Kata Containers: 2 CPU, 2Gi μνήμη με παρεμβολή iBench (1 CPU)

Στην περίπτωση όπου και τα δύο container είναι pinned, τα Regular containers παρουσιάζουν σταθερά χαμηλότερες καθυστερήσεις γύρω στα 38 ms, ενώ τα Kata κινούνται στην περιοχή των 50–58 ms. Στην περίπτωση όπου pinned είναι μόνο το Kata, τα Regular containers εμφανίζουν υψηλότερη καθυστέρηση, περίπου 39–41 ms, ενώ τα Kata διατηρούνται στα ίδια επίπεδα με πριν. Αυτό φανερώνει ότι η απουσία pinning στα Regular containers επιφέρει μια μικρή επιβάρυνση , ενώ η συμπεριφορά των Kata παραμένει αναλλοίωτη. Η απόκλειστη πρόσβαση στις CPU μέσω pinning επιτρέπει προβλέψιμη και σταθερή απόδοση ακόμα και παρουσία του pod παρεμβολής.

Όσον αφορά τις μετρικές σε επίπεδο εφαρμογής, όταν και τα δύο container είναι pinned, οι μετρικές για τα Regular containers εμφανίζουν εξαιρετικά χαμηλό και σταθερό χρόνο φόρτωσης (0.0002 s) και χρόνο επεξεργασίας που παραμένει σχεδόν αμετάβλητος από τα 15 έως τα 30 RPS (0.0362–0.0369 s). Τα Kata containers, αντίθετα, παρουσιάζουν αυξημένο χρόνο επεξεργασίας, ο οποίος κυμαίνεται από 0.0456 s στα 15 RPS έως 0.0514 s στα 30 RPS, ενώ ο συνολικός χρόνος ολοκλήρωσης αυξάνεται αναλόγως. Στην περίπτωση όπου pinned είναι μόνο το Kata, τα Regular containers εξακολουθούν να εμφανίζουν χαμηλούς χρόνους επεξεργασίας, αλλά αυτοί παρουσιάζουν μικρή αύξηση στα υψηλότερα RPS (από 0.0367 s στα 15 RPS σε 0.0386 s στα 30 RPS), σε αντίθεση με την πλήρη σταθερότητα που παρατηρείται όταν και τα δύο container είναι pinned. Τα Kata containers, αν και παραμένουν πιο αργά σε σχέση με τα Regular, εμφανίζουν παρόμοιες μετρικές με την προηγούμενη περίπτωση. Συμπεραίνεται λοιπόν ότι η ύπαρξη pinning και για τα Regular containers συμβάλλει στη διατήρηση σταθερότερων επιπέδων χρόνου επεξεργασίας.

Το overhead για τα regular containers παραμένει εξαιρετικά χαμηλό στα 1.5–1.7 ms, ακόμα χαμηλότερο από το σενάριο με pinning μόνο στα Kata containers όπου κυμαινόταν στα 2.1–2.4 ms, υποδεικνύοντας ότι το pinning και στα δύο περιβάλλοντα μειώνει την επιβάρυνση.

Για τις μετρικές συστήματος, διαπιστώνεται ότι οι κύκλοι CPU για τα regular containers είναι σημαντικά λιγότεροι έναντι των Kata containers. Αυτή η διαφορά είναι μεγαλύτερη από το σενάριο με pinning μόνο στα Kata, όπου οι κύκλοι ήταν συγκρίσιμοι για τα Kata containers.

