ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ
ΥΠΟΛΟΓΙΣΤΩΝ

Προσαρμοστικός διαμοιρασμός πόρων σε υπολογιστικά νέφη με χρήση περιεκτών και βαθιάς ενισχυτικής μάθησης

ΔΙΠΛΩΜΑΤΚΗ ΕΡΓΑΣΙΑ

Μιλτιαδης Β. Χρυσόπουλος

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος, 2022

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ
ΥΠΟΛΟΓΙΣΤΩΝ

Προσαρμοστικός διαμοιρασμός πόρων σε υπολογιστικά νέφη με χρήση περιεκτών και βαθιάς ενισχυτικής μάθησης

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Μιλτιάδης Β. Χρυσόπουλος

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 25η Αυγούστου 2022.

Αθήνα, Ιούλιος 2022

.....	.....	.....
Νεκτάριος Κοζύρης	Γεώργιος Γκούμας	Ιωάννης Κωνσταντίνου
Καθηγητής Ε.Μ.Π.	Αναπληρωτής Καθηγητής Ε.Μ.Π.	Επίκουρος Καθηγητής Πανεπιστήμιο Θεσσαλίας

.....

Μιλτιάδης Β. Χρυσόπουλος

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

Περίληψη

Η ελαστική διαχείριση πόρων είναι μια πολύ επιθυμητή λειτουργικότητα σε εφαρμογές που σχεδιάζονται και εκτελούνται σε υπολογιστικά νέφη καθώς πολλές συνηθισμένες πολιτικές τιμολόγησης κοστολογούν με βάση την κατανάλωση των πόρων. Είναι λοιπόν αρκετά σημαντική η βέλτιστη χρησιμοποίηση των πόρων ώστε τα κόστη να είναι τέτοια που να δικαιολογούν τη μεταφορά εφαρμογών σε υπολογιστικά νέφη. Η χρήση περιεκτών (containers) για το σχεδιασμό και την μεταφορά εφαρμογών σε υπολογιστικά νέφη είναι μια πιο αποδοτική προσέγγιση σε σχέση με την συνηθισμένη τακτική της χρήσης εικονικών μηχανών. Αυτό γιατί οι εικονικές μηχανές εισάγουν ένα σημαντικό κόστος σε χρόνο και σε πόρους αφού κάθε μηχανή πρέπει να τρέχει δικό της λειτουργικό σύστημα, σε αντίθεσή με τους περιέκτες που διαμοιράζονται τους πόρους της εικονικής ή φυσικής μηχανής. Η χρήση περιεκτών συνδράμει στην ανάπτυξη εφαρμογών με το πρότυπο των microservices όπου η εφαρμογή χωρίζεται σε πολλές μικρότερες ανεξάρτητες οντότητες οι οποίες μπορούν να τρέχουν ανεξάρτητα. Αυτό το πρότυπο ενισχύει τις δυνατότητες της ελαστικής διαχείρισης πόρων αφού η κάθε οντότητα μπορεί να εκτελείται με διαφορετικό αριθμό περιεκτών ανάλογα με τις ανάγκες.

Η χρήση περιεκτών δημιούργησε γρήγορα ανάγκες συστημικής οργάνωσης και συντονισμού τους καθώς αυξάνεται το πλήθος και η διαφορετικότητα αυτών στο πλαίσιο μιας ενιαίας εφαρμογής. Το σύστημα Kubernetes είναι ένα σύστημα που εξυπηρετεί ακριβώς αυτό τον σκοπό. Ενώ είναι πολύ αποδοτικό σε ζητήματα συντονισμού των περιεκτών πάσχει σε ζητήματα διαχείρισης πόρων αφού προσφέρει μόνο απλοϊκές προσεγγίσεις , οι οποίες είναι ανεπαρκείς για σύνθετες εφαρμογές όπως μια NoSQL βάση που χρησιμοποιήθηκε στα πλαίσια της εργασίας.

Για το λόγο αυτό προτείνουμε ένα μοντέλο βαθιάς ενισχυτικής μάθησης το οποίο λειτουργεί συμπληρωματικά σε ένα σύμπλεγμα μηχανών Kubernetes για τον έλεγχο και τη δυναμική διαχείριση των περιεκτών της εφαρμογής. Για να ξεπεράσουμε θεμελιώδεις δυσκολίες σχετικά με την εκπαίδευση τέτοιων μοντέλων σε ρεαλιστικές συνθήκες και να μειώσουμε το χρόνο και τον αριθμό των αλληλεπιδράσεων του μοντέλου με το σύστημα, χρησιμοποιούμε τεχνικές βελτιστοποίησης της εκπαίδευσης από τη σύγχρονη βιβλιογραφία καθώς και αλγορίθμους ασύγχρονης ενισχυτικής μάθησης. Δείχνουμε ότι το προτεινόμενο σύστημα έχει συστηματική βελτίωση της απόδοσης σε σχέση με το αντίστοιχο σύστημα σύγχρονης εκπαίδευσης και πως καταφέρνει να εξάγει βελτιωμένες πολιτικές λήψης αποφάσεων ακόμα και σε χαμηλής ποιότητας δεδομένων.

Λέξεις Κλειδιά: Ελαστικότητα, Διαχείριση Πόρων, Περιέκτης, Υπολογιστικό Νέφος, Βαθιά Ενισχυτική Μάθηση, Ασύγχρονη Ενισχυτική Μάθηση, CQL, Kubernetes, NoSQL

Abstract

Elastic resource allocation is a highly desirable functionality in cloud native applications because many common pricing policies of cloud vendors are based on resource allocation. Consequently, optimal utilization of computational resources is important so that a migration of an application from on-premises to the cloud is reasonable pricewise. Using containerized applications for cloud deployment is a more promising alternative to the usual deployment with virtual machines, in terms of resource allocation. The reason is that every virtual machine must run its own operating system kernel, a fact that introduces significant time and resource overhead. Containers on the other hand, share the resources of the hosting machine and take less time to initiate and terminate their execution. Additionally, containerization promotes the microservices paradigm, where the application is divided in discrete and independent components. This enhances the elasticity capabilities, since every component can be scaled independently according to the needs of the application at different times.

Containerization rapidly creates needs for systematic orchestration and coordination of the containers, as application comprise of containers that increase in number and complexity. The Kubernetes system is an orchestration system for containerized applications that fulfills this exact need. Although it is very effective in terms of container coordination it lacks in resource administration because it offers only simplistic threshold-based approaches that are inadequate for complex application components, such as a NoSQL database that was used for the scope of this work.

To address this issue, we propose a Deep Reinforcement Learning model that is complementary to a Kubernetes cluster and its role is to monitor and automatically scale the containers of the application according to incoming workload. To overcome core obstacles that are involved in training such models in realistic scenarios and reduce the training time as well as the number of interactions between the model and the system, we use training optimization techniques from contemporary literature as well as offline Reinforcement Learning algorithms. We provide empirical results that show that our model achieves systematic improvement compared to its online equivalent for a given number of experiences and that it is able to extract improved decision-making policies even from data of lower quality.

Keywords: Elasticity, Resource Management, Container, Cloud Computing, Deep Reinforcement Learning, Offline Reinforcement Learning, CQL, Kubernetes, NoSQL

Table of Contents

List of Figures

1.Εισαγωγή

1.1 Κίνητρο

Οι εγγενείς εφαρμογές cloud είναι ένα από τα κύρια σημεία εστίασης της ανάπτυξης εταιρικού λογισμικού. Το 2021 περίπου το 30% του νέου ψηφιακού φόρτου εργασίας που δημιουργείται αναπτύσσεται σε εγγενείς πλατφόρμες cloud και εκτιμάται ότι το ποσοστό αυτό μπορεί να ανέλθει σε περισσότερο από 90% έως το 2025. Η δυναμική και επεκτάσιμη φύση των εγγενών εφαρμογών cloud είναι μια πολλά υποσχόμενη εναλλακτική λύση σε ανάπτυξη εφαρμογών επειδή απαλλάσει μερικώς έναν οργανισμό από το κόστος και τον κίνδυνο διατήρησης δαπανηρής υποδομής για την υποστήριξη των λειτουργιών λογισμικού του. Επιπλέον, η επέκταση των συστημάτων για την κάλυψη νέων αναγκών είναι περιορισμένη, χρονοβόρα και ακόμη και αδύνατη σε ορισμένα σενάρια.

Το Containerization ενισχύει τις δυνατότητες των εγγενών εφαρμογών cloud επειδή προσφέρει μια λιγότερο κοστοβόρα εναλλακτική λύση σε εικονικές μηχανές που πρέπει να τρέχουν λειτουργικό σύστημα για να εκτελεστούν. Χρησιμοποιώντας contianers, τα στοιχεία μιας εφαρμογής μπορούν να εκτελούνται με σημαντικά μικρότερο κόστος υλικού και χρόνου. Αξιοποιώντας αυτές τις ιδιότητες, οι εφαρμογές cloud μπορούν να οργανωθούν σε στοιχεία που εκτελούν ατομικές λειτουργίες της εφαρμογής, αντί να εκτελούν μονολιθικά στιγμιότυπα της εφαρμογής. Αυτό το πρότυπο ανάπτυξης λογισμικού ονομάζεται microservices και προσφέρει αυξημένη ευελιξία όσον αφορά την ελαστικότητα, επειδή κάθε στοιχείο μπορεί να εκτελείται με διαφορετικό αριθμό containers τα οποία αυξομειώνονται ανεξάρτητα ανάλογα με τις ανάγκες.

Καθώς οι containerized εφαρμογες γίνονται πιο περίπλοκες και προστίθενται πιο σύνθετα στοιχεία, η ανάπτυξη, η παρακολούθηση, η κλιμάκωση και η σύνδεση των στοιχείων της εφαρμογής γίνεται μια επίπονη διαδικασία. Η ταχεία αύξηση της πολυπλοκότητας DevOps των εφαρμογών με κοντέινερ περιόρισε την υιοθέτησή τους και δημιούργησε την ανάγκη για μηχανισμούς συστηματικού συντονισμού και οργανωσής τους. Το Kubernetes είναι ένα σύστημα λογισμικού που επιλύει αποτελεσματικά το πρόβλημα αυτό, γεγονός που είχε ως αποτέλεσμα την αυξημένη υιοθέτησή του από τις επιχειρήσεις από την κυκλοφορία του το 2014. Το Kubernetes προσφέρει επίσης λειτουργίες αυτόματου scaling βάσει πόρων, όπως το Horizontal Pod Autoscaler (HPA). Αυτή η λειτουργία χρησιμοποιεί κατώφλια χρήσης CPU και μνήμης για να εκτιμήσει τον βέλτιστο αριθμό containers που θα εκτελεστούν. Ορισμένοι πάροχοι υπηρεσιών cloud προσφέρουν παρόμοιες λειτουργικότητες αυτόματου scaling βάσει κατωφλίου που αυξάνουν τον αριθμό των παρουσιών εφαρμογών καθώς αυξάνεται η εισερχόμενη κίνηση. Αυτές οι μέθοδοι είναι υπεραπλουστευτικές και μπορούν να παρέχουν μόνο περιορισμένες εγγυήσεις απόδοσης.

Μια εναλλακτική προσέγγιση για την αυτόματη κλιμάκωση βάσει κατωφλίου είναι η χρήση αλγορίθμων από το πεδίο της Ενισχυτικής Μάθησης (RL). Η Ενισχυτική Μάθηση επισημοποιεί την ιδέα ενός μοντέλου που μαθαίνει αποτελεσματικές πολιτικές λήψης αποφάσεων εκτελώντας αλληλεπιδράσεις trial and error με ένα σύστημα. Η βαθιά ενισχυτική μάθηση ενισχύει τις δυνατότητες του RL χρησιμοποιώντας νευρωνικά δίκτυα ως μη γραμμικούς προσεγγιστές συναρτήσεων υψηλής τάξης. Με αυτόν τον τρόπο, προβλήματα που περιγράφονται από πιο σύνθετους χώρους καταστάσεων και δεν

μπορούσαν να λυθούν με αρχικούς αλγόριθμους RL, είναι πλέον επιλύσιμα. Ωστόσο, υπάρχει ένας βασικός περιορισμός στην εφαρμογή της Εκμάθησης Βαθιάς Ενίσχυσης στο πρόβλημα της αυτόματης δέσμευση πόρων. Στα τυπικά προβλήματα μάθησης βαθιάς ενίσχυσης που αναφέρονται στη βιβλιογραφία, ο αριθμός των βημάτων εκπαίδευσης που απαιτούνται για την επίτευξη μιας βέλτιστης λύσης είναι της τάξης των εκατομμυρίων. Σε τυπικά σενάρια αυτόματου scaling, ο χρόνος που απαιτείται για να εφαρμοστεί μια ενέργεια είναι η της τάξης των λεπτών. Επιπλέον, η αλληλεπίδραση με την εφαρμογή με τυχαίο τρόπο δεν είναι σε ορισμένες περιπτώσεις επιθυμητή επειδή μπορεί να προκαλέσει διακοπές ή να την οδηγήσει σε καταστροφικές καταστάσεις. Λαμβάνοντας υπόψη τα παραπάνω, είναι προφανές ότι για να εξαχθεί μια εφικτή λύση για το πρόβλημα της αυτόματης κλιμάκωσης ο αριθμός των αλληλεπιδράσεων με το σύστημα πρέπει να μειωθεί σημαντικά ή εναλλακτικά να μεγιστοποιηθεί το κέρδος πληροφοριών από κάθε αλληλεπίδραση.

1.2 Σχετικές Εργασίες

Ο πιο συνηθισμένος τρόπος αντιμετώπισης του προβλήματος της ελαστικότητας είναι το auto-scaling. Ο μηχανισμός auto-scaling της Amazon για παράδειγμα αυξάνει ή μειώνει δυναμικά τους πόρους ενός χρήστη με βάση τα όρια που εφαρμόζονται στις συγκεκριμένες μετρήσεις του συμπλέγματος χρηστών. Το Azure της Microsoft και το Celar χρησιμοποιούν την ίδια τεχνική. Ωστόσο αυτές οι προσεγγίσεις είναι δύσκολο να βαθμονομηθούν και να βελτιστοποιηθούν.

Οι συγγραφείς του (R. Taft, 2018) χρησιμοποιούν έναν αλγόριθμο δυναμικού προγραμματισμού που προσπαθεί να προσδιορίσει μέσω μιας σειράς προηγούμενων εμπειριών τη βέλτιστη συμπεριφορά για την επόμενη κατάσταση του συστήματος. Οι Μαρκοβιανές Διακασίες Αποφάσεων (MDPs) και οι αλγόριθμοι Ενίσχυσης μάθησης έχουν χρησιμοποιηθεί για την αντιμετώπιση του ζητήματος για την πρόβλεψη μιας κατάστασης νέφους και την παροχή πόρων. Ωστόσο, η αποτελεσματικότητα αυτών των προσεγγίσεων μειώνεται, καθώς αυξάνεται ο αριθμός των πιθανών καταστάσεων. Οι παράμετροι εισόδου του συστήματος (μετρικές του συμπλέγματος) είναι συνεχείς μεταβλητές. Συνεπώς, ο αριθμός των διακριτών καταστάσεων μπορεί να αυξηθεί εκθετικά.

Για τη διαχείριση αυτού του ζητήματος στο (K. Lolos, 2017) οι συγγραφείς προτείνουν μια προσέγγιση RL σε συνδυασμό με αλγόριθμους δένδρων αποφάσεων, προκειμένου να χωρίσουν τις παραμέτρους εισόδου με βάση ορισμένα κριτήρια διαχωρισμού. Αυτή η προσέγγιση καταφέρνει να γενικεύσει τα δεδομένα εισόδου και να εκπαιδεύσει τον πράκτορα έτσι ώστε να μπορεί να ανακαλύψει μόνος του ποιες παράμετροι κατάστασης έχουν σημασία για το επιθυμητό αποτέλεσμα και ποιες όχι. Ωστόσο, αυτή η προσέγγιση αντιμετωπίζει επίσης μεγάλο χώρο καταστάσεων και χρειάζεται επίσης ένα μεγάλο σύνολο δεδομένων για να δείξει τις δυνατότητες γενίκευσης.

Οι συγγραφείς του (Κωνσταντίνος Μπιτσάκος, 2018) πρότειναν ένα μοντέλο μάθησης βαθιάς ενίσχυσης για την αντιμετώπιση του προβλήματος της ελαστικότητας σε περιβάλλοντα του cloud. Το μοντέλο είναι σε θέση να συγκλίνει σε μια λύση και να παρέχει αυξημένες ανταμοιβές σε σύγκριση με προηγούμενες προσεγγίσεις που δεν

χρησιμοποιούσαν νευρωνικό δίκτυο. Το κύριο ζήτημα εξακολουθεί να είναι το γεγονός ότι ο αριθμός των δειγμάτων εκπαίδευσης είναι σημαντικά μεγάλος.

Οι συγγραφείς του (Lucia Schuler, 2021) χρησιμοποιούν αλγόριθμους μάθησης Ενίσχυσης για να αντιμετωπίσουν το πρόβλημα του προσαρμοστικού auto-scaling για serverless εφαρμογές. Η προσέγγισή τους αφορά την κατανομή του διαθέσιμου φόρτου εργασίας σε container και την παρακολούθηση της απόδοσης του συστήματος όσον αφορά τον το throughput και του latency βάσει των ταυτόχρονων αιτημάτων που πρέπει να εξυπηρετήσει κάθε κοντέινερ. Στη συνέχεια, δημιουργούν μια πολιτική χρησιμοποιώντας Ενισχυτική Μάθηση για να εκτελέσουν τη βέλτιστη κατανομή του φόρτου εργασίας. Η προσέγγισή τους είναι παρόμοια με τη δική μας, αν και εφαρμόζεται σε serverless εφαρμογές. Ωστόσο, οι καταστάσεις περιγράφονται χρησιμοποιώντας μόνο τρεις παραμέτρους, γεγονός που καθιστά τον χώρο εξερεύνησης πολύ μικρότερο και ευκολότερο να συγκλίνει προς μια λύση.