Στην περίπτωση όπου και τα δυο container είναι pinned, οι τιμές των CPU Migrations για τα regular containers είναι σημαντικά υψηλότερες από το σενάριο

με pinning μόνο στα Kata. Η παρατηρούμενη αύξηση φαίνεται να συνδέεται με τον περιορισμό που επιβάλλει το pinning για το προγραμματισμό και την κατανομή των πόρων. Όταν και τα δύο containers δεσμεύουν συγκεκριμένους πυρήνες, υπάρχει μικρότερη ευελιξία στην κατανομή του φόρτου, με αποτέλεσμα να αυξάνονται οι μετακινήσεις μεταξύ των διαθέσιμων πυρήνων. Αντίθετα, στο σενάριο όπου μόνο το Kata είναι pinned, τα Regular containers έχουν τη δυνατότητα να χρησιμοποιούν μεγαλύτερο εύρος πυρήνων, γεγονός που περιορίζει την ανάγκη για συχνές μετακινήσεις και οδηγεί σε χαμηλότερες τιμές CPU migrations.

Μετρική	**Regular (Both)**	**Kata (Both)**	**Regular (Kata Only)**	**Kata (Kata Only)**	**Διαφορά Regular**	**Διαφορά Kata**
Μέση Καθυστέρηση 15 RPS	38.1	50.6	39.4	50.1	+3.4%	-1.0%
Μέση Καθυστέρηση 30 RPS	38.9	57.9	41.4	57.2	+6.4%	-1.2%
Network Overhead 15 RPS	1.6	4.4	2.3	4.4	+43.8%	0.0%
Network Overhead 30 RPS	1.6	6.0	2.4	5.6	+50.0%	-6.7%
Χρόνος Επεξεργασίας 30 RPS	36.9	51.4	38.6	51.1	+4.6%	-0.6%
IPC 30 RPS	1.85	1.78	1.96	1.92	+5.9%	+7.9%
Cache Miss Rate 30 RPS (%)	28.8	31.5	26.8	29.6	-6.9%	-6.0%
CPU Migrations 30 RPS (K)	408.4	307.5	373.2	264.8	-8.6%	-13.9%
CPU Utilization 30 RPS (%)	25–50	40–80	30–60	45–80	+20%	+6.3%

Πίνακας 5.30: Συγκριτικός πίνακας μετρικών απόδοσης για CPU pinning με παρεμβολή 1 CPU

Επίλογος

6.1 Σύνοψη και συμπεράσματα

Τα αποτελέσματα των πειραμάτων αποδεικνύουν ότι τα Kata containers, παρά τα σημαντικά πλεονεκτήματα ασφαλείας που προσφέρουν μέσω της πλήρους απομόνωσης σε επίπεδο υλικού, παρουσιάζουν υποβάθμιση της απόδοσης με συστηματικά υψηλότερο overhead σε σύγκριση με τα regular containers. Αυτή η επιβάρυνση οφείλεται κυρίως στις συχνές μεταβάσεις λειτουργίας (mode transitions) μεταξύ guest και host συστήματος, στο επιπλέον overhead της διαχείρισης I/O μέσω virtio-block devices που εισάγει πρόσθετα επίπεδα εικονικοποίησης, και στην πολυπλοκότερη ιεραρχία μνήμης που παρουσιάζει αυξημένα ποσοστά cache misses.

Κρίσιμη παρατήρηση αποτελεί η συμπεριφορά των Kata containers υπό συνθήκες υψηλού φορτίου και έντονης παρεμβολής πόρων. Τα Kata containers αντιμετωπίζουν προβλήματα κλιμακωσιμότητας όταν οι διαθέσιμοι πόροι πλησιάζουν στο σημείο κορεσμού, καθιστώντας τα λιγότερο κατάλληλα για σενάρια όπου η μέγιστη απόδοση και η προβλεψιμότητα απόκρισης είναι σημαντικές απαιτήσεις.

Η εφαρμογή της τεχνικής CPU pinning αποδείχθηκε ότι βελτιώνει σημαντικά την απόδοση και για τα δύο περιβάλλοντα υπό συνθήκες παρεμβολής. Η δέσμευση CPU μειώνει τις μεταναστεύσεις μεταξύ CPUs και βελτιώνει την αποδοτικότητα της μνήμης cache, επιτρέποντας πιο προβλέψιμη και σταθερή απόδοση. Ωστόσο, ακόμα και με pinning, η επιβάρυνση της εικονικοποίησης παραμένει, με τα Kata containers να απαιτούν περισσότερο χρόνο απόκρισης σε σύγκριση με τα regular containers ειδικά κατά την επεξεργασία λειτουργιών I/O.

Η μελέτη επιβεβαιώνει ακόμη, ότι η επιλογή μεταξύ regular και Kata containers απαιτεί προσεκτική εξισορρόπηση των απαιτήσεων ασφαλείας και απόδοσης. Για εφαρμογές με χαμηλές απαιτήσεις ασφαλείας όπου η μέγιστη απόδοση είναι κρίσιμη, τα regular containers αποτελούν προτιμότερη επιλογή λόγω του ελάχιστου overhead και της προβλέψιμης κλιμάκωσης. Αντίθετα, για εφαρμογές που χειρίζονται ευαίσθητα δεδομένα ή εκτελούνται σε μη αξιόπιστα περιβάλλοντα όπου η ισχυρή απομόνωση είναι απαραίτητη, τα Kata containers προσφέρουν σημαντικά πλεονεκτήματα παρά την επιβάρυνση απόδοσης. Σε τέτοιες περιπτώσεις, η

χρήση τεχνικών όπως το CPU pinning και η προσεκτική διαχείριση των διαθέσιμων πόρων, ώστε να αποφεύγονται σημεία κορεσμού, μπορούν να μειώσουν την επιβάρυνση και να διασφαλίσουν αποδεκτή απόδοση. Στο πλαίσιο του edge computing συγκεκριμένα, όπου οι κόμβοι χαρακτηρίζονται από περιορισμένους υπολογιστικούς πόρους και την ανάγκη για χαμηλή καθυστέρηση, τα συμπεράσματα αυτής της εργασίας παρέχουν οδηγίες για την ανάπτυξη συστημάτων. Η τάση των Kata containers να αντιμετωπίζουν προβλήματα κλιμακωσιμότητας υπό υψηλό φορτίο και παρεμβολή υποδηλώνει ότι η χρήση τους πρέπει να συνοδεύεται από επιπλέον μηχανισμούς διαχείρισης πόρων, όπως η δυναμική κατανομή πόρων με προτεραιότητα σε κρίσιμες εφαρμογές και η χρήση ML μοντέλων για πρόβλεψη φορτίου και προληπτική προσαρμογή πόρων με βάση τη χωρητικότητα του συστήματος.

Συνοψίζοντας, η παρούσα διπλωματική εργασία συνεισφέρει στην κατανόηση των επιπτώσεων της εικονικοποίησης στην απόδοση συστημάτων containers, προσδιορίζοντας ποσοτικά το κόστος ασφαλείας που επιβάλλουν τα Kata containers και εντοπίζοντας τα κρίσιμα σημεία όπου η απόδοσή τους υποβαθμίζεται. Τα αποτελέσματα παρέχουν τεκμηριωμένη βάση για τη λήψη αποφάσεων σχετικά με την επιλογή κατάλληλης τεχνολογίας containers ανάλογα με τις ειδικές απαιτήσεις κάθε εφαρμογής και περιβάλλοντος ανάπτυξης.

6.2 Μελλοντικές Επεκτάσεις

Η παρούσα μελέτη ανοίγει πολλαπλές κατευθύνσεις για περαιτέρω έρευνα σχετικά με την απόδοση των Kata containers και την εφαρμογή τους σε διαφορετικά σενάρια.