1.3 Προτεινόμενη Λύση

Η προτεινόμενη λύση αυτής της διπλωματικής εργασίας είναι η δημιουργία ενός συστήματος που παρακολουθεί μια containerized εφαρμογή και αυξομειώνει δυναμικά τις τα εκτελούμενα containers και κατά συνέπεια τους υπολογιστικούς πόρους που καταναλώνονται. Για να αναπτύξουμε αυτόν το μοντέλο, δημιουργούμε δύο νευρωνικά δίκτυα που εκπαιδεύονται χρησιμοποιώντας σύγχρονους και ασύγχρονους αλγόριθμους ενισχυτικής μάθησης. Υπάρχουν δύο κύριες προκλήσεις που πρέπει να αντιμετωπιστούν από την προτεινόμενη λύση.

• Ο αριθμός των παραμέτρων που απαιτούνται για την περιγραφή της κατάστασης του συστήματος μπορεί να αυξηθεί, ανάλογα με την πολυπλοκότητα της αναπτυσσόμενης εφαρμογής. Επίσης, οι παράμετροι δεν είναι διακριτές. Αυτό καθιστά αναποτελεσματικές τις πινακοποιημένες εφαρμογές της Ενισχυτικής Μάθησης και κατά συνέπεια δικαιολογείται η χρήση ενός νευρωνικού δικτύου.

• Ο χρόνος μεταξύ των διαδοχικών ενεργειών είναι της τάξης των λεπτών. Αυτό περιορίζει τον ρυθμό συγκέντρωσης δεδομένων, πράγμα που σημαίνει ότι το μοντέλο πρέπει να εξάγει όσο το δυνατόν περισσότερες πληροφορίες από τα διαθέσιμα δεδομένα.

Για να αντιμετωπίσουμε αυτές τις προκλήσεις, προτείνουμε ένα μοντέλο που αποτελείται από δύο νευρωνικά δίκτυα. Το πρώτο μοντέλο εκπαιδεύεται με σύγχρονο τρόπο, χρησιμοποιώντας τον αλγόριθμο Double Deep Q Learning. Το δεύτερο εκπαιδεύεται με ένα σύνολο δεδομένων χρησιμοποιώντας τον αλγόριθμο Conservative Q Learning χωρίς να αλληλεπιδρά με το σύστημα. Το σύνολο δεδομένων αποτελείται από τις εκπαιδευτικές εμπειρίες που δημιουργούνται από το σύγχρονο μοντέλο κατά τη διάρκεια της εκπαίδευσής του. Ουσιαστικά, το μοντέλο εκτός σύνδεσης λαμβάνει τα δεδομένα που παράγονται από την πολιτική του σύγχρονου μοντέλου και εκπαιδεύεται χωρίς περαιτέρω αλληλεπίδραση με το σύστημα. Για να μειώσουμε τον χρόνο που απαιτείται για την εκτέλεση των ενεργειών κλιμάκωσης, αναπτύσσουμε την εφαρμογή μας μέσα σε ένα κατανεμημένο cluster Kubernetes για να αξιοποιήσουμε τις αυτοματοποιημένες δυνατότητες συντονισμού και διαμοιρασμού των containers στους worker nodes του και τους ελεγκτές ανοιχτού κώδικα που χειρίζονται τις απαραίτητες λειτουργίες, μετά από αύξηση ή μείωση του αριθμού των

containers της αναπτυγμένης εφαρμογή. Η εφαρμογή που επιλέξαμε να παρακολουθήσουμε είναι ένα cluster Cassandra. Η επιλογή αυτής της εφαρμογής βασίζεται στο γεγονός ότι η απόδοσή της εξαρτάται από αρκετές παραμέτρους , γεγονός που αναδεικνύει την αποτελεσματικότητα του μοντέλου μας.

Δοκιμάζουμε και τα δύο μοντέλα σε πολλά σημεία της εκπαίδευσης και δείχνουμε ότι το ασύγχρονο ξεπερνά συστηματικά τις επιδόσεις της σύγχρονης εκπαίδευσης μέχρι ένα συγκεκριμένο μέγεθος δεδομένων. Παρέχουμε επίσης εμπειρικά αποτελέσματα που υποδεικνύουν ότι από το σημείο και μετά όπου η σύγχρονη εκπαίδευση ξεπερνά την ασύχρονη, τα κέρδη απόδοσης μειώνονται δραματικά σε σύγκριση με τα βήματα εκπαίδευσης που απαιτούνται για την επίτευξη αυτής της βελτίωσης.

1.4 Τεχνολογικό Υπόβαθρο

Περιέκτες

Οι τεχνολογίες εικονικοποίησης αποτελούν ουσιαστικό μέρος της σύγχρονης υποδομής cloud, επειδή διευκολύνουν την αποτελεσματική χρήση των πόρων των φυσικών μηχανών που φιλοξενούν εικονικές μηχανές (VM). Επιπλέον, απομονώνουν εικονικές μηχανές, κάτι που είναι σημαντικό για τη διασφάλιση της ασφάλειας. Ωστόσο, η εικονικοποίηση συνοδεύεται από σημαντική επιβάρυνση υλικού, καθώς κάθε VM πρέπει να έχει τον αποκλειστικό πυρήνα του, κάτι που οδηγεί σε αυξημένη χρήση πόρων. Επίσης, ο hypervisor, το πρόγραμμα που είναι υπεύθυνο για τη δημιουργία και τη λειτουργία VM σε μια φυσική μηχανή, καταναλώνει από μόνος του ένα αρκετά μεγάλο κλάσμα πόρων (Scheepers, 2014).

Οι περιέκτες μετριάζουν αυτό το πρόβλημα παρέχοντας μια «ελαφριά» εναλλακτική για την εκτέλεση μεμονωμένων εφαρμογών, ενώ μοιράζονται τον ίδιο πυρήνα του κεντρικού λειτουργικού συστήματος (OS). Το Containerization εξελίχθηκε από τις cgroups Linux (citrix.com, 2022), οι οποίες έγιναν Linux containers(LXC). Οι ομάδες C απομονώνουν και ελέγχουν τους πόρους που μπορεί να εκχωρήσει οποιαδήποτε δεδομένη διεργασία, για παράδειγμα τον αριθμό νήματος και τη χρήση της CPU ή της μνήμης RAM. Το LXC παρέχει πρόσθετη απομόνωση χρησιμοποιώντας χώρους ονομάτων. Οι χώροι ονομάτων απομονώνουν περαιτέρω τους περιέκτες, που σημαίνει ότι κάθε περιέκτης έχει το αποκλειστικό του σύστημα αρχείων, τη στοίβα δικτύου, τη διαχείριση χρηστών και τα αναγνωριστικά διεργασιών. Χρησιμοποιώντας αυτήν την αφαίρεση, κάθε περιέκτης μπορεί να εκτελέσει τη δική του διανομή λειτουργικού συστήματος ανεξάρτητα από το λειτουργικό σύστημα της μηχανής φιλοξενίας, δεδομένου ότι η υποκείμενη διανομή εκτελεί έναν πυρήνα Linux (Scheepers, 2014). Με αυτόν τον τρόπο, η εκκίνηση ενός περιέκτη έχει ως αποτέλεσμα την ίδια εμπειρία ανεξάρτητα από το υπολογιστικό περιβάλλον. Αυτή η συμβατότητα μεταξύ πλατφορμών είναι απαραίτητη για τα σημερινά περιβάλλοντα cloud, τα οποία συνήθως αποτελούνται από ετερογενή συστήματα και λειτουργικά συστήματα.

Οι εφαρμογές μέσα σε περιέκτες έχουν μειώσει σημαντικά τους χρόνους εκκίνησης, καθώς δεν υπάρχει ανάγκη εκκίνησης ενός λειτουργικού συστήματος, εκτός από ορισμένα δυαδικά αρχεία που περιλαμβάνονται στο αρχείο του περιέκτη. Αυτό σημαίνει ότι η ανάπτυξη και η καταστροφή περιεκτών είναι πολύ φθηνότερη από άποψη πόρων και

χρόνου. Τα προαναφερθέντα χαρακτηριστικά επιτρέπουν μεγαλύτερη ευελιξία κατά την κλιμάκωση των εφαρμογών cloud. Αν λάβουμε επίσης υπόψη το γεγονός ότι η πιο κοινή προσέγγιση για την ανάπτυξη εγγενών εφαρμογών cloud είναι οι μικροϋπηρεσίες, όπου η εφαρμογή χωρίζεται σε ξεχωριστά τεχνουργήματα λογισμικού, τα οποία λειτουργούν και κλιμακώνονται ανεξάρτητα και επικοινωνούν μόνο μέσω API, είναι προφανές ότι η αποθήκευση είναι

η πιο κατάλληλη τεχνική απομόνωσης για τη διευκόλυνση αυτού του παραδείγματος ανάπτυξης λογισμικού.

Kubernetes

Το Docker παρέχει ένα πλαίσιο για τη δημιουργία κοντέινερ που είναι αναπαραγώγιμα και λειτουργούν με τον ίδιο τρόπο, ανεξάρτητα από την υποκείμενη αρχιτεκτονική. Επίσης, παρέχοντας αποτελεσματικούς τρόπους αποθήκευσης και εκτέλεσης των κοντέινερ, προωθεί και υποστηρίζει το παράδειγμα ανάπτυξης λογισμικού των μικροϋπηρεσιών. Στην ιδανική περίπτωση, κάθε κοντέινερ εκτελεί ένα μόνο στοιχείο λογισμικού της εφαρμογής που εκτελείται και μπορεί να κλιμακωθεί ανεξάρτητα. Ωστόσο, καθώς οι εφαρμογές γίνονται πιο περίπλοκες και τα στοιχεία λογισμικού αυξάνονται σε αριθμό και ποικιλία, η εκτέλεση DevOps γίνεται μια επίπονη εργασία και η εκτέλεση ενημερώσεων και κλιμάκωσης δεν είναι τετριμμένη διαδικασία. Οι παραπάνω εργασίες δημιούργησαν την ανάγκη για έναν μηχανισμό συντονισμού της ανάπτυξης, της παρακολούθησης, της συντήρησης και την κλιμάκωσης των εφαρμογών σε κοντέινερ. Το Kubernetes, που κυκλοφόρησε αρχικά το 2014, είναι μια προσπάθεια δημιουργίας ενός τέτοιου μηχανισμού.

Μερικά από τα βασικά βοηθητικά προγράμματα του Kubernetes είναι:

• Ανακάλυψη υπηρεσίας και εξισορρόπηση φορτίου: Το Kubernetes μπορεί να αποκαλύψει ένα κοντέινερ χρησιμοποιώντας το όνομα DNS ή χρησιμοποιώντας τη δική του διεύθυνση IP. Εάν η επισκεψιμότητα σε ένα κοντέινερ είναι υψηλή, το Kubernetes μπορεί να φορτώσει επιπλέον αντίγραφα του ίδιου κοντέινερ και να διανείμει την κίνηση του δικτύου έτσι ώστε η λειτουργία να είναι σταθερή.

• Ενορχήστρωση αποθήκευσης: Το Kubernetes σάς επιτρέπει να προσαρτηθεί αυτόματα ένα σύστημα αποθήκευσης, όπως τοπικούς αποθηκευτικούς χώρους, μόνιμο αποθηκευτικός χώρος δημόσιους παρόχους cloud και άλλα.

• Αυτοματοποιημένη διαχείριση κοντέινερ: Μπορείτε να περιγράψετε την επιθυμητή κατάσταση για τα κοντέινερ που έχουν αναπτυχθεί χρησιμοποιώντας το Kubernetes και μπορεί να αλλάξει την πραγματική κατάσταση στην επιθυμητή κατάσταση με ελεγχόμενο ρυθμό. Για παράδειγμα, μπορείτε να αυτοματοποιήσετε το Kubernetes για να δημιουργήσετε νέα κοντέινερ για την ανάπτυξή σας, να αφαιρέσετε υπάρχοντα κοντέινερ και να υιοθετήσετε όλους τους πόρους τους στο νέο κοντέινερ.

• Αυτόματη αναδιανομή κοντέινερ: Το Kubernetes δέχεται ένα σύμπλεγμα κόμβων που μπορεί να χρησιμοποιήσει για την εκτέλεση εργασιών με κοντέινερ. Δίντονας στο Kubernetes πόση CPU και μνήμη (RAM) χρειάζεται κάθε κοντέινερ, το Kubernetes μπορεί να τοποθετήσει κοντέινερ στους κόμβους για να κάνει την καλύτερη χρήση των πόρων.

• Το αυτο-θεραπευόμενο Kubernetes επανεκκινεί τα κοντέινερ που αποτυγχάνουν, αντικαθιστά τα κοντέινερ, σκοτώνει τα κοντέινερ που δεν ανταποκρίνονται στον καθορισμένο από τον χρήστη έλεγχο υγείας και δεν τα διαφημίζει στους πελάτες μέχρι να είναι έτοιμοι να εξυπηρετηθούν.

Συνολικά, το Kubernetes παρέχει εργαλεία για τη διασφάλιση της ανθεκτικότητας και της ανοχής σε σφάλματα της εφαρμογής, καθώς και ελάχιστο ή καθόλου χρόνο διακοπής λειτουργίας κατά τη διάρκεια ενημερώσεων σε περιβάλλοντα παραγωγής.

Βασικά Στοιχεία του Kubernetes

Kube-apiserver: Είναι το frontend του Kubernetes cluster. Επικυρώνει δεδομένα και εκθέτει την κοινόχρηστη κατάσταση του cluster σε όλα τα άλλα στοιχεία του , για παράδειγμα ομάδες, υπηρεσίες και ελεγκτές. Παρέχει επίσης ένα API που ο διαχειριστής του cluster μπορεί να χρησιμοποιήσει την οθόνη και να πραγματοποιήσει μη αυτόματες αλλαγές στη διαμόρφωση του. Μπορεί να κλιμακωθεί οριζόντια για να προσφέρει μεγαλύτερη διαθεσιμότητα όταν αυξάνεται η κίνηση.

Etcd: Είναι μια μόνιμη βάση δεδομένων κλειδιού-τιμής που χρησιμοποιεί το Kubernetes για να αποθηκεύσει την κατάσταση του cluster καθώς και τις προηγούμενες καταστάσεις. Κάθε κλειδί που δημιουργείται στη βάση δεδομένων είναι αμετάβλητο. Αυτό σημαίνει ότι οι επακόλουθες ενημερώσεις στην τιμή του κλειδιού δημιουργούν νέες εκδόσεις του κλειδιού αντί να μεταλλάσσονται τα δεδομένα του υπάρχοντος κλειδιού. Τούτου λεχθέντος, είναι προφανές ότι οι εκδόσεις κάθε κλειδιού αυξάνονται μονότονα σε όλο τον κύκλο ζωής του cluster. Για λόγους αποδοτικότητας αποθήκευσης, οι παλαιότερες εκδόσεις είναι συμπαγείς. Όταν διαγράφεται ένα ζεύγος κλειδιού-τιμής, η έκδοσή του επαναφέρεται στο 0 και στη συνέχεια στην επόμενη συμπίεση όλα τα κλειδιά με την έκδοση 0 δεν περιλαμβάνονται στη συμπύκνωση. Όσον αφορά την υλοποίηση, το etcd χρησιμοποιεί ένα μόνιμο δέντρο b+ για την αποθήκευση πληροφοριών. Τα κλειδιά του δέντρου στη μνήμη δείχνουν την πιο πρόσφατη μετάλλαξη των δεδομένων. Κατά τη διάρκεια συμπίεσης, η δομή ενημερώνεται και οι νεκροί δείκτες αφαιρούνται (https://etcd.io/, 2022).

Kube-scheduler: Όπως δηλώνεται από το όνομα, ο scheduler είναι υπεύθυνος για την παρακολούθηση νέων Pods που δεν έχουν εκχωρηθεί σε έναν κόμβο προς εκτέλεση και, στη συνέχεια, επιλέγει τον κόμβο στον οποίο θα εκτελεστούν. Για να επιλέξει τον καταλληλότερο κόμβο για κάθε Pod, ο scheduler εκτελεί δύο λειτουργίες σε υπάρχοντες κόμβους, φιλτράρισμα και βαθμολόγηση.

Κάθε pod έρχεται με ένα σύνολο απαιτήσεων. Αυτές οι απαιτήσεις μπορεί να αφορούν στοιχεία όπως τους απαραίτητους πόρους. Για παράδειγμα, ένα pod που εκτελεί ένα στοιχείο λογισμικού που πρέπει να αποθηκεύει κατάσταση, πρέπει να εκτελείται σε έναν κόμβο που περιέχει τις σωστές πληροφορίες κατάστασης σε μια εφήμερη ή μόνιμη μορφή. Επίσης, οι κανόνες προγραμματισμού συνάφειας ή αντι-συνάφειας που περιλαμβάνονται στην προδιαγραφή pod, δηλώνουν ρητά τους κόμβους ή/και τις ομάδες με τους οποίους μπορεί ή δεν μπορεί να γίνει εκτέλεση του νέου pod στο ίδιο μηχάνημα. Λαμβάνοντας υπόψη τους παραπάνω περιορισμούς, ο scheduler δημιουργεί μια λίστα με όλους τους εφικτούς κόμβους, τους κόμβους στους οποίους μπορεί να προγραμματιστεί το pod χωρίς να παραβιάζεται κανένας από τους περιορισμούς. Εάν η λίστα των εφικτών κόμβων είναι κενή, τότε το Pod παραμένει σε μη προγραμματισμένη κατάσταση μέχρι να παρατηρηθεί

ένας κατάλληλος κόμβος. Αυτός μπορεί να είναι είτε ένας νέος κόμβος είτε ένας υπάρχων κόμβος, του οποίου ο φόρτος εργασίας άλλαξε. Αφού δημιουργηθεί η λίστα των εφικτών κόμβων, οι κόμβοι ταξινομούνται με βάση τις συναρτήσεις βαθμολόγησης. Η βαθμολόγηση διασφαλίζει ένα τοπικό βέλτιστο που είναι πιθανό να οδηγήσει σε καλύτερη κατάσταση συμπλέγματος διαδοχικές αναθέσεις pod σε κόμβους. Υπάρχουν πολλοί παράγοντες που λαμβάνονται υπόψη κατά τη βαθμολόγηση κόμβου. Για παράδειγμα, είναι λιγότερο πιθανό να επιλεγεί ένας κόμβος που πρόκειται να φτάσει την πλήρη χωρητικότητα πόρων του αφού του εκχωρηθεί το νέο pod. Είναι πιο πιθανό να επιλεγεί ένας κόμβος που έχει ήδη αποθηκευμένη την εικόνα του docker. Επίσης, ο scheduler κατατάσσει τους κόμβους, που ήδη εκτελούν κοντέινερς που ανήκουν στην ίδια υπηρεσία, χαμηλότερα για να εξασφαλίσει μια πιο ομοιόμορφη κατανομή του φόρτου εργασίας της υπηρεσίας εντός του συμπλέγματος. Με αυτόν τον τρόπο, εάν ένας δεδομένος κόμβος αποτύχει, είναι λιγότερο πιθανό να προκαλέσει διακοπή στην υπηρεσία. Αφού ο scheduler υπολογίσει τις βαθμολογίες κάθε κόμβου, επιλέγεται ο κόμβος με την υψηλότερη κατάταξη και ο το pod ανατίθεται σε αυτόν τον κόμβο και ενημερώνει τον διακομιστή kube-apiserver.