Μία πρώτη κατεύθυνση αφορά τη διερεύνηση εναλλακτικών hypervisors πέραν του Firecracker. Ενώ το Firecracker σχεδιάστηκε για ελαφριά εικονικοποίηση με έμφαση στην ασφάλεια και την ταχύτητα εκκίνησης, άλλοι hypervisors όπως το QEMU με διαφορετικές παραμέτρους βελτιστοποίησης ή το Cloud Hypervisor που αναπτύσσεται ειδικά για cloud-native περιβάλλοντα, ενδέχεται να προσφέρουν διαφορετικά trade-offs μεταξύ ασφαλείας και απόδοσης. Συγκριτική μελέτη πολλαπλών hypervisors θα μπορούσε να αποκαλύψει ποιες αρχιτεκτονικές επιλογές επηρεάζουν περισσότερο την απόδοση και σε ποια σενάρια χρήσης προτιμάται κάθε λύση.

Η συγκεκριμένη εργασία εστίασε σε μία εφαρμογή επεξεργασίας εικόνας ως φορτίο, ωστόσο τα περιβάλλοντα edge computing φιλοξενούν ποικιλία εφαρμογών με διαφορετικά χαρακτηριστικά, όπως εφαρμογές επεξεργασίας βίντεο σε πραγματικό χρόνο, εφαρμογές machine learning για inference, εφαρμογές Internet of Things με έντονη ροή δεδομένων, και εφαρμογές βάσεων δεδομένων με χαρακτηριστικά I/O-intensive. Η αξιολόγηση των Kata containers με αυτά τα διαφορε-

τικά φορτία θα μπορούσε να αποκαλύψει πώς αυτά τα χαρακτηριστικά κάθε εφαρμογής αλληλεπιδρούν με τους μηχανισμούς εικονικοποίησης και σε ποιες περιπτώσεις το overhead είναι πιο σημαντικό.

Η μελέτη του δυναμικού scaling και της ελαστικότητας των Kata containers σε μεταβαλλόμενα φορτία εργασίας αποτελεί μία ακόμη ερευνητική ευκαιρία. Τα edge computing συστήματα χαρακτηρίζονται από έντονη μεταβλητότητα του φορτίου λόγω της κινητικότητας των χρηστών και των χρονικών προτύπων χρήσης. Η διερεύνηση του πώς τα Kata containers συμπεριφέρονται κατά τη δυναμική αύξηση ή μείωση των πόρων, τη μετανάστευση μεταξύ κόμβων, και την προσαρμογή σε ξαφνικές αιχμές φορτίου θα μπορούσε να παρέχει πληροφορίες για την καταλληλότητά τους σε πραγματικά περιβάλλοντα παραγωγής. Επιπλέον, η ανάπτυξη και αξιολόγηση έξυπνων αλγορίθμων χρονοπρογραμματισμού που λαμβάνουν υπόψη τα ιδιαίτερα χαρακτηριστικά απόδοσης των Kata containers θα μπορούσε να βελτιώσει την αποδοτικότητα των συστημάτων.

Τέλος, η έρευνα σε αρχιτεκτονικές που συνδυάζουν regular και Kata containers στο ίδιο cluster ανοίγει προοπτικές για εξισορρόπηση μεταξύ ασφαλείας και απόδοσης σε επίπεδο συστήματος. Ένα σύστημα ενορχήστρωσης θα μπορούσε να επιλέγει δυναμικά τον κατάλληλο τύπο container για κάθε εφαρμογή βάσει των απαιτήσεών της, χρησιμοποιώντας regular containers για εφαρμογές με υψηλές απαιτήσεις απόδοσης και χαμηλότερες απαιτήσεις ασφαλείας, και Kata containers για εφαρμογές που χειρίζονται ευαίσθητα δεδομένα. Η ανάπτυξη πολιτικών και αλγορίθμων για την αυτοματοποιημένη κατηγοριοποίηση και τοποθέτηση εφαρμογών θα αποτελούσε σημαντική πρακτική συνεισφορά.

Βιβλιογραφία

[1] "Edge Computing." [Online]. Available: https://en.wikipedia.org/wiki/Edge_computing

[2] A. K. Yadav, M. L. Garg, and R. Mehra, "Docker containers versus virtual machine-based virtualization," in *Proceedings of IEMIS 2018, Volume 3*, ser. Advances in Intelligent Systems and Computing, vol. 814. Singapore: Springer, 2019, pp. 141–150.

[3] A. Estrada, M. Flores-Rios, M. Mandujano, O. Jaramillo, and A. Tchernykh, "Containers and virtual machines at scale: A comparative study," in *Proceedings of the 18th International Conference on Distributed Computing and Artificial Intelligence*. Springer, 2016, pp. 33–40.

[4] K. Lee, J. Kim, I.-H. Kwon, H. Park, and C.-H. Hong, "Impact of secure container runtimes on file i/o performance in edge computing," *Applied Sciences*, vol. 13, no. 24, p. 13329, 2023.

[5] X. Sun, Q. Wu, Y. Tan, and F. Wu, "Mvei: An interference prediction model for cpu-intensive application in cloud environment," in *2014 13th International Symposium on Distributed Computing and Applications to Business, Engineering and Science*, 2014, pp. 83–87.

[6] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2015, pp. 171–172.

[7] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "A performance interference model for managing consolidated workloads in qos-aware clouds," in *2011 IEEE/ACM International Conference on Grid Computing*. IEEE, 2011, pp. 170–179.

[8] X. Yu, Q. Zhang, V. Achalkar, and R. Pamu, "Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines," in *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC '11)*. Cascais, Portugal: ACM, 2011, pp. 1–14.

[9] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, and C. Pu, "Understanding performance interference of i/o workload in virtualized cloud environments," in *2010 IEEE 3rd International Conference on Cloud Computing*, 2010, pp. 51–58.

[10] R. C. Chiang and H. H. Huang, "Tracon: Interference-aware scheduling for data-intensive applications in virtualized environments," in *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12.

[11] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*. Santa Clara, CA: USENIX Association, 2020, pp. 419–434.

[12] A. M. Sampaio, J. G. Barbosa, and R. Prodan, "Piasa: A power and interference aware resource management strategy for heterogeneous workloads in cloud data centers," *Simulation Modelling Practice and Theory*, vol. 57, pp. 142–160, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1569190X15001069

[13] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: a performance comparison," *The Journal of Supercomputing*, vol. 71, no. 11, pp. 4110–4143, 2015.

[14] M. Melo Alves, L. Teylo, Y. Frota, and L. M. Drummond, "An interference-aware virtual machine placement strategy for high performance computing applications in clouds," in *2018 Symposium on High Performance Computing Systems (WSCAD)*, 2018, pp. 94–100.

[15] N. Rameshan, L. Navarro, E. Monte, and V. Vlassov, "Stay-away, protecting sensitive applications from performance interference," in *Proceedings of the 15th International Middleware Conference*, ser. Middleware '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 301–312. [Online]. Available: https://doi.org/10.1145/2663165.2663327

[16] W. Viktorsson, C. Klein, and J. Tordsson, "Performance and isolation analysis of runc, gvisor and kata containers runtimes," in *2022 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2022, pp. 107–114.

[17] V. van Rijn and J. S. Rellermeyer, "A fresh look at the architecture and performance of contemporary isolation platforms," in *Proceedings of the*

22nd International Middleware Conference, ser. Middleware '21. New York, NY, USA: ACM, 2021, pp. 128–140.

[18] A. Rajan, R. Bose, V. Cardellini, E. Casalicchio, and V. Interino, "Exploring the performance of container runtimes within kubernetes clusters," in *2023 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2023, pp. 49–56.

[19] Amazon Web Services, "Enhancing kubernetes workload isolation and security using kata containers," https://aws.amazon.com/blogs/containers/enhancing-kubernetes-workload-isolation-and-security-using-kata-containers/, 2023, accessed: 2024-01-15.

[20] A. Randazzo and I. Tinnirello, "Kata containers: An emerging architecture for enabling mec services in fast and secure way," in *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, 2019, pp. 209–214.