Βασικές Ιδέες του Kubernetes

Pods: Τα Pods είναι η μικρότερη μονάδα υπολογισμού που μπορεί να αναπτυχθεί στο οικοσύστημα Kubernetes. Το pod είναι μια ομάδα κοντέινερ που προγραμματίζονται πάντα στον ίδιο κόμβο και μοιράζονται τους ίδιους υπολογιστικούς πόρους, σύστημα αρχείων και αποθήκευση. Το κοινόχρηστο περιβάλλον σε ένα pod αποτελείται από ένα κοινό σύνολο cgroups και χώρων ονομάτων, ανάλογο με μια ρύθμιση απομόνωσης κοντέινερ Docker. Τα Pods αντιπροσωπεύουν μια συνεκτική μονάδα εξυπηρέτησης της εφαρμογής μας και αυτά τα κοντέινερ πρέπει πάντα να λειτουργούν και να κλιμακώνονται μαζί. Τα Pod σπάνια δημιουργούνται ως αυτόνομα αντικείμενα επειδή ένα Pod είναι μια εφήμερη δομή, που σημαίνει ότι εάν δεν δηλωθούν πρόσθετα αντικείμενα που χρησιμοποιούν ένα συγκεκριμένο Pod, τότε ο κύκλος ζωής του μπορεί να τερματιστεί χωρίς περαιτέρω ειδοποίηση, λόγω έλλειψης πόρων για παράδειγμα.

Deployments: Είναι αντικείμενα που επιτρέπουν την εκτέλεση κυλιόμενων ενημερώσεων ή επαναφοράς σε υπάρχουσες ομάδες Pods ή τη δημιουργία νέων. Είναι ένα σημαντικό χαρακτηριστικό του Kubernetes επειδή ένα από τα σημαντικότερα ζητήματα που ισχυρίζεται ότι μπορεί να το λύσει, τα ζητήματα ανθεκτικότητας και τον ελάχιστο χρόνο διακοπής λειτουργίας κατά τη διάρκεια των ενημερώσεων. Κατά τη διάρκεια μιας κυλιόμενης ενημέρωσης, οι ομάδες που εκτελούν την πιο πρόσφατη διαμόρφωση των παρεχόμενων προδιαγραφών, προγραμματίζονται σταδιακά στους κόμβους και ξεκινούν. Καθώς περισσότερες ομάδες που εκτελούν την πιο πρόσφατη έκδοση φτάνουν σε κατάσταση ετοιμότητας, οι ομάδες που εκτελούν την παλιά έκδοση σταδιακά τερματίζονται και αντικαθίστανται. Από προεπιλογή, οι αναπτύξεις διασφαλίζουν ότι θα είναι διαθέσιμο περισσότερο από το 25% του επιθυμητού αριθμού ομάδων ανά πάσα στιγμή και όχι περισσότερο από το 125% του επιθυμητού αριθμού ομάδων που ξεκινούν ταυτόχρονα. Φυσικά, αυτές οι παράμετροι μπορούν να διαμορφωθούν. Εάν για κάποιο λόγο προκύψει μια εντολή αλλαγής του αριθμού των εκτελούμενων κοντέινερ κατά τη διάρκεια μιας κυλιόμενης ενημέρωσης, για παράδειγμα εάν είναι ενεργοποιημένη η οριζόντια αυτόματη

κλιμάκωση, τότε το σύμπλεγμα εκτελεί ανάλογη κλίμακα. Αυτό σημαίνει ότι το σύμπλεγμα δημιουργεί ομάδες που εκτελούν την πιο πρόσφατη έκδοση καθώς και ομάδες που εκτελούν την παλιά, σε αναλογία που διατηρεί την αναλογία πριν ξεκινήσει η κλιμάκωση. Με αυτόν τον τρόπο διασφαλίζεται ότι η κλιμάκωση είναι λιγότερο πιθανό να διαταράξει τη διαδικασία κυλιόμενης ενημέρωσης.

ReplicaSets: Ένα ReplicaSet είναι ένα αντικείμενο Kubernetes που παρακολουθεί τα Pods που ανήκουν σε μια ομάδα που έχει δηλωθεί στις προδιαγραφές του και προσπαθεί να διασφαλίσει ότι ανά πάσα στιγμή ο επιθυμητός αριθμός αυτών των pods εκτελείται στο cluster. Στην προδιαγραφή ReplicaSet μπορούν να ορισθούν πρόσθετες παράμετροι, για τον καθορισμό της διαδικασίας επιλογής κόμβου. Όταν εφαρμόζονται αλλαγές τότε το ReplicaSet διασφαλίζει ότι τα παλιά pods τερματίζονται και εκτελούνται νέα, που συμμορφώνονται με τις ενημερωμένες προδιαγραφές. Αυτό το αντικείμενο είναι κατάλληλο για στοιχεία εφαρμογής φόρτου εργασίας χωρίς κατάσταση, επειδή όταν μια παρουσία τερματίζεται ή αντικαθίσταται, δεν διατηρούνται πληροφορίες κατάστασης.

StatefulSets: Αρχικά το Kubernetes αναπτύχθηκε με σκοπό να υποστηρίξει κυρίως εφαρμογές serverless. Ωστόσο, καθώς η δημοτικότητά του αυξανόταν και η επιθυμία για υποστήριξη πιο περίπλοκων εφαρμογών αυξήθηκε, ήταν αναπόφευκτο να υποστηριχθούν και άλλοι τύποι φόρτου εργασίας. Τα StatefulSets εκπληρώνουν αυτόν τον σκοπό. Όπως όλα τα προηγούμενα αντικείμενα, τα StatefulSets παρακολουθούν μια ομάδα αντικειμένων και διασφαλίζουν ότι ο επιθυμητός αριθμός εκτελείται στο σύμπλεγμα. Ωστόσο, σε αντίθεση με τις προηγούμενες κατηγορίες, κάθε ομάδα που δημιουργείται ως μέρος ενός StatefulSets έχει μια μόνιμη ταυτότητα που παραμένει όσο ο αριθμός των αντιγράφων του συνόλου δεν έχει αλλάξει. Αυτό σημαίνει ότι ακόμα κι αν ένα pod τερματιστεί ή αποσυρθεί, μετά τη δημιουργία μιας νέας παρουσίας θα διατηρήσει το ίδιο αναγνωριστικό. Επίσης, το pod λαμβάνει ένα μοναδικό και σταθερό αναγνωριστικό δικτύου.

Η διατήρηση της κατάστασης των pod εισάγει επίσης ορισμένους περιορισμούς στην ανάπτυξη και την κλιμάκωση των StatefulSets. Κατά τη διάρκεια της αύξησης ή της μείωσης των pod, μόνο ένας συγκεκριμένος αριθμός Pod μπορεί να δημιουργηθεί ή να τερματιστεί ταυτόχρονα. Από προεπιλογή, δημιουργείται ή τερματίζεται μόνο ένα κάθε φορά, εκτός εάν το StatefulSet χωριστεί σε υποομάδες. Σε αυτήν την περίπτωση δημιουργείται ή τερματίζεται μόνο ένα ανά υποομάδα. Σε εφαρμογές που διατηρούν κατάσταση, κατά τη διάρκεια της αύξησης ή της μείωσης της κλίμακας, ορισμένες πληροφορίες πρέπει να μεταδίδονται μεταξύ των Pods του συνόλου, για να ενημερωθούν οι καταστάσεις τους σχετικά με την αλλαγή που πρόκειται να συμβεί, έτσι ώστε η κατάσταση του φορτίου εργασίας να είναι συνεπής. Για αυτόν τον λόγο, ένα νέο pod που ανήκει στο σύνολο μπορεί να ξεκινήσει μόνο εάν όλοι οι προκάτοχοι βρίσκονται σε κατάσταση λειτουργίας και ένα pod μπορεί να τερματιστεί εάν όλοι οι διάδοχοί του έχουν ήδη τερματιστεί. Οι περιορισμοί που εισάγονται μπορούν να δημιουργήσουν καθυστερήσεις ή ακόμα και να εμποδίσουν τη διαδικασία κλιμάκωσης εάν ένα από τα pod αποτύχει και δεν μπορεί να φτάσει ξανά στην κατάσταση λειτουργίας. Αυτό δημιουργεί την ανάγκη για πιο προηγμένους ελεγκτές που είναι συγκεκριμένοι για την εφαρμογή, για να χειρίζονται ανάλογα εσωτερικά σφάλματα, ώστε τα pods να μπορούν να ανακάμψουν από την αποτυχία.

Τα StatefulSets παρέχουν τη βάση για την εφαρμογή stateful εφαρμογών, αλλά τα pod εξακολουθούν να παραμένουν εφήμερα. Εάν η εφαρμογή απαιτεί τη διατήρηση της κατάστασης και δεν μπορεί ή δεν πρέπει να ανακτηθεί πλήρως από άλλες ομάδες, τότε το StatefulSet πρέπει να χρησιμοποιήσει κάποια μορφή μόνιμης αποθήκευσης για να αποθηκεύσει μόνιμα τις απαραίτητες πληροφορίες.

Cassandra

Η Cassandra είναι ένα κατανεμημένο σύστημα διαχείρισης βάσεων δεδομένων ανοιχτού κώδικα. Έχει σχεδιαστεί για να χειρίζεται πολύ μεγάλες ποσότητες δεδομένων που κατανέμονται σε πολλούς διακομιστές, προσφέροντας υψηλή διαθεσιμότητα και κανένα μοναδικό σημείο αποτυχίας. Είναι μια λύση NoSQL και αναπτύχθηκε αρχικά από το Facebook για να υποστηρίξει τη δυνατότητα Αναζήτησης Εισερχομένων μέχρι το 2010 (Avinash Lakshman, 2014). Τα κύρια χαρακτηριστικά του είναι:

Αποκεντρωμένη:

Κάθε κόμβος στο σύμπλεγμα έχει τον ίδιο ρόλο. Δεν υπάρχει κανένα σημείο αποτυχίας. Τα δεδομένα κατανέμονται σε όλο το σύμπλεγμα (άρα κάθε κόμβος περιέχει διαφορετικά δεδομένα), αλλά δεν υπάρχει κύριος καθώς κάθε κόμβος μπορεί να εξυπηρετήσει οποιοδήποτε αίτημα.

Υποστηρίζει αναπαραγωγή και αναπαραγωγή πολλαπλών κέντρων δεδομένων:

Οι στρατηγικές αναπαραγωγής είναι διαμορφώσιμες. Το Cassandra έχει σχεδιαστεί ως ένα κατανεμημένο σύστημα, για την ανάπτυξη μεγάλου αριθμού κόμβων σε πολλαπλά κέντρα δεδομένων. Τα βασικά χαρακτηριστικά της κατανεμημένης αρχιτεκτονικής της Cassandra είναι ειδικά προσαρμοσμένα για ανάπτυξη κέντρων πολλαπλών δεδομένων, για πλεονασμό, για ανακατεύθυνση και ανάκτηση καταστροφών.

Επεκτασιμότητα:

Η απόδοση ανάγνωσης και εγγραφής αυξάνεται γραμμικά καθώς προστίθενται νέα μηχανήματα, χωρίς διακοπές λειτουργίας ή διακοπές στις εφαρμογές.

Ανεκτικό σε σφάλματα:

Τα δεδομένα αναπαράγονται αυτόματα σε πολλούς κόμβους για ανοχή σφαλμάτων. Υποστηρίζεται η αναπαραγωγή σε πολλά κέντρα δεδομένων. Οι μη αποκρίσιμοι κόμβοι μπορούν να αντικατασταθούν χωρίς χρόνο διακοπής λειτουργίας.

Συντονιζόμενη συνέπεια:

Οι εγγραφές και οι αναγνώσεις προσφέρουν ένα ρυθμίσιμο επίπεδο συνέπειας, από το "writes never fail" έως το "block for all replicas to be readable", με το επίπεδο concensus στη μέση.

Υποστήριξη MapReduce:

Η Cassandra διαθέτει ενσωμάτωση Hadoop, με υποστήριξη MapReduce. Υπάρχει επίσης υποστήριξη για το Apache Pig και το Apache Hive.

Γλώσσα ερωτήματος:

Αν και είναι η βάση δεδομένων NoSQL στον πυρήνα της, εισήχθη η CQL (Cassandra Query Language), μια εναλλακτική λύση που μοιάζει με SQL στην παραδοσιακή διεπαφή RPC.

Μοντέλο Δεδομένων

Στήλη:

Είναι η ατομική μονάδα πληροφοριών στην Cassandra και αναπαρίσταται με τη μορφή όνομα:τιμή.

Ύπερ-στήλη:

Οι υπερ-στλήλες ομαδοποιούν τις στήλες και παρέχουν ένα κοινό όνομα. Με αυτόν τον τρόπο μπορούν να μοντελοποιηθούν πιο σύνθετες δομές δεδομένων μέσα στην Κασσάνδρα.

Σειρές:

Οι σειρές είναι τα μοναδικά αναγνωρίσιμα δεδομένα που αποθηκεύονται στην Κασσάνδρα. Ομαδοποιούν τις τιμές στηλών και υπερστήλων και τις συνδέουν με ένα μοναδικό κλειδί. Η Cassandra εκτελεί ερωτήματα με βάση αυτό το μοναδικό κλειδί.

Οικογένειες στηλών:

Οι οικογένειες στηλών είναι ανάλογες με έναν πίνακα σχεσιακών βάσεων δεδομένων. Ομαδοποιούν σειρές με κλειδί που αποτελούνται από παρόμοιες στήλες και υπερστήλες. Η βασική διαφορά με τις σχεσιακές βάσεις δεδομένων είναι ωστόσο ότι δεν πρόκειται για ισχυρό περιορισμό και ότι το σχήμα των οικογενειών στηλών δεν είναι προκαθορισμένο ή αυστηρά το ίδιο. Ο χρήστης είναι ελεύθερος να συμπεριλάβει όσες στήλες και τιμές υπερστήλων επιθυμεί για κάθε σειρά. Οι οικογένειες στηλών απλώς παρέχουν μια αφαίρεση για να ομαδοποιήσουν σειρές που είναι πιθανό να υποβληθούν σε ερωτήματα μαζί, προκειμένου να βελτιωθεί η απόδοση.

Keyspaces:

Τα Keyspaces είναι το υψηλότερο επίπεδο αναπαράστασης πληροφοριών σε ένα σύμπλεγμα Cassandra. Κάθε οικογένεια στηλών ανήκει ακριβώς σε έναν χώρο κλειδιών. Επίσης, τα επίπεδα συνέπειας και ο παράγοντας αναπαραγωγής ορίζονται μοναδικά για χώρο κλειδιών, πράγμα που σημαίνει ότι διαφορετικοί χώροι κλειδιών μπορούν να έχουν διαφορετικά επίπεδα συνέπειας και στρατηγικές αναπαραγωγής μέσα στο ίδιο σύμπλεγμα.

Αρχιτεκτονική συμπλέγματος

Ένα σύμπλεγμα Cassandra λειτουργεί με μια αρχιτεκτονική Peer to Peer (P2P), που σημαίνει ότι κάθε κόμβος συνδέεται με όλους τους άλλους κόμβους. Επίσης, κάθε κόμβος γνωρίζει την κατανομή δεδομένων σε όλους τους άλλους κόμβους. Ως αποτέλεσμα, κάθε κόμβος είναι ικανός να εξυπηρετεί πελάτες και να εκτελεί όλες τις λειτουργίες της βάσης δεδομένων. Όλα τα παραπάνω συμβάλλουν στο γεγονός ότι δεν υπάρχει κανένα σημείο αστοχίας σε ένα σύμπλεγμα Cassandra και αυτό προάγει την αυξημένη ανοχή σφαλμάτων. Επίσης, η απόδοση κλιμακώνεται σχεδόν γραμμικά με την ποσότητα των διαθέσιμων κόμβων στο σύμπλεγμα.

Το σύμπλεγμα Cassandra είναι επίσης λογικά οργανωμένο σε rack και datacenters. Στα φυσικά κέντρα δεδομένων, ένα rack είναι μια ομάδα φυσικών μηχανημάτων που μοιράζονται πόρους όπως τροφοδοσία, δικτύωση κ.λπ. Όταν πολλά rack είναι διαθέσιμα σε ένα σύμπλεγμα, η Cassandra επιλέγει να διανέμει αντίγραφα δεδομένων σε διαφορετικά rack. Με αυτόν τον τρόπο, εγγυάται υψηλότερη διαθεσιμότητα και ανοχή σφαλμάτων, επειδή είναι βέβαιο ότι θα ερωτώνται περισσότεροι από ένας κόμβοι κάθε φορά και ότι όλες οι πληροφορίες αναπαράγονται σε περισσότερους από έναν φυσικούς κόμβους. Μια ομάδα racks μπορεί να ομαδοποιηθεί για να σχηματίσει ένα κέντρο δεδομένων. Η διαίρεση του συμπλέγματος σε κέντρα δεδομένων συμβάλλει στον μετριασμό των επιπτώσεων διαφορετικών φόρτων εργασίας μεταξύ τους, εκχωρώντας δεδομένα διαφορετικού φόρτου εργασίας σε διαφορετικά κέντρα δεδομένων. Τα κέντρα δεδομένων μπορούν να έχουν διαφορετικά επίπεδα συνέπειας και μπορούν να κλιμακωθούν ανεξάρτητα και ταυτόχρονα, απομονώνοντας περαιτέρω την παρεμβολή απόδοσης.

Η Cassandra προσφέρει ρυθμιζόμενα επίπεδα συνέπειας. Οι επιλογές επιπέδου συνέπειας είναι μια αντιστάθμιση μεταξύ Συνέπειας και Διαθεσιμότητας σύμφωνα με το θεώρημα CAP (Simon, 2000). Όσο υψηλότερα είναι τα επιλεγμένα επίπεδα συνέπειας, τόσο περισσότερες επιβεβαιώσεις αιτημάτων πρέπει να ληφθούν για να ολοκληρωθεί ένα ερώτημα που οδηγεί σε υψηλότερο λανθάνοντα χρόνο και περιορίζει τη διαθεσιμότητα. Γενικά, οι λειτουργίες ανάγνωσης έχουν σημαντικά υψηλότερο λανθάνοντα χρόνο από τις λειτουργίες εγγραφής. Ο λόγος πίσω από αυτό είναι ο τρόπος με τον οποίο υλοποιείται η αποθήκευση δεδομένων.

Ένα τελευταίο σύνολο λειτουργιών που είναι κρίσιμες για τη συνέπεια των δεδομένων είναι οι μηχανισμοί κατά της εντροπίας. Όπως αναφέραμε προηγουμένως, οι αναγνώσεις εκτελούν ορισμένες μερικές λειτουργίες αντι-εντροπίας. Ωστόσο, είναι ευκαιριακές, πράγμα που σημαίνει ότι δεν εκτελούν όλες οι λειτουργίες ανάγνωσης ελέγχους κατά της εντροπίας για να αποφευχθεί η μείωση της απόδοσης. Ένας άλλος μηχανισμός κατά της εντροπίας είναι οι υπονοούμενες μεταβιβάσεις(hinted handoffs). Όταν ένας κόμβος δεν είναι διαθέσιμος για λίγο, όλες οι πληροφορίες αναπαραγωγής που πρέπει να αποθηκευτούν στον κόμβο, αποθηκεύονται σε έναν ομότιμο κόμβο. Όταν ο μη διαθέσιμος κόμβος εισέρχεται ξανά στο σύμπλεγμα, λαμβάνει αυτές τις πληροφορίες (που ονομάζονται "υπαινιγμοί") από τον ομότιμο κόμβο και ανταποκρίνεται στην τελευταία κατάσταση του συμπλέγματος. Προφανώς, υπάρχουν περιορισμοί χρόνου και μεγέθους για υπονοούμενες μεταβιβάσεις και κατά συνέπεια δεν μπορεί να θεωρηθεί πρωταρχικός μηχανισμός κατά της εντροπίας. Το πιο αξιόπιστο εργαλείο για την αντιεντροπία είναι οι επισκευές. Οι επισκευές δημιουργούν ειδικές δομές δεδομένων που ονομάζονται Merkel-trees που

κατακερματίζουν τις υπάρχουσες εγγραφές δεδομένων σε ένα αντίγραφο και στη συνέχεια διανέμονται σε άλλους κόμβους για τον εντοπισμό ασυνεπειών. Τέλος, η τελευταία έκδοση όλων των καταχωρήσεων μεταδίδεται μεταξύ κόμβων. Παρόλο που, μερικές επισκευές εκτελούνται σποραδικά κατά τη διάρκεια περιόδων χαμηλής επισκεψιμότητας, μια πλήρης επισκευή πρέπει να προγραμματιστεί χειροκίνητα, επειδή είναι μια υπολογιστικά εντατική λειτουργία που μπορεί να καταστήσει το σύμπλεγμα προσωρινά μη διαθέσιμο.

K8ssandra

Όταν κυκλοφόρησε αρχικά το Kubernetes, η κύρια εστίασή του ήταν η υποστήριξη stateless εφαρμογών. Αν και αρκετοί οργανισμοί έχουν μετεγκαταστήσει τις εφαρμογές cloud τους στο οικοσύστημα Kubernetes, η αρχικά περιορισμένη υποστήριξη για stateful εφαρμογές είχε ως αποτέλεσμα την ωρίμανση της υπολογιστικής υποδομής με μεγαλύτερο ρυθμό από την υποδομή δεδομένων μέσα σε αυτό το οικοσύστημα. Πριν από το K8ssandra, η πιο κοινή πρακτική ήταν η ανάπτυξη υπολογιστικών πόρων μέσα στο σύμπλεγμα Kubernetes και η ανάπτυξη μιας εξωτερικής βάσης δεδομένων και η σύνδεσή της με το σύμπλεγμα. Αυτή η προσέγγιση δεν είναι η βέλτιστη επειδή απαιτούνται τουλάχιστον 2 στοίβες παρακολούθησης για τη διασφάλιση της ομαλής λειτουργίας της εφαρμογής, η παραγωγικότητα ανάπτυξης είναι περιορισμένη επειδή απαιτείται ικανότητα σε τουλάχιστον δύο περιβάλλοντα και το καθένα μπορεί να αποτελέσει εμπόδιο για το άλλο και τελικά η υποδομή cloud δεν είναι αξιοποιούνται βέλτιστα οδηγώντας σε μεγαλύτερο κόστος (K8ssandra, 2022).

Η προφανής λύση είναι η μεταφορά η της υποδομής δεδομένων μέσα στο σύμπλεγμα. Με αυτόν τον τρόπο η εφαρμογή μπορεί να παρακολουθείται πιο αποτελεσματικά και να κλιμακώνεται με μειωμένο κόστος. Η K8ssandra προσφέρει μια ευέλικτη ανάπτυξη ενός συμπλέγματος Cassandra μέσα στο Kubernetes που είναι κατάλληλο τόσο για προγραμματιστές που αναζητούν μια επεκτάσιμη λύση δεδομένων όσο και για προγραμματιστές που αναζητούν μια απλή και έτοιμη προς εκτέλεση λύση χρησιμοποιώντας προρυθμισμένες παραμέτρους.

Cass Operator: Αρχικά, ο cass-operator ήταν ένα αυτόνομο έργο που χρησιμοποιήθηκε για την ανάπτυξη ενός συμπλέγματος Cassandra μέσα σε ένα σύμπλεγμα Kubernetes. Παρείχε περιορισμένες δυνατότητες παρακολούθησης και επίσης έλεγχε το bootstrapping κόμβων και την κλιμάκωση συμπλέγματος. Λίγο μετά την κυκλοφορία του έργου K8ssandra, ο Cass-operator μεταφέρθηκε μέσα σε αυτό και τώρα η ανάπτυξη και η συντήρησή του έχουν ανατεθεί στην κοινότητα της K8ssandra.

Όσον αφορά τους πόρους του Kubernetes, ο cass-operator είναι ένας προσαρμοσμένος ελεγκτής που παρακολουθεί όλους τους πόρους που χρησιμοποιούνται για την λειτουργία του συμπλέγματος Cassandra. Κατά την εγκατάσταση, λαμβάνει ένα αρχείο yaml που περιέχει παραμέτρους σχετικά με το σύμπλεγμα Cassandra. Αυτή η προσέγγιση επιτρέπει μεγάλη ευελιξία όσον αφορά τη διαμόρφωση, επειδή κάθε στοιχείο του συμπλέγματος Cassandra μπορεί να διαμορφωθεί σε μεγάλο βαθμό. Για παράδειγμα, μπορούμε να ορίσουμε ρητά όρια πόρων για κάθε μεμονωμένο στοιχείο, να διαμορφώσουμε τις συνδέσεις δικτύου μεταξύ των στοιχείων και επίσης να δημιουργήσουμε ρυθμίσεις

απορρήτου για να αυξήσουμε την ασφάλεια του συμπλέγματος. Μπορούμε επίσης να επιλέξουμε να μην εγκαταστήσουμε ένα συγκεκριμένο στοιχείο της προεπιλεγμένης εγκατάστασης του K8ssandra εάν δεν ταιριάζει στις ανάγκες της εφαρμογής μας. Μετά την ανάπτυξη του πόρου, ο τελεστής cass δημιουργεί ένα StatefulSet με μέγεθος ίσο με το δηλωμένο μέγεθος του συμπλέγματος Cassandra μας. Στη συνέχεια παρακολουθεί την εκκίνηση των κόμβων, διασφαλίζοντας ότι μόνο ένας κόμβος εισέρχεται στο σύμπλεγμα κάθε φορά για να διασφαλίσει σταθερή ανάπτυξη. Δεδομένου ότι ο χειριστής γνωρίζει την τοπολογία του δικτύου ως προς τους πόρους του Kubernetes, οργανώνει κόμβους σε rack με τρόπο που διασφαλίζει ότι υπάρχει η ελάχιστη πιθανότητα απώλειας δεδομένων λόγω αποτυχίας κόμβου. Πιο συγκεκριμένα, επιλέγει να οργανώνει κόμβους που τρέχουν στον ίδιο κόμβο Kubernetes στο ίδιο rack. Με αυτόν τον τρόπο, εάν ο κόμβος καταστεί μη διαθέσιμος, τα δεδομένα δεν χάνονται εάν ο συντελεστής αναπαραγωγής είναι μεγαλύτερος από ένα, λόγω του πρωτοκόλλου αναπαραγωγής της Cassandra για αποθήκευση αντιγράφων σε διαφορετικά rack.

Ο Cass-operator δημιουργεί επίσης πολλές υπηρεσίες για να επιτρέπει συνδέσεις δικτύου μεταξύ κόμβων Cassandra. Πρώτα δημιουργεί μια δικτυακή υπηρεσία έτσι ώστε οι κόμβοι που εισέρχονται στο σύμπλεγμα να μπορούν να βρουν seed nodes για να κάνουν bootstrap. Δημιουργεί επίσης μια υπηρεσία all-pod που επιτρέπει στους κόμβους Cassandra να συνδέονται μεταξύ τους μόλις εισέλθουν στο σύμπλεγμα. Και πάλι, τα endpoints ενημερώνονται από τον cass-operator καθώς οι κόμβοι εισέρχονται ή εξέρχονται από το σύμπλεγμα.

Συνολικά, ο τελεστής cass είναι το πιο αξιόπιστο σημείο παρατήρησης του συμπλέγματος Cassandra. Η κατάσταση του συμπλέγματος μπορεί να παρακολουθηθεί από το αρχείο yaml json που μπορεί να παραχθεί από την Kubernetes ανά πάσα στιγμή και περιγράφει την τρέχουσα κατάσταση του πόρου cass-operator και κατά συνέπεια του συμπλέγματος. Από αυτό το μέρος της εγκατάστασης, ο μηχανισμός που δημιουργήσαμε δίνει εντολές για αλλαγές στο σύμπλεγμα και λαμβάνει ενημερώσεις σχετικά με το πότε εφαρμόζονται αυτές οι αλλαγές.

Cassandra Pods: Αποτελούνται από τρία containers. Ένα container Cassandra που εκτελεί τον πηγαίο κώδικα της Cassandra, ένα κοντέινερ καταγραφής που αποθηκεύει πληροφορίες σχετικά με τη λειτουργία της Cassandra εντός του pod και είναι χρήσιμο για την αντιμετώπιση προβλημάτων και ένα init-container που χειρίζεται τις ρυθμίσεις διαμόρφωσης που λαμβάνονται από το Cass-operator όταν το σύμπλεγαμα εκκινείται. Το Όπως αναφέραμε, τα pods Cassandra αποτελούν μέρος ενός StatefulSet που παρακολουθεί ολόκληρο το σύμπλεγμα Cassandra. Με αυτόν τον τρόπο, κάθε pod είναι μοναδικό και αν αποτύχει ένα pod, επανεκκινείται με τα ίδια αναγνωριστικά. Επίσης, δεδομένου ότι υπάρχει η προφανής ανάγκη για μόνιμη αποθήκευση, δημιουργείται ένα PersistentVolumeClaim όταν δημιουργείται το pod που συνδέει το pod σε έναν μόνιμο δίσκο αποθήκευσης που είναι διαθέσιμος στο σύμπλεγμα Kubernetes. Ένα pod μπορεί να ξεκινήσει να εκτελείται μόνο εάν είναι συνδεδεμένος ένας κατάλληλος δίσκος σε αυτό.

Stargate: Μπορούμε να πούμε ότι η εγκατάσταση K8ssandra δεν είναι μια πιστή αντιγραφή της εγκατάστασης Cassandra και τα pods Stargate είναι ο λόγος για αυτό. Το έργο Stargate

είναι μια πύλη δεδομένων ανοιχτού κώδικα που παρέχει ένα επίπεδο αφαίρεσης μεταξύ μιας βάσης δεδομένων και των αιτημάτων πελατών. Με αυτόν τον τρόπο, το αίτημα πελάτη μπορεί να διαμορφωθεί ώστε να ταιριάζει στις ανάγκες της εφαρμογής και στη συνέχεια να μετασχηματιστεί ώστε να ταιριάζει στη διεπαφή της βάσης δεδομένων.

Το κίνητρο πίσω από την ενσωμάτωση του stargate στο έργο K8ssandra οφείλεται σε δύο βασικούς λόγους. Το πρώτο είναι ότι παρόλο που η Cassandra προσφέρει τη διεπαφή CQL για τη δημιουργία ερωτημάτων τύπου SQL, μια τέτοια προσέγγιση θα ήταν κατάλληλη μόνο για εφαρμογές που χρησιμοποιούσαν ήδη μια λύση Cassandra. Στη γενική περίπτωση, ειδικά σε περιβάλλον μικροϋπηρεσιών, η κοινή προσέγγιση είναι ότι διαφορετικά στοιχεία αλληλεπιδρούν χρησιμοποιώντας αιτήματα API. Το Stargate ικανοποιεί αυτή την ανάγκη για έναν πιο ομοιόμορφο τρόπο αλληλεπίδρασης με το σύμπλεγμα Κασσάνδρας.

Ένας άλλος σκοπός που εξυπηρετούν οι κόμβοι Stargate είναι να εκτελέσουν ένα μέρος του συντονισμού των αιτημάτων δεδομένων προς το σύμπλεγμα. Κάθε pod stargate εκτελεί μια serverless εγκατάσταση της Cassandra. Αυτό σημαίνει ότι όταν δημιουργείται το pod, θεωρείται μέρος του δακτυλίου, αλλά δεν λαμβάνει δεδομένα. Λαμβάνει και διατηρεί μόνο πληροφορίες διανομής των κλειδιών και της τοπολογίας, έτσι ώστε να γνωρίζει τη κατανομή των δεδομένων. Όταν φτάνουν αιτήματα, είναι υπεύθυνο να τα προωθήσει στους κατάλληλους κόμβους Cassandra και επίσης να επαληθεύσει τις επιστρεφόμενες απαντήσεις σύμφωνα με τα πρωτόκολλα συνέπειας της Cassandra. Με αυτόν τον τρόπο, τα pods Stargate μετριάζουν κάποια από την υπολογιστική πίεση από την Cassandra, καθώς οι υπολογισμοί για την εξυπηρέτηση των αιτημάτων διαχωρίζονται από τους υπολογισμούς για την ανάκτηση δεδομένων. Τα δύο μέρη των συστάδων μπορούν να κλιμακωθούν ανεξάρτητα.

Αν και το Stargate θεωρείται βασικό στοιχείο του συμπλέγματος K8ssandra και έχει διαμορφωθεί για βέλτιστη απόδοση, δεν είναι υποχρεωτικό να συμπεριληφθεί στην εγκατάσταση. Μια εντελώς βιώσιμη εναλλακτική είναι να απενεργοποιηθούν τα pods Stargate κατά την εγκατάσταση και να εκτελεστούν μόνο τα pods Cassandra. Σε αυτήν την περίπτωση, ωστόσο, ο χρήστης πρέπει να δημιουργήσει χειροκίνητα μια πρόσθετη υπηρεσία για να εκθέσει την Cassandra στο υπόλοιπο σύμπλεγμα ή έξω από το σύμπλεγμα, ανάλογα με τις ανάγκες.

Ενισχυτική μάθηση

Τυπικά μια διαδικασία ενισχυτικής μάθησης ορίζεται από:

- Ένα σύνολο καταστάσεων S
- Ένα σύνολο κινήσεων A
- Μια συνάρτηση αντμοιβής $R : S \times A \rightarrow \mathbb{R}$
- Μια συνάρτηση μετάβασης $S \times A \times S \rightarrow \Pi(S)$

Επίσης ορίζουμε βέλτιστη τιμή μια κατάστασης την μέση αναμενόμενη απομειωμένη ανταμοιβή που θα λάβει το μοντέλο αν ξεκινήσει από την κατάσταση s και εκτελέσει την κίνηση α και από εκεί και πέρα λάβει αποφάσεις με τη βέλτιστη πολιτική λήψης αποφάσεων.

$$V^*_{(s)} = \max E\left(\sum_{t=0}^{\infty} \gamma^t r_t\right)$$

Q Learning:

Η μέθοδος του Q Learning είναι μια ευρέως χρησιμοποιούμενη μέθοδος ενισχυτικής μάθησης λόγω της ευκολίας υλοποίησής της σε σχέση με άλλες μεθόδους. Για την εφαρμογή της ορίζεται ως συνάρτηση Q^* των καταστάσεων και των κινήσεων. Το Q αναπαριστά τη μέση απομειούμενη ανταμοιβή αν εκτελεστεί η κίνηση α στη κατάσταση s και στη συνέχεια ληφθούν βέλτιστες αποφάσεις στις επόμενες καταστάσεις

Με βάση τα παραπάνω και με την προϋπόθεση ότι εκτελούμε βέλτιστες αποφάσεις από το πρώτο βήμα η Q συνάρτηση ορίζεται αναδρομικά:

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s \in S} T(s,a,s') \max_{a'} Q^*(s',a')$$

$Q^*(s,a)$ μας υπολογίζει και τη βέλτιστη πολιτική $\pi^*_{(s)}$ ως $\pi^*_{(s)} = \arg\max Q^*(s,a)$. Η αναδρομική φύση της συνάρτησης και η δυνατότητα να υπολογίζεται μονοσήμαντα σε κάθε βήμα η βέλτιστη κίνηση επιτρέπει να υπολογίζονται εκτιμήσει για τις τιμές και ταυτόχρονα να της χρησιμοποιούμε για την εξαγωγή της βέλτιστης πολιτικής. Ο κανόνας μάθησης των τιμών της συνάρτησης είναι:

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$$

όπου $\langle s,a,r,s' \rangle$ είναι μία εμπειρία.

Βαθιά ενισχυτική μάθηση:

Deep Q Learning

Όπως αναφέραμε στο κεφάλαιο του Q Learning, αυτός ο αλγόριθμος επιχειρεί να βρει την ακολουθία ενεργειών που μεγιστοποιούν τις συνολικές ανταμοιβές με έκπτωση προσπαθώντας να υπολογίσει το Q για κάθε δεδομένο ζεύγος κατάστασης-δράσης και στη συνέχεια να κάνει άπληστες επιλογές επιλέγοντας την ενέργεια που έχει την υψηλότερη τιμή Q κάθε φορά. Είναι προφανές ότι όσο καλύτερη είναι η εκτίμηση των τιμών Q, τόσο πιο κοντά στη βέλτιστη θα είναι η ακολουθία ενεργειών. Η προσπάθεια επίλυσης ενός προβλήματος χρησιμοποιώντας τον ορισμό του Q Learning είναι αναποτελεσματική, επειδή σύμφωνα με τον αρχικό αλγόριθμο κάθε ακολουθία αξιολογείται ανεξάρτητα και δεν μπορεί να πραγματοποιηθεί καμία μορφή γενίκευσης. Για να υπάρξει γενίκευση, χρησιμοποιείται μια παραμετροποιημένη συνάρτηση Q Q(s,a;θ)≈ Q^* (s,a), όπου θ είναι ένα σύνολο εκπαιδεύσιμων παραμέτρων. Στο Deep Q Learning, ο εκτιμητής της συνάρητσης είναι ένα νευρωνικό δίκτυο που προσπαθεί να εκτιμήσει τις βέλτιστες τιμές Q για όλες τις ακολουθίες ενεργειών χρησιμοποιώντας την ίδια συνάρτηση.

Τα μοντέλα Deep Q Learning εκπαιδεύονται χρησιμοποιώντας εμπειρίες 〈s,a,r,s'〉 (κατάσταση, δράση, ανταμοιβή, επόμενη κατάσταση). Η διαφορά στην υλοποίηση σε σύγκριση με τον αρχικό αλγόριθμο και άλλες υλοποιήσεις Ενισχυτικής Μάθησης είναι ότι οι πλειάδες δεν χρημοποιούνται για εκπαίδευση με την ίδια σειρά που τις παρατηρεί ο

πράκτορας. Στο DQL χρησιμοποιείται μια δομή γνωστή ως replay memory. Είναι μια προσωρινή μνήμη σταθερού μεγέθους που αποθηκεύει τις N πιο πρόσφατες πλειάδες που παρατηρήθηκαν από το μοντέλο. Σε κάθε βήμα πρώτα γίνεται παρατήρηση της κατάστασης. Στη συνέχεια επιλέγεται μια ενέργεια με βάση την παρατηρούμενη κατάσταση. Τέλος, παρατηρείται η κατάσταση στην οποία οδηγεί η δράση και υπολογίζεται η ανταμοιβή. Η νέα εμπειρία αποθηκεύεται στη μνήμη και στη συνέχεια επιλέγονται m τυχαίες πλειάδες από τη μνήμη για την εκπαίδευση του πράκτορα. Στη συνέχεια η διαδικασία επαναλαμβάνεται.

Η χρήση μιας μνήμης επανάληψης αντί της απλής χρήσης των δειγμάτων καθώς λαμβάνονται για εκπαίδευση παρέχει τρία σημαντικά πλεονεκτήματα. Το πρώτο είναι ότι αυτή η μέθοδος εκπαίδευσης οδηγεί σε πιο αποτελεσματική χρήση του δείγματος. Κάθε δείγμα χρησιμοποιείται σχεδόν σίγουρα περισσότερες από μία φορές για ενημερώσεις βάρους. Σε συνδυασμό με το γεγονός ότι τα νευρωνικά δίκτυα απαιτούν μικρούς ρυθμούς μάθησης για να συγκλίνουν, είναι πολύ πιθανό ότι η χρήση μιας πλειάδας εκπαίδευσης μόνο μία φορά για την ενημέρωση του βάρους του δικτύου δεν αρκεί για να αποκτηθούν όλες οι πιθανές πληροφορίες από αυτό το δείγμα εκπαίδευσης. Το δεύτερο πλεονέκτημα είναι ότι αυτή η τεχνική εκπαίδευσης σπάει τους ισχυρούς συσχετισμούς μεταξύ διαδοχικών δειγμάτων. Αυτό είναι σημαντικό επειδή τα αναδιαταγμένα δείγματα μειώνουν τη διακύμανση των ενημερώσεων, γεγονός που οδηγεί σε ταχύτερη σύγκλιση. Επιπλέον, η εκμάθηση σχετικά με την πολιτική είναι επιρρεπής να κολλήσει στα τοπικά ελάχιστα. Ο λόγος είναι ότι σε κάθε βήμα το δίκτυο κάνει μια επιλογή με βάση τις παραμέτρους του και μετά εκπαιδεύεται με βάση την επιλογή που έκανε. Εάν η επιλογή είναι τοπικά βέλτιστη, το δίκτυο πρόκειται να επαναλάβει την επιλογή και να αγνοήσει άλλες επιλογές που δυνητικά οδηγούν σε μεγαλύτερη συνολική ανταμοιβή (Martin, 2013).

Double Deep Q Learning

Ο αρχικός αλγόριθμος Q Learning είναι γνωστό ότι υπερεκτιμά τις τιμές Q του μοντέλου. Αυτό το φαινόμενο δεν είναι απαραίτητα επιβλαβές για την απόδοση του αλγορίθμου και την πολιτική που θα έχει ως αποτέλεσμα. Μια κοινή τεχνική εξερεύνησης που ονομάζεται αισιοδοξία μπροστά στην αβεβαιότητα, βασίζεται σε αυτήν την ιδέα. Σε κάθε ανεξερεύνητη τιμή Q εκχωρείται μια υψηλή αριθμητική τιμή, έτσι ώστε ο αλγόριθμος να έχει κίνητρα να εξερευνήσει επαρκώς τον χώρο κατάστασης πριν κάνει άπληστες τοπικά βέλτιστες ενέργειες. Επιπλέον, εάν η υπερεκτίμηση των τιμών είναι ομοιόμορφη, τότε διατηρείται η δυναμική των προτιμήσεων δράσης, οδηγώντας στη βέλτιστη πολιτική. Στην εργασία τους ωστόσο, οι Hado et al. (Hado van Hasselt, 2016) ισχυρίζονται ότι σε αρκετές εφαρμογές του DQN η υπερεκτίμηση δεν είναι ομοιόμορφη και όντως βλάπτει την απόδοση.

Στην εργασία τους, προτείνουν μια προσαρμογή στο αρχικό μοντέλο DQN, όπου ένα δεύτερο μοντέλο προστίθεται στον πράκτορα. Η υλοποίησή τους ονομάζεται Double Deep Q Learning. Η ιδέα είναι ότι ένας από τους κύριους λόγους υπερεκτίμησης είναι το γεγονός ότι το ίδιο δίκτυο χρησιμοποιείται για την επιλογή της βέλτιστης ενέργειας και στη συνέχεια για την αξιολόγηση της αξίας της. Αυτό δημιουργεί έναν ανεπιθύμητο βρόχο ανάδρασης, όπου μια ελαφρώς αυξημένη τιμή Q είναι πιο πιθανό να επιλεγεί ξανά στο μέλλον χωρίς να είναι η βέλτιστη λύση στην πραγματικότητα, αυξάνοντας τη μακροπρόθεσμη υπερεκτίμηση. Αυτό το πρόβλημα είναι πιο πιθανό να παρουσιαστεί όταν ο προσεγγιστής είναι ένα νευρωνικό δίκτυο επειδή στα αρχικά στάδια της εκπαίδευσης οι τιμές Q είναι αυθαίρετες

λόγω της τυχαιοποιημένης αρχικοποίησης βάρους. Ακόμη και μικρές αρχικές υπερεκτιμήσεις, μπορεί να οδηγήσουν σε αισθητή μακροπρόθεσμη απόκλιση από τη βέλτιστη πολιτική (Hado van Hasselt, 2016).

Για να μετριαστεί η υπερεκτίμηση, οι συγγραφείς προτείνουν την αποσύνδεση της επιλογής δράσης από την αξιολόγηση της. Τα δύο δίκτυα του μοντέλους καλούνται online και target networks αντίστοιχα. Το νευρωνικό ενημερώνεται κανονικά χρησιμοποιώντας τα δείγματα εκπαίδευσης. Το target network είναι ένα αντίγραφο του διαδικτυακού δικτύου με καθυστέρηση, που σημαίνει ότι κάθε N βήματα, οι παράμετροι του διαδικτυακού δικτύου αντιγράφονται στο δίκτυο προορισμού. Η άπληστη πολιτική ή η επιλογή δράσης αξιολογείται χρησιμοποιώντας το online network. Στη συνέχεια, το target network χρησιμοποιείται για την εκτίμηση της τιμής Q του ζεύγους ενεργειών κατάστασης και η ενημέρωση βάρους του διαδικτυακού δικτύου πραγματοποιείται χρησιμοποιώντας αυτήν την εκτίμηση. Ο κανόνας εκτίμησης Q για το DDQN φαίνεται παρακάτω:

$$Y^{DDQN} = R_{t+1} + \gamma Q\left(S_{t+1}, arg\max_{a} Q(S_t, a; \theta_t); {\theta'}_t\right)$$
$$\text{όπου } \theta_t \text{ οι παράμετροι του online network}$$
$${\theta'}_t \text{ οι παράμετροι του target network}$$

Return based scaling

Τα ζητήματα κανονικοποίησης στα μοντέλα Ενισχυτικής Εκμάθησης είναι μια κουραστική εργασία αλλά και απαραίτητη, διότι όταν οι κλίμακες των σφαλμάτων ποικίλλουν σε διαφορετικά στάδια εκπαίδευσης, μπορεί να εμποδίσει ή να εμποδίσει τη σύγκλιση του μοντέλου. Ειδικά σε αλγόριθμους χωρίς μοντέλο, όπου ο πράκτορας πρέπει να εκτιμήσει με ακρίβεια τις τιμές της συνάρτησης Q που περιγράφουν την υποκείμενη δυναμική του προβλήματος, τα ζητήματα κανονικοποίησης είναι ακόμη πιο σοβαρά.

Υπάρχουν διάφοροι παράγοντες που μπορούν να επηρεάσουν τις κλίμακες των σφαλμάτων κατά την εκπαίδευση και καθένας από αυτούς μπορεί να είναι επιζήμιος για τη σύγκλιση του μοντέλου. Η πιο κοινή είναι η συνάρτηση ανταμοιβής. Κάθε τιμή Q είναι το απομειούμενο άθροισμα των τρεχουσών και μελλοντικών ανταμοιβών που το μοντέλο αναμένει να συγκεντρώσει. Όσο μεγαλύτερη είναι η διακύμανση των ανταμοιβών, τόσο μεγαλύτερη μπορεί να είναι η διαφορά των τιμών Q κατά τη διάρκεια της εκπαίδευσης. Αυτό μπορεί να οδηγήσει σε κλίμακες σφάλματος που ποικίλλουν σε πολλές τάξεις μεγέθους λόγω της αθροιστικής φύσης των τιμών Q. Ακόμη και αν η συνάρτηση ανταμοιβής δεν εμφανίζει υψηλή διακύμανση, είναι πιθανό κατά τη διάρκεια μιας ενημέρωσης η εκτιμώμενη και παρατηρούμενη τιμή Q να ποικίλλει πολύ, οδηγώντας σε υψηλή αριθμητική τιμή σφάλματος. Η χρήση μιας τέτοιας τιμής για μια ενημέρωση μπορεί να διαταράξει τη σύγκλιση των βαρών επειδή τα νευρωνικά δίκτυα είναι προσεγγιστές ομαλών συναρτήσεων. Αυτό το φαινόμενο είναι πιο πιθανό να συμβεί κατά τα πρώτα στάδια της εκπαίδευσης, όπου ο πράκτορας εξερευνά τον χώρο κατάστασης και είναι πιθανό ότι δεν είχε ενεργήσει βέλτιστα γύρω από ένα συγκεκριμένο μέρος του χώρου κατάστασης μέχρι εκείνο το σημείο. Είναι επίσης πιθανό να συμβεί όταν κάποια στιγμή στην εκπαίδευση ο πράκτορας ανακαλύπτει νέες δυνατότητες για υψηλότερες ανταμοιβές που έγιναν διαθέσιμες αφού η πολιτική άρχισε να αλλάζει λόγω της εκπαίδευσης. Τέλος, ο συντελεστής έκπτωσης επηρεάζει επίσης σε μεγάλο βαθμό τις αριθμητικές τιμές των τιμών Q.

Για την αντιμετώπιση αυτού του προβλήματος έχει προταθεί η εξής μέθοδος. Προτείνεται ο υπολογισμός ενός συντελεστή δ με τον οποίο κανονικοποιείται το σφάλμα της εκπαίδευσης σε κάθε στάδιο εκμάθησης. Ο συντελεστής υπολογίζεται ως εξής:

$\delta = R_t + \gamma V'_{t+1} - V_t$
$\text{όπου } R_t \text{ είναι η ανταμοιβή στο βήμα } t$
$\text{και } V_t \text{ η εκτιμώμενη } Q \text{ value στο βήμα } t$

Για να προσεγγίσουμε το δ πρέπει να υπολογίσουμε το $V[\delta]$. Στα πρώτα στάδια εκπαίδευσης τα μοντέλα εκτελούνται τυχαίες κινήσεις οπότε το Q μπορεί να θεωρηθεί ανεξάρτητο του R. Υπό αυτή την υπόθεσή το $V[\delta]$ γράφεται:

$V[\delta] = V[R] + V[\gamma(V' - V)] + V[(1-\gamma)V]$
$= V[R] + \gamma^2 V[V' - V] + V[\gamma]E[(V' - V)^2] + (1-\gamma^2)V[V] + V[\gamma]E[V^2]$

Σε κάθε βήμα οι τιμές υπολογίζονται ως $Q = R + \gamma Q'$ και τα Q εκτιμούν το κέρδος του μοντέλου. Μπορούμε λοιπόν να αντικαταστήσουμε V με G και για την απόκλιση ενός βήματος G' – G = R + (1 − γ)G. Επίσης $G = \sum_t \gamma^t R_t$ so $E[G] \approx (1-\gamma)E[R]$. Οπότε:

$V[\delta] = V[R] + (1-\gamma)^2 V[G] + V[\gamma]E[G^2]$

Ο όρος $(1-\gamma)^2 V[G]$ μπορεί να αγνοηθεί γιατί είναι αμελητέος σε σχέση με τους άλλους 2. Όταν το γ είανι σταθερό ή δεν χρησιμοποιούμε επεισοδιακή εκπαίδευση ο όρος $V[\gamma]E[G^2]$ αγνοείται οπότε $V[\delta] = V[R]$. Ο συντελεστής κανονικοποίησης είναι σ όπου $\sigma^2 = V[\delta]$.

Ασύγχρονη Ενισχυτική Μάθηση

Η ασύγχρονη ενισχυτική μάθηση είναι ένα προσοδοφόρο ερευνητικό πεδίο που τραβάει την προσοχή τα τελευταία χρόνια. Ο λόγος είναι ότι θεωρητικά, μπορεί να αξιοποιήσει τα τεράστια σύνολα δεδομένων που υπάρχουν και να εκπαιδεύσει αποτελεσματικά μοντέλα που βασίζονται σε αυτά τα στατικά σύνολα δεδομένων χωρίς περαιτέρω αλληλεπίδραση. Επί του παρόντος, η ενισχυτική μάθηση είναι μια ενεργή διαδικασία μάθησης, όπου το μοντέλο εκτελεί μια ενέργεια παρατηρεί τα αποτελέσματα και στη συνέχεια επαναλαμβάνει. Αυτή η προσέγγιση έχει περιορισμένη εφαρμογή επειδή πρώτα απ 'όλα οι ποσότητες δεδομένων που μπορούν να δημιουργηθούν είναι περιορισμένες σε σύγκριση με την ασύγχρονη εκπαίδευση. Εκτός από τους περιορισμούς των δεδομένων, οι αλληλεπιδράσεις με το περιβάλλον μπορεί να είναι δαπανηρές ή/και καταστροφικές σε πολλές εφαρμογές όπως η ρομποτική ή οι ιατρικές εφαρμογές. Για τους λόγους που αναφέρθηκαν παραπάνω, η αποτελεσματική εφαρμογή ασύγχρονης ενισχυτικής μάθησης είναι μια βασική πρόκληση για την υιοθέτηση της ενισχυτικής μάθησης σε ρεαλιστικά περιβάλλοντα.

Ο κύριος λόγος περιορισμένης απόδοσης αυτών των μοντέλων είναι οι υπερπροσαρμογές και οι ενέργειες εκτός κατανομής (OOD). Αυτά τα προβλήματα συνήθως εκδηλώνονται ως εσφαλμένες υπερεκτιμήσεις της συνάρτησης αξίας σε ορισμένες καταστάσεις. Πιο συγκεκριμένα, το πρόβλημα έγκειται στο γεγονός ότι ο αλγόριθμος βελτιστοποίησης Bellman προσπαθεί να εκτελέσει ενέργειες από την πολιτική εκμάθησης που δημιουργείται καθώς εκπαιδεύεται το μοντέλο, αλλά οι τιμές Q μπορούν να εκπαιδευτούν μόνο σε τιμές που λήφθηκαν από την πολιτική που δημιούργησε το σύνολο δεδομένων εκτός σύνδεσης. Δεδομένου ότι ο αλγόριθμος έχει δημιουργηθεί για να χρησιμοποιεί την πολιτική που έχει μάθει, συχνά οδηγεί σε ενέργειες OOD. Όταν αυτές οι ενέργειες έχουν λανθασμένα υψηλές

ανταμοιβές, οδηγούν σε υπερεκτιμήσεις. Τυπικές εφαρμογές Ενίσχυσης Εκμάθησης εκτός σύνδεσης μετριάζουν αυτό το αποτέλεσμα περιορίζοντας τον αλγόριθμο από το να επιλέξει μη παρατηρούμενες καταστάσεις. Ωστόσο, αυτές οι προσπάθειες καταλήγουν σε υπερβολικά περιοριστικές πολιτικές που περιορίζουν την απόδοση του μοντέλου.

Συντηρητικό Q Learning:

Ο στόχος του είναι ο υπολογισμός του $V^{\pi}(s)$ μιας πολιτικής $\pi(s)$ δεδομένου στατικού συνόλου δεδομένων D που παράγεται από πολιτική $\pi_{\beta}(a|s)$. Για να γίνει συντηρητική η εκμάθηση, προστίθεται ένας επιπλέον όρος δίπλα από την συνάρτηση κόστους Bellman. Η ιδέα πίσω από αυτόν τον όρο είναι ότι καθώς αποκλίνουμε από την αρχική κατανομή των κινήσεων μειώνεται η βεβαιότητά μας για την εκτίμηση των τιμών και ο επιπλέον όρος δρα ως τιμωρία αυτών των κινήσεων. Η αντικειμενική συνάρτηση λοιπόν ορίζεται ως:

$$Q^{k+1} = \min_{Q}\ a \cdot \left(E_{s\sim D, a\sim\mu(a|S)}[Q(s,a)] - E_{s\sim D, a\sim\pi_{\beta}(a|S)}[Q(s,a)]\right) + \frac{1}{2}L \qquad (1)$$

όπου L είναι η συνάρτηση κόστους Bellman και $\mu(a|s)$ η κατανομή action-states τη στιγμή της εκπαίδευσης. Οι συγγραφείς που προτείνουν τον αλγόριθμο αποδεικνύουν ότι για $\mu = \pi$ οι τιμές Q ικανοποιούν τη σχέση

$$\hat{V}^{\pi}(s) < V^{\pi}(s) \forall s \in D$$

Άρα οι εκτιμώμενες τιμές φράζονται από τις πραγματικές άρα επιλύεται το πρόβλημα της υπερεκτίμησης.

Παρατηρώντας το (1) είναι προφανές ότι η ελαχιστοποίηση περιλαμβάνει εκ των προτέρων γνώση της κατανομής μ(α|s). Ωστόσο, το μ είναι μέρος της εκπαίδευσης και μετά από επαρκή αριθμό βημάτων θέλουμε μ = π. Εφόσον το μ είναι μέρος του προβλήματος βελτιστοποίησης, μπορούμε να συμπεριλάβουμε μια μεγιστοποίηση έναντι του μ στον όρο συντηρητικής μάθησης έτσι ώστε σε κάθε επανάληψη η αντικειμενική συνάρτηση να είναι:

$$Q^{k+1} = \min_{Q}\ \max_{\mu}\ a \cdot \left(E_{s\sim D, a\sim\mu(a|S)}[Q(s,a)] - E_{s\sim D, a\sim\pi_{\beta}(a|S)}[Q(s,a)]\right) + \frac{1}{2}L + R(\mu)$$

όπου το R(μ) είναι όρος κανονικοποίησης. Μια λογική επιλογή του R(μ) είναι η απόκλιση Kullback-Liebler (KL). KL-απόκλιση D_KL (P||Q) είναι ένας τύπος στατιστικής απόστασης που εκφράζει την πρόσθετη έκπληξη ή αβεβαιότητα που εισάγεται λόγω της επιλογής μας να χρησιμοποιήσουμε ως μοντέλο μια κατανομή Q όταν η πραγματική κατανομή είναι P. Στην περίπτωσή μας P=μ και Το Q είναι μια προηγούμενη από κοινού κατανομή state-action. Όταν η κατανομή των ενεργειών είναι σχεδόν ομοιόμορφη σε κάθε κατάσταση, τότε η μεγιστοποίηση έναντι του μ έχει ως αποτέλεσμα ένα soft-max των τιμών Q σε οποιαδήποτε δεδομένη κατάσταση και η αντικειμενική συνάρτηση μετατρέπεται σε:

$$Q^{k+1} = \min_{Q}\ a \cdot E_{s\sim D}\left(log \sum_{a} \exp\left(Q(s,a)\right) - E_{s\sim D, a\sim\pi_{\beta}(a|S)}[Q(s,a)]\right) + \frac{1}{2}L \quad (2)$$

Μετατρέπονας τη (2) σε συνάρτηση κόστους μπορούμε να υπολογίσουμε το σφάλαμα για την ενημέρωση των βαρών του δικτύου.

$$L = a \cdot E_{s\sim D}\left(log\sum_a \exp\left(Q(s,a)\right) - E_{s\sim D, a\sim\pi_\beta(a|S)}[Q(s,a)]\right) + \frac{1}{2}L$$

1.5 Πειραματικά αποτελέσματα

Περιγραφή της Υλοποίησης

Σε αυτή την ενότητα περιγράφουμε εν συντομία τον τρόπο με τον οποίο συντονίσαμε τα διάφορα στοιχεία που χρησιμοποιήθηκαν για την εκτέλεση των πειραμάτων μας. Χρησιμοποιήσαμε μια εγκατάσταση της K8ssandra που εκτελείται μέσα σε ένα κατανεμημένο σύμπλεγμα Kubernetes. Οι πελάτες που δημιούργησαν τον φορτίο εργασίας έτρεξαν την υπηρεσία YCSB και ένα απομακρυσμένο script παρακολουθούσε τον αριθμό και τη φύση των αιτημάτων. Για τη συλλογή μετρήσεων, χρησιμοποιήσαμε μία βάση Prometheus που εγκαταστάθηκε και μέσα στο σύμπλεγμα Kubernetes. Τα VM φιλοξενήθηκαν στο περιβάλλον cloud του Okeanos.

Το σύμπλεγμα Kubernetes αποτελείται από 10 VMs, ένα από αυτά λειτουργούσε ως master node και οι υπόλοιποι ως worker nodes. Καθένας από τους κόμβους εργασίας είχε 4 GB μνήμης RAM, 30 GB αποθηκευτικού χώρου και 2 εικονικούς πυρήνες CPU. Ο κύριος κόμβος είχε 8 GB μνήμης RAM, 30 GB αποθηκευτικού χώρου και 4 εικονικούς πυρήνες CPU. Από τα 30 GB διαθέσιμου αποθηκευτικού χώρου κάθε κόμβου, τα 15 GB διατέθηκαν ως εικονικός δίσκος και παραχωρήθηκαν στο σύμπλεγμα Kubernetes. Κάθε worker node έτρεχε ένα pod από το Daemonset του static provisioner του Kubernetes. Ο ρόλος του είναι να διαχειρίζεται τον κύκλο ζωής του PersistentVolume για υπάρχροντες δίσκους, ανιχνεύοντας και δημιουργώντας PersistentVolumes για κάθε τοπικό δίσκο στον κεντρικό υπολογιστή και καθαρίζοντας τους δίσκους όταν απελευθερωθούν. Κάθε κόμβος K8ssandra χρειάζεται τουλάχιστον 2 GB RAM για να λειτουργεί χωρί σφάλματα. Για αυτόν τον λόγο, μόνο ένας κόμβος K8ssandra μπορούσε να εκτελείται κάθε φορά ανά κόμβο εργαζομένου. Τα όρια πόρων που απέδωσαν καλύτερα στις ρυθμίσεις μας ήταν η χρήση 2 GB μνήμης RAM και 1 πυρήνα CPU ανά κόμβο K8ssandra. Επιτρέψαμε επίσης στους κόμβους να υπερβούν τη χρήση RAM μέχρι ένα περιθώριο 0,5 GB. Αυτό επέτρεψε στο σύμπλεγμα K8ssandra να εκτελεί λειτουργίες κλιμάκωσης ακόμη και κάτω από την υψηλή κίνηση και υψηλό ποσοστό χρήσιμοποίησης πόρων. Τέλος, εγκαταστήσαμε 3 pod Stargate για τον συντονισμό των εισερχόμενων αιτημάτων.

Αποτελέσματα

Για την εκπαίδευση των μοντέλων χρησιμοποιήσαμε 17 παραμέτρους για την περιγραφή της κατάστασης του συμπλέγματος:

- Το μέγεθος του συπλέγματος
- Το μέσο latency για το 98% των αιτημάτων
- Το μέσο latency για το 99% των αιτημάτων
- Το μέσο latency για το 999% των αιτημάτων
- Το μέσο ρυθμό εξυπηρέτησης των αιτημάτων
- Το μέσο ρυθμό εξυπηρέτησης των αιτημάτων στο προηγούμενο βήμα
- Τη συνολική ελεύθερη μνήμη ως ποσοστό της συνολικής μνήμης του συμπλέγματος
- Τη συνολική cached μνήμη ως ποσοστό της συνολικής μνήμης

- Τη μέση χρησιμοποιούμενη CPU από το σύμπλεγμα
- Την ελάχιστη χρησιμοποιούμενη CPU από κάποιον κόμβο του συμπλέγματος
- Τη μέγιστη χρησιμοποιούμενη CPU από κάποιον κόμβο του συμπλέγματος
- Τη μέση άεργη CPU του συμπλέγματος
- Το μέσο χρόνο που η CPU ήταν σε αναμονή για αιτήματα I/O.
- Το μέσο αριθμό IOPS στο σύμπλεγμα
- Το μέσο ρυθμό εξυπηρέτησης αιτημάτων ανάγνωσης στο δίσκο
- Το μέσο ρυθμό εξυπηρέτησης αιτημάτων εγγραφής στο δίσκο
- Το ποσοστό των αιτημάτων ανάγνωσης

Η συνάρτηση ανταμοιβής που χρησιμοποιήθηκε είναι:

$$R = 0.01 * throughput - (vms - B) \text{ όπου } B \text{ το ελάχιστο μέγεθος το συμπλέγματος}$$

Σχετικά με την αρχιτεκτονική του σύγχρονου μοντέλου χρησιμοποιήσαμε ένα πλήρως συνδεδεμένο δίκτυο με 2 κρυφά στρώματα νευρώνων. Το πρώτο κρυφό στρώμα αποτελούνταν από 48 κόμβους και το δεύτερο στρώμα αποτελούνταν από 24 κόμβους. Ο replay memory buffer ορίστηκε να αποθηκεύει τις τελευταίες 300 εμπειρίες και τα βάρη ενημερώθηκαν με ρυθμό εκμάθησης a=0,001. Ο συντελεστής έκπτωσης ορίστηκε σε γ=0,99. Για να αποφύγουμε ενημερώσεις βάρους που θα μπορούσαν να οδηγήσουν σε απόκλιση του συστήματος, χρησιμοποιήσαμε την τεχνική return based scaling που περιεγράφηκε στο προηγούμενο κεφάλαιο για να ομαλοποιήσουμε την απώλεια σε κάθε βήμα ενημέρωσης σύμφωνα με την τρέχουσα διακύμανση των ανταμοιβών των προηγούμενων εμπειριών. Η εκπαίδευση ξεκινά με μια μνήμη επανάληψης 300 τυχαίων εμπειριών. Στη συνέχεια, ο παράγοντας εκτελεί 500 βήματα ανόπτησης, με το έψιλον να μειώνεται από το 1 στο 0,1 γραμμικά κατά τη διάρκεια των 500 βημάτων. Διατηρούμε μια μικρή αξία έψιλον στην υπόλοιπη εκπαίδευση για να διατηρήσουμε τη δυνατότητα του πράκτορα να εξερευνήσει υψηλότερες καταστάσεις ανταμοιβής σε μεταγενέστερα στάδια της εκπαίδευσης. Το μοντέλο σε κάθε βήμα παρατηρεί την τρέχουσα κατάσταση του συμπλέγματος και επιλέγει ανάμεσα σε 3 ενέργειες. Αύξηση του μεγέθους του συμπλέγματος κατά 1, μείωση του μεγέθους του συμπλέγματος κατά 1 ή καμία αλλαγή. Στις περιπτώσεις που άλλαζε το μέγεθος του συμπλέγματος, ο πράκτορας παρατηρούσε περιοδικά την κατάσταση του συμπλέγματος όπως περιγράφεται από τη μεταβλητή status.cassandraOperatorProgress. Όταν η τιμή αυτής της μεταβλητής ορίστηκε από "Ενημέρωση" σε "Έτοιμο", ο πράκτορας περίμενε για 5 λεπτά και στη συνέχεια σύλλεγε τις μετρήσεις από τον Prometheus για να εκτελέσει την επόμενη ενέργεια. Όταν το σύμπλεγμα παρέμεινε αμετάβλητο, ο πράκτορας περίμενε 2 λεπτά και στη συνέχεια συγκέντρωσε τις μετρήσεις για να εκτελέσει την επόμενη ενέργεια. Το batch sized για την εκπαίδευση μετά από κάθε απόφαση ορίστηκε σε 32 τυχαία δείγματα αναμνήσεων από το buffer επανάληψης.

Για την αρχιτεκτονική δικτύου του πράκτορα εκτός σύνδεσης, χρησιμοποιήσαμε ξανά ένα πλήρως συνδεδεμένο δίκτυο με 2 κρυφά επίπεδα. Δεδομένου του γεγονότος ότι είχαμε πρόσβαση σε ένα σύνολο δεδομένων εκτός σύνδεσης και η εκπαίδευση μπορεί να πραγματοποιηθεί σε λογικό χρόνο, είχαμε μεγαλύτερη ελευθερία να συντονίσουμε τις υπερπαραμέτρους του μοντέλου. Για κάθε σημείο ελέγχου που συγκρίνουμε τους πράκτορες μας, χρησιμοποιούμε διαφορετικό αριθμό κόμβων για τα κρυφά επίπεδα για να

βελτιστοποιήσουμε την απόδοση του μοντέλου. Επίσης, η υπερπαράμετρος a της συνάρτησης απώλειας CQL κυμαίνεται από 5 για το μικρότερο σύνολο δεδομένων έως 1 για το μεγαλύτερο σύνολο δεδομένων.

Τέλος, χρησιμοποιήσαμε μια πρόσθετη βελτιστοποίηση που περιγράφεται στη βιβλιογραφία ως μετατόπιση αρχικής τιμής (Schaul, 2021). Αν και η αποτελεσματικότητα αυτής της μεθόδου δεν έχει αποδειχθεί θεωρητικά, τα εμπειρικά αποτελέσματα δείχνουν ότι σε ορισμένες περιπτώσεις επιταχύνει δραματικά τη σύγκλιση. Η διαίσθηση πίσω από αυτή τη μέθοδο είναι ότι στο επίπεδο εξόδου του αλγορίθμου στο Deep Q Learning προσπαθεί να εκτιμήσει την αριθμητική τιμή της συνάρτησης Q για ένα ζεύγος δράσης κατάστασης. Όταν η φύση της συνάρτησης ανταμοιβής είναι τέτοια που μετατοπίζεται σημαντικά από το 0, τότε η αρχικοποίηση των biases του επιπέδου εξόδου στο 0, όπως συνήθως, μπορεί να είναι προβληματική. Ο λόγος είναι ότι ο πράκτορας θα αφιερώσει πολύ χρόνο εκπαίδευσης για να αυξήσει αριθμητικά τα bias για να φτάσει στην τάξη μεγέθους της συνάρτησης Q, δεδομένου ότι οι τιμές Q είναι απομειούμενα αθροίσματα των παρατηρούμενων τιμών ανταμοιβής, προτού αρχίσει να μαθαίνει αποτελεσματικά τη δυναμική του προβλήματος. Για να ξεπεραστεί αυτή η καθυστέρηση στη μάθηση, μπορούμε να αρχικοποιήσουμε τα bias με μια εκτίμηση της μέσης τιμής του συνολικού κέρδους του πράκτορα E[G] με βάση ορισμένα αρχικά στατιστικά στοιχεία. Τα πειράματά μας έχουν δείξει ότι αυτή η εκκίνηση με προκατάληψη πράγματι βοηθά την εκπαίδευση να ξεκινήσει νωρίτερα από την προσέγγιση μηδενικής προετοιμασίας.

Παρουσιάζουμε τώρα τη συγκριτική απόδοση του διαδικτυακού και του offline πράκτορα σε συγκεκριμένα σημεία ελέγχου εκπαίδευσης.

Ελάχιστο dataset(300 τυχαίες εμπειρίες):

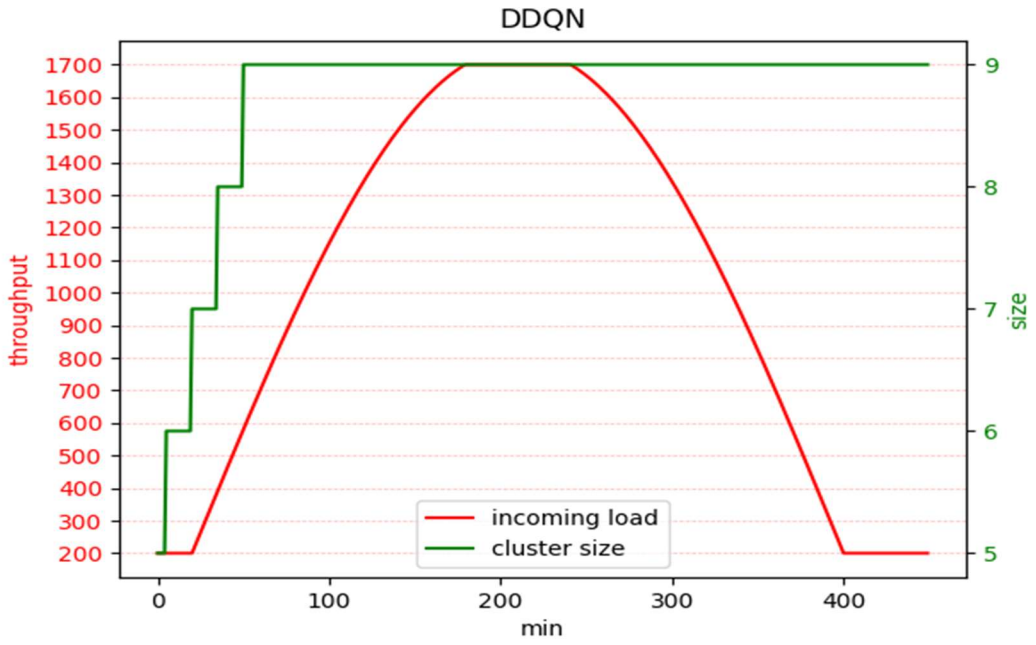

Figure 1: DDQN υπό ημιτονοειδές φορτίο

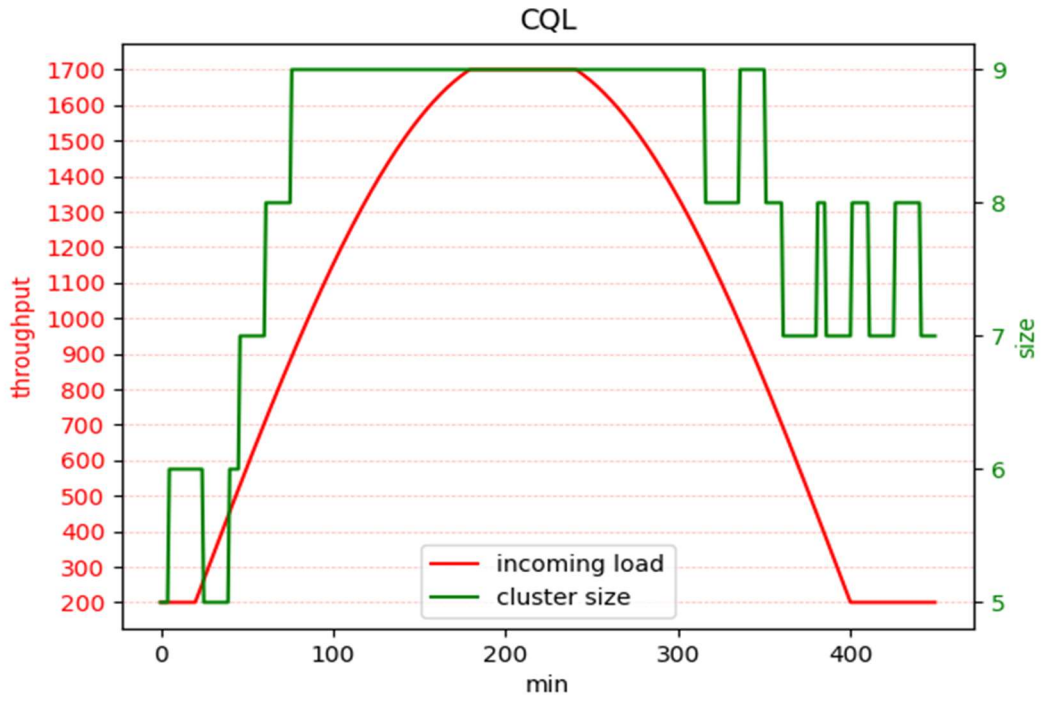

Figure 2 CQL agent υπό ημιτονοειδές φορτίο

Μικρό Dataset(800 εμπειρίες):

Σε αυτό το σημείο κάνουμε σύγκριση των μοντέλων μόλις έχει τελειώσει το κομμάτι της έντονης εξερευνητικής διαδικασίας δηλαδή όταν πλέον το ε έχει μειωθεί στο 0.1.

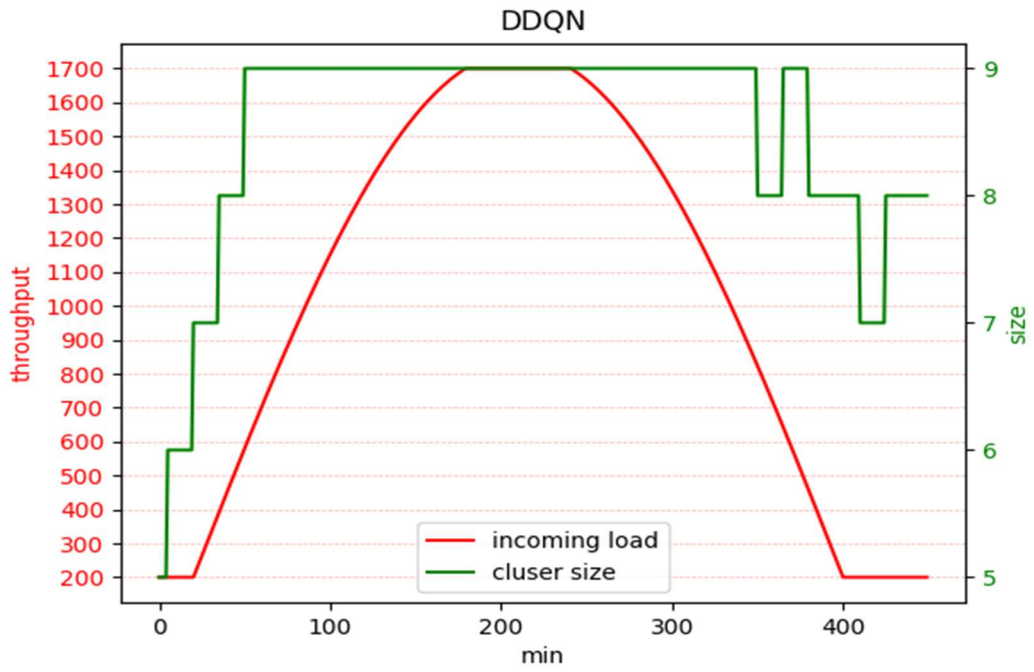

Figure 3: DDQN υπό ημιτονοειδές φορτίο για το μικρό dataset

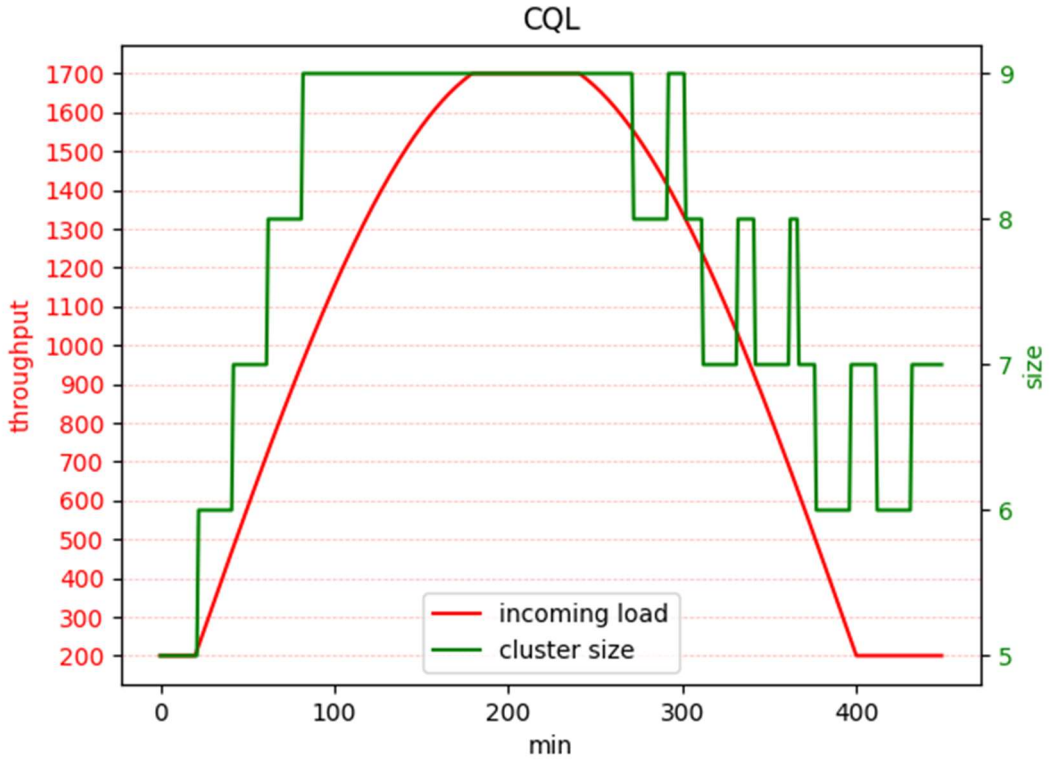

Figure 4: CQL υπό ημιτονοειδές φορτίο για το μικρό dataset

Dataset 1800 εμπειριών:

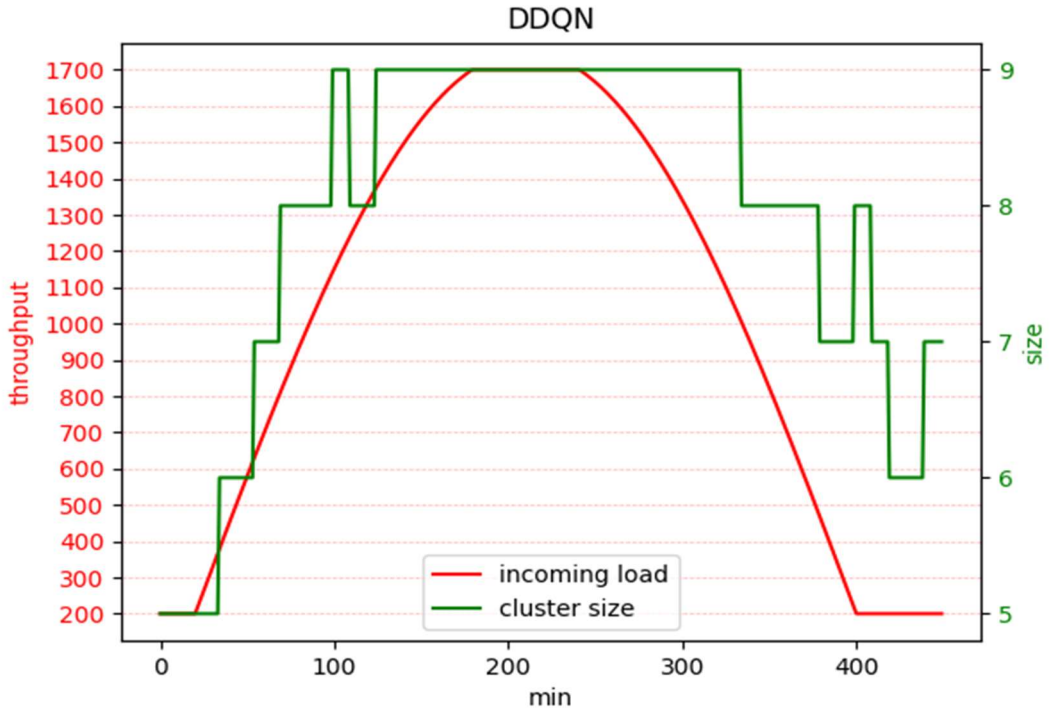

Figure 5: DDQN υπό ημιτονοειδές φορτίο για dataset 1800 εμπειριών

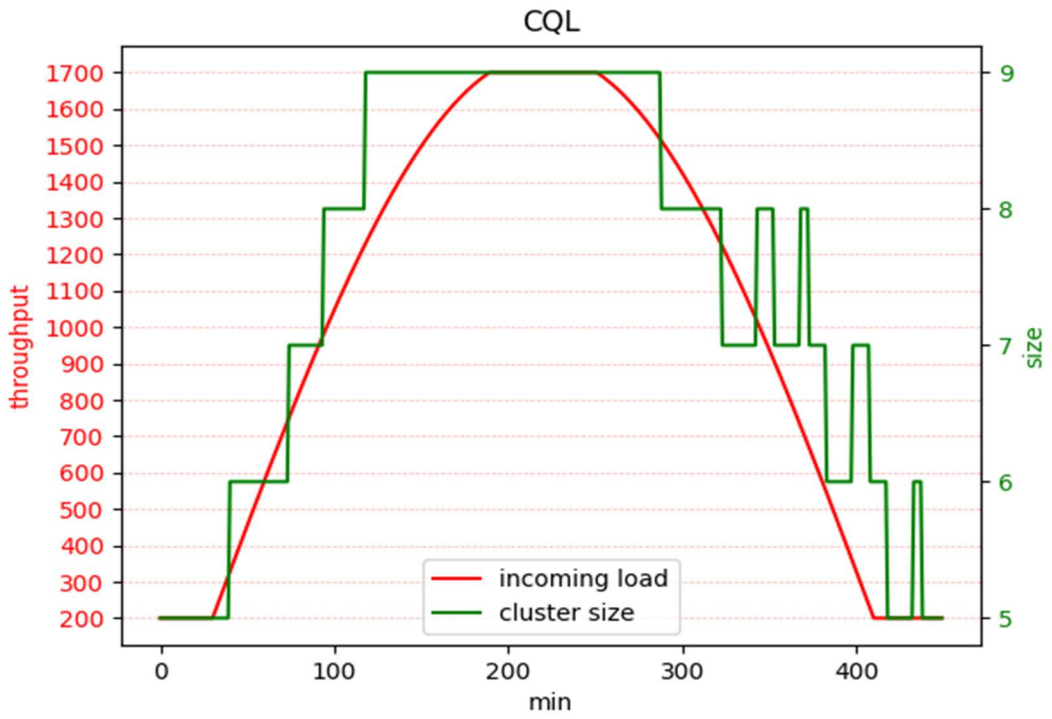

Figure 6: CQL υπό ημιτονοειδές φορτίο για dataset 1800 εμπειριών

Τελικό Dataset(3300 εμπειρίες):

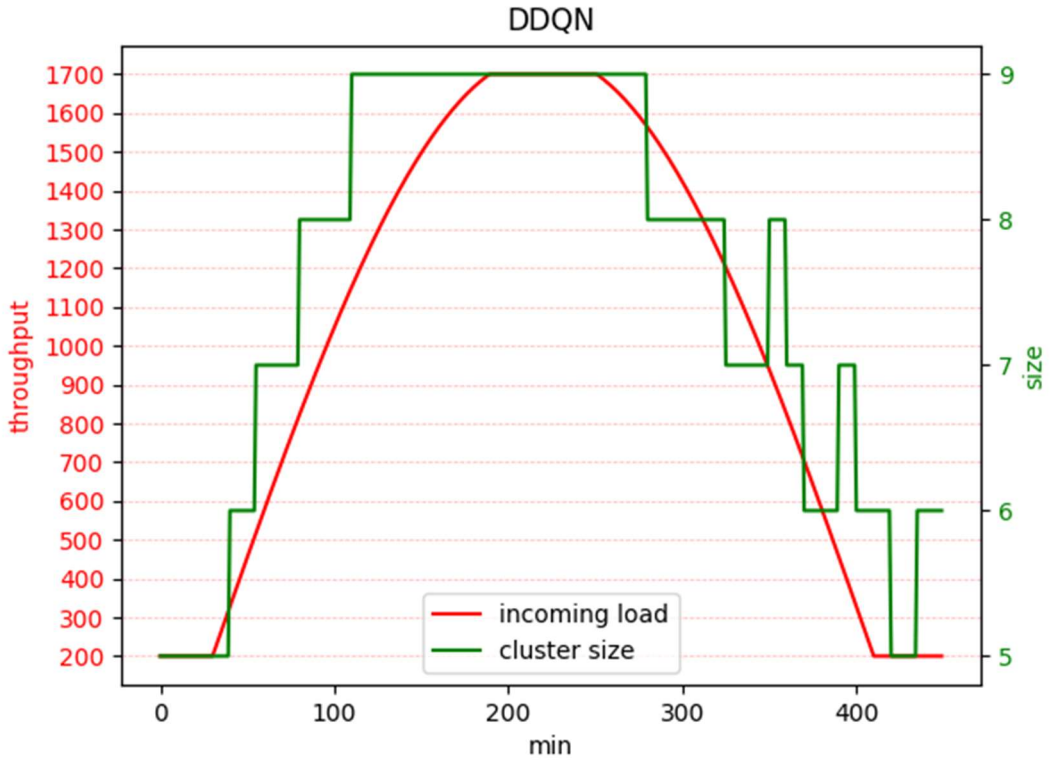

Figure 7: DDQN υπό ημιτονοειδές φορτίο για το μεγάλο dataset

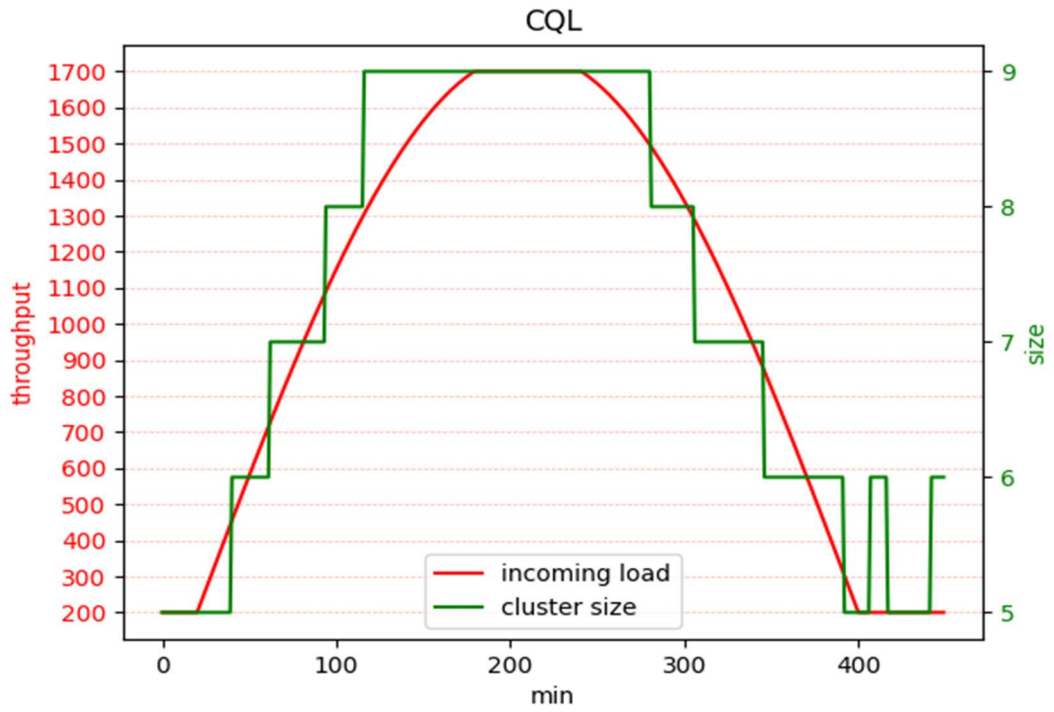

Figure 8: CQL υπό ημιτονοειδές φορτίο για το μεγάλο dataset

Από αυτό το σημείο και μετά δεν υπάρχει αξιοσημείωτη βελτίωση για το ασύγχρονο μοντέλο και η εκπαίδευση τερματίζεται. Το σύγχρονο μοντέλο συνεχίζει να βελτιώνει την απόδοσή του αλλά με πολύ πιο αργό ρυθμό.

Dataset	**DDQN**	**CQL**	**Improvement**
Minimal (300 exp)	581.43	642.73	10.5%
Small (800 exp)	601.74	670.05	11.3%
Medium (1800 exp)	659.71	698.36	6.1%
Final (3300 exp)	690.42	714.62	3.5%

Τέλος κάνουμε μερικά πειράματα για φορτία στα οποία δεν έχει εκπαιδευτεί ρητά το ασύγχρονο δίκτυο για να ελέγξουμε αν έχει δυνατότητες γενίκευσης.

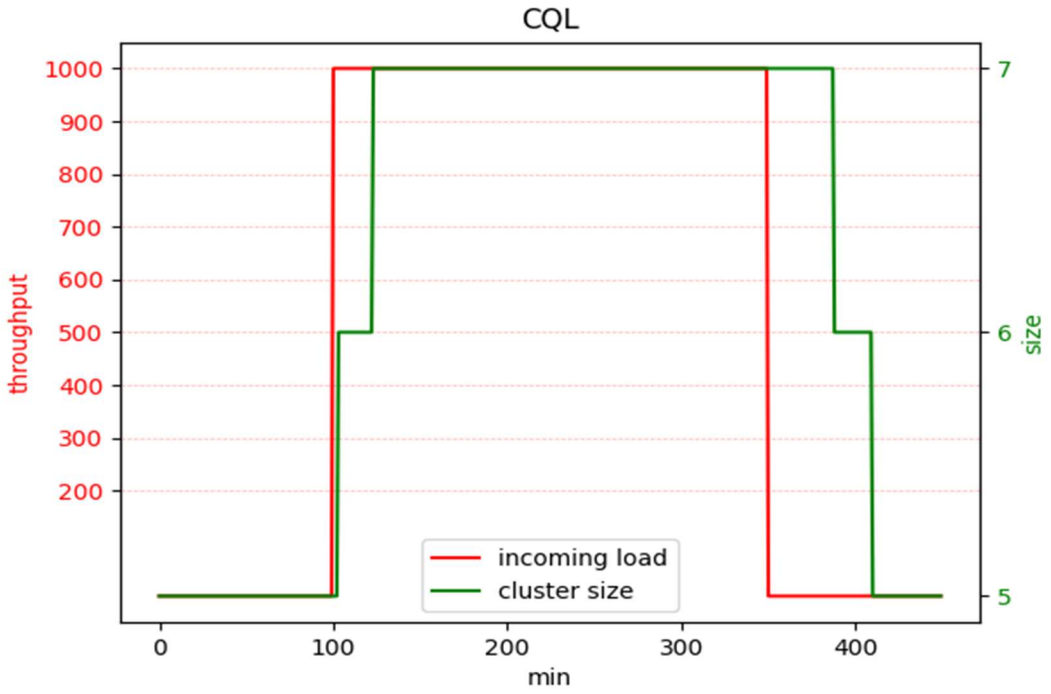

Figure 9 CQL υπό σταθερό φορτίο

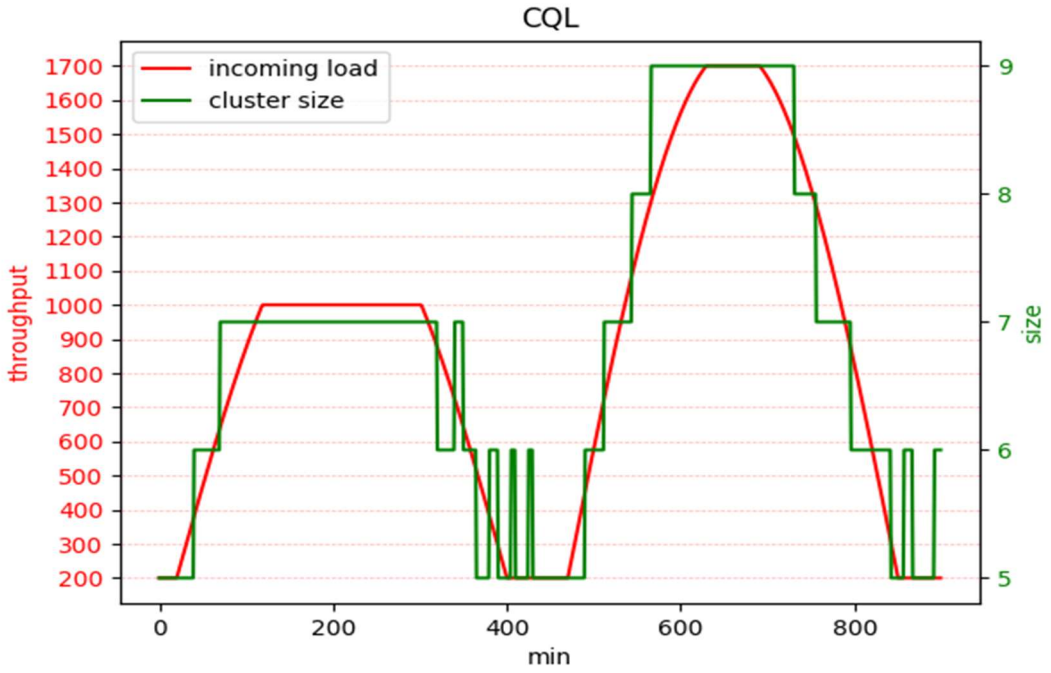

Figure 10 CQL υπό μεταβαλλόμενα φορτία με διαφορετικά ύψη

Συμπεράσματα

Κατά τη διάρκεια αυτής της διπλωματικής εργασίας, είχαμε την ευκαιρία να πειραματιστούμε με σύγχρονες τεχνικές μάθησης βαθιάς ενίσχυσης για να ενισχύσουμε τις δυνατότητες ενός ήδη ισχυρού εργαλείου για containerized εφαρμογές. Το προτεινόμενο μοντέλο μας παρέχει ένα σύστημα παρακολούθησης που μπορεί αποτελεσματικά να κλιμακώσει αυτόματα σύνθετες εφαρμογές προκειμένου να μεγιστοποιήσει τη χρήση των πόρων χωρίς να βλάψει την απόδοση. Επιπλέον, ο πράκτορας είναι σε θέση να ανακαλύψει τη δυναμική του συστήματος παρακολούθησης ακόμη και από πολύ μικρά σύνολα δεδομένων.

Η χρήση μιας containerized έκδοσης της εφαρμογής βελτίωσε τη διαδικασία εκπαίδευσης επειδή επέτρεψε τη συσσώρευση αυξημένου αριθμού διαφορετικών εμπειριών στο ίδιο χρονικό διάστημα σε σύγκριση με προηγούμενες προσπάθειες που βασίζονταν σε εγκαταστάσεις με VM. Η διαδικασία μπορεί να επιταχυνθεί περαιτέρω εάν επιλέξουμε να μειώσουμε τη χρησιμοποίηση των πόρων, ώστε να μπορούν να αφιερωθούν περισσότεροι πόροι για τις εργασίες κλιμάκωσης αντί για την εκτέλεση του φόρτου εργασίας. Παρόλα αυτά, η εφαρμογή μας έδειξε υψηλό επίπεδο ανθεκτικότητας και ήταν σε θέση να εκτελεί λειτουργίες κλιμάκωσης ακόμη και υπό σοβαρή πίεση πόρων. Αυτό οφείλεται στους αποτελεσματικούς αλγόριθμους scheduling του Kubernetes, οι οποίοι αναδιοργανώνουν τα εκτελούμενα container για να εξασφαλίσουν ελάχιστες διακοπές.

Τα πειράματα με τους σύγχρονους αλγόριθμους βαθιάς ενισχυτικής μάθησης τόνισαν τις πρακτικές προκλήσεις που εμφανίζονται κατά την εφαρμογή αυτών των μοντέλων σε ρεαλιστικά σενάρια. Το πιο σημαντικό είναι το γεγονός ότι αυτά τα μοντέλα χρειάζονται συνεχή και εκτεταμένη αλληλεπίδραση με το περιβάλλον που παρακολουθούν. Αυτό σημαίνει ότι για να πραγματοποιηθεί επιτυχής εκπαίδευση, το σύστημα πρέπει να διαμορφωθεί σχολαστικά, ώστε να αποφευχθούν απροσδόκητες συμπεριφορές λόγω των αποφάσεων του μοντέλου που μπορεί να οδηγήσουν το σύστημα σε καταστροφικές καταστάσεις. Η δεύτερη πρόκληση είναι ο περιορισμός στις συσσωρευμένες εμπειρίες. Η τυπική εφαρμογή βαθιάς ενισχυτικής μάθησης απαιτεί εκατομμύρια εμπειρίες και αυτό μπορεί να είναι ανέφικτο για ρεαλιστικές εφαρμογές. Τέλος, λόγω του περιορισμού στις εμπειρίες, η εκτέλεση βελτιστοποίησης υπερπαραμέτρων είναι μια πολύ χρονοβόρα εργασία.

Το ασύχρονο μοντέλο που προτείνουμε αντιμετωπίζει όλες αυτές τις προκλήσεις αποτελεσματικά. Πρώτα απ 'όλα, δεδομένου ότι το πρόβλημα μετατρέπεται ουσιαστικά σε ένα πρόβλημα μάθησης χωρίς επίβλεψη, είμαστε σε θέση να εκτελέσουμε βελτιστοποίηση των υπερπαραμέτρων για να εξαγάγουμε το βέλτιστο μοντέλο για το πρόβλημα. Επιπλέον, δεδομένου ότι το μοντέλο εκπαιδεύεται χωρίς καμία αλληλεπίδραση με το περιβάλλον του, είναι πολύ λιγότερο πιθανό μετά την ανάπτυξη να οδηγήσει το σύστημα σε καταστροφικές καταστάσεις, εάν το πρόβλημα οριστεί σωστά. Το ασύγχρονο μοντέλο είναι σε θέση να εξάγει σημαντικά περισσότερες πληροφορίες από το παρεχόμενο σύνολο δεδομένων, σε σύγκριση με τον σύγχρονο πράκτορα. Ως αποτέλεσμα, είναι σε θέση να συγκλίνει σε μια λύση πολύ πιο γρήγορα. Τέλος, παρατηρούμε ότι σε κάθε σημείο ελέγχου της εκπαίδευσης, το μοντέλο εκτός σύνδεσης είναι σε θέση να μετριάσει την προκατάληψη σύγχρονου μοντέλου προς καταστάσεις υψηλότερου μεγέθους συμπλέγματος, γεγονός που

υποστηρίζει τον ισχυρισμό ότι το μοντέλο μας είναι σε θέση να αντλεί συστηματικά καλύτερη λήψη αποφάσεων πολιτική από αυτή που παρέχεται από το σύνολο δεδομένων.

Αν και περιορίστηκε σε ορισμένες περιπτώσεις, το μοντέλο μας έδειξε κάποιες δυνατότητες γενίκευσης σε φόρτους εργασίας που δεν είχε γίνει εκπαίδευση. Η εκτεταμένη γενίκευση με τη Βαθιά Ενισχυτική Μάθηση παραμένει ακόμη ένα άλυτο ζήτημα. Ωστόσο, αυτά τα αποτελέσματα είναι ενθαρρυντικά για περαιτέρω πειραματισμό, ειδικά με τεχνικές ασύγχρονης Ενισχυτικής Μάθησης που αντιμετωπίζουν άμεσα το πρόβλημα γενίκευσης.

2 Introduction

2.1 Motivation

Cloud native applications are one of the main focus points of enterprise software development. In 2021 about 30% of new generated digital workload is deployed on cloud native platforms and it is estimated that this figure can climb up to more than 90% by 2025. The dynamic and scalable nature of cloud native applications is a promising alternative to on-premises development because it alleviates an organization from the costs and risk of maintaining costly infrastructure to support its software operations. Moreover, scaling is limited, time consuming and even impossible in certain scenarios.

Containerization enhances the potential of cloud native applications because it offers a lightweight alternative to virtual machines that need to boot an OS instance to run. Using containerization, the components of an application can run with a significantly smaller hardware and time overhead. Capitalizing on these properties, cloud applications can be organized in components that execute atomic functions of the application, rather then running monolithic instances of the application. This software development paradigm is called *microservices* and it offers increased flexibility in terms of scaling because each component can be scaled independently.

As containerized applications become more complex and more distinct components are added to the deployment, monitoring, scaling and connecting the components of the application becomes a tedious task. The rapid increase in DevOps complexity of containerized applications limited their adoption and created the need for orchestration mechanisms. Kubernetes is a software solution that solves the orchestration problem effectively, a fact that resulted in its increased adoption by enterprises since its release in 2014. Kubernetes also offers resource-based autoscaling features such as the *Horizontal Pod Autoscaler* (HPA). This auto-scaling feature uses CPU and memory utilization thresholds to estimate the optimal number of instances to run. Some cloud services vendors offer threshold based autoscalers that increase the number of application instances as the incoming traffic increases. These methods are oversimplistic and can only provide limited performance guarantees.

An alternative approach to threshold based autoscaling, is to utilize algorithms from the domain of Reinforcement Learning (RL). Reinforcement Learning formalizes the idea of an agent that learns effective decision-making policies by performing trial and error interactions with a system. Deep Reinforcement Learning enhances the capabilities of RL by utilizing neural networks as high order non-linear function approximators. In that way, problems that are described by more complex state spaces and could not be solved by original RL algorithms, are now solvable. However, there is a key limitation to applying Deep Reinforcement Learning to the resource auto-scaling problem. In typical Deep Reinforcement Learning problems mentioned in literature, the number of training steps required to reach an optimal solution is in the order of millions. In typical auto-scaling scenarios the time it takes for an action to take effect is in the order or minutes. Additionally, interacting with the application in a random manner is in some cases not desirable because it can cause disruptions or lead it to destructive states. Considering the

above, it is evident that in order to derive a feasible solution for the auto-scaling problem the number of interactions with the system must be reduced significantly or alternatively the information gain from each interaction must be maximized.

2.2 Related Work

The most common way to deal with the issue of elasticity is auto-scaling. Amazon's auto-scaling (AWS, Amazon Autoscaling) for instance dynamically increases or decreases a user's resources based on thresholds applied on user cluster's specific metrics. Microsoft's Azure (Microsoft's Azure) and Celar (I. Giannakopoulos, 2014) use the same technique. Yet, as shown in (K. Lolos, 2017)these approaches are difficult to calibrate and optimize.

The authors of (R. Taft, 2018) use a dynamic programming algorithm that tries to determine through a series of past experiences the optimal behavior for the system's next-state. Markov Decision Processes (MDPs) and Reinforcement learning algorithms have been used in (D. Tsoumakos, 2013) to address issue, as well as an approach involving wavelets for prediction of a cloud state and resource provisioning. However, the efficiency of those approaches decreases, as the number of possible states increases. The input parameters of the system (metrics of the cluster) are continuous variables; therefore the number of discrete states can grow exponentially.

To manage this issue in (K. Lolos, 2017) the authors propose an RL approach combined with decision-trees algorithms, in order to split the input parameters based on some split criteria. This approach manages to generalize over the input and to train the agent so that it can find out on its own which state parameters matter to the desired outcome and which not. Nevertheless, this approach also struggles with large space of states and also need a large dataset to show generalization capabilities.

The authors of (Constantinos Bitsakos, 2018) proposed a Deep Reinforcement Learning model to address the problem of elasticity in cloud native environments. The model is able to converge to a solution and provide increased rewards compared to previous approaches that did not utilize neural network. The main issue still is the fact that the number of training samples is significantly large.

The authors of (Lucia Schuler, 2021) utilize Reinforcement learning algorithms to address the problem of adaptive auto-scaling for serverless applications. Their approach concerns the distribution of available workload to application containers and monitoring the performance of the system in terms of latency and throughput based on the concurrent requests each container has to serve. They then generate a policy using Reinforcement Learning to perform optimal distribution of workload. Their approach is similar to ours although it only applies to serverless applications. However, the states are described using only three parameters, a fact that makes the exploration space much smaller and easier to converge to a solution.

2.3 Proposed Solution

The proposed solution of this thesis is to create an agent that monitors a containerized application and dynamically scales the deployed instances and consequently the consumed computational resources. To develop this agent, we create two neural networks that are trained using online and offline Reinforcement learning algorithms. There are two main challenges that need to be tackled by the proposed solution.

- The number of parameters needed to describe the state of the system are can increase, depending on the complexity of the deployed application. Also, the parameters are not discrete. This makes tabular applications of Reinforcement Learning ineffective and consequently the utilization of a neural network is justified.
- The time between consecutive actions is in the order of minutes. This limits the rate of data accumulation, meaning that the model must extract as much information as possible from available data points.

To address these challenges, we propose a model that consists of two neural networks. The first model is trained in an online manner, using the *Double Deep Q Learning* algorithm. The second is trained with an offline dataset using the *Conservative Q Learning* algorithm. The offline dataset consists of the training experiences that are generated by the online model during its training. Essentially, the offline model receives the data generated by policy of the online agent and trains without further interacting with the system. To decrease the time it takes to perform the scaling actions we deploy our application inside a distributed Kubernetes cluster to leverage its automated scheduling features and the open-source controllers that handle the operations necessary, after an increase of decrease in the number of instances of the deployed application. The application we chose to monitor is a Cassandra cluster. The choice of this application is based on the fact that its performance depends on serval parameters, a fact that highlights the effectiveness of our model.

We test both models at several checkpoints of the training and show that the offline model outperforms online training up to a certain size of dataset. We also provide empirical results that indicate that from the point onwards where online training outperforms offline training, performance gains diminish dramatically in comparison with the training steps required to achieve this improvement.

3. Theoretical Preliminaries

3.1 Containerization

Virtualization technologies are an essential part of modern cloud infrastructure because they facilitate efficient utilization of the physical machines' resources that host virtual machines (VMs). Additionally, they isolate virtual machines which is important to ensure security. Virtualization however, comes with a significant hardware overhead, since every VM needs to have its dedicated kernel which results in increased resource usage. Also, the hypervisor, the program responsible for creating and running VMs on a physical machine, consumes a rather large fraction of resources by itself (Scheepers, 2014).

Containers alleviate this problem by providing a "lightweight" alternative for running isolated applications, while sharing the same kernel of the host Operating System (OS). Containerization evolved from Linux *cgroups* (citrix.com, 2022), which became Linux containers (LXC). Cgroups isolate and control the resources any given process can allocate, for example thread number and CPU or RAM usage. LXC provide additional isolation using *namespaces.* Namespaces further isolate containers, meaning every container has its dedicated filesystem, network stack, user management and process ids