[21] F. Sierra-Arriaga, R. Branco, and B. Lee, "Security issues and challenges for virtualization technologies," *ACM Comput. Surv.*, vol. 53, no. 2, May 2020. [Online]. Available: https://doi.org/10.1145/3382190

[22] Wikipedia contributors, "Hypervisor — Wikipedia, the free encyclopedia," https://en.wikipedia.org/wiki/Hypervisor, 2024, accessed: 2025-01-15.

[23] D. Banavalikar, S. Chavhan, A. Dudhanale, and M. Bartere, "Performance evaluation of hypervisors and the effect of virtual cpu on performance," in *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*. IEEE, 2018, pp. 1–6.

[24] Firecracker Team, "Firecracker: Secure and fast microvms for serverless computing," https://firecracker-microvm.github.io/, 2024, accessed: 2025-01-15.

[25] ——, "Firecracker jailer documentation," https://github.com/firecracker-microvm/firecracker/blob/main/docs/jailer.md, 2024, accessed: 2025-01-15.

[26] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/agache

[27] The Kubernetes Authors, "Kubernetes: Production-grade container orchestration," https://kubernetes.io/, 2025.

[28] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Commun. ACM*, vol. 59, no. 5, p. 50–57, Apr. 2016. [Online]. Available: https://doi.org/10.1145/2890784

[29] "Pods | kubernetes," https://kubernetes.io/docs/concepts/workloads/pods/, 2025, accessed: 2025-10-02.

[30] "Deployments | kubernetes," https://kubernetes.io/docs/concepts/workloads/controllers/deployment/, 2025, accessed: 2025-10-02.

[31] D. Goel, "Advanced kubernetes – dzone refcard," https://dzone.com/refcardz/advanced-kubernetes, 2025, accessed: 2025-10-02.

[32] "Architecture | kubernetes," https://kubernetes.io/docs/concepts/architecture/, 2025, accessed: 2025-10-02.

[33] A. M. Beltre, P. Saha, M. Govindaraju, A. Younge, and R. E. Grant, "Enabling hpc workloads on cloud infrastructure using kubernetes container orchestration mechanisms," in *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, 2019, pp. 11–20.

[34] "kubelet | kubernetes," https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/, 2025, accessed: 2025-10-02.

[35] "kube-proxy | kubernetes," https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/, 2025, accessed: 2025-10-02.

[36] "Container runtimes | kubernetes," https://kubernetes.io/docs/setup/production-environment/container-runtimes/, 2025, accessed: 2025-10-02.

[37] K. Lee, J. Kim, I.-H. Kwon, H. Park, and C.-H. Hong, "Impact of secure container runtimes on file i/o performance in edge computing," *Applied Sciences*, vol. 13, no. 24, 2023. [Online]. Available: https://www.mdpi.com/2076-3417/13/24/13329

[38] B. Kunwar and S. Telfer, "I/o performance of kata containers," https://www.stackhpc.com/kata-io-1.html, May 2019, accessed: 2025-10-03.

[39] D. Ghatrehsamani, C. Denninnart, J. Bacik, and M. A. Salehi, "The art of cpu-pinning: Evaluating and improving the performance of virtualization and containerization platforms," in *Proceedings of the 49th International Conference on Parallel Processing (ICPP '20)*. ACM, 2020, pp. 1–11.

[40] "Control cpu management policies on the node | kubernetes," https://kubernetes.io/docs/tasks/administer-cluster/cpu-management-policies/, 2025, accessed: 2025-10-03.

[41] B. Subramaniam and C. Doyle, "Feature highlight: Cpu manager," https://kubernetes.io/blog/2018/07/24/feature-highlight-cpu-manager/, July 2018, accessed: 2025-10-03.

[42] F. Musthafa, "Cpu limits and aggressive throttling in kubernetes," https://engineering.omio.com/cpu-limits-and-aggressive-throttling-in-kubernetes-c5b20bd8a718, Feb 2020, accessed: 2025-10-03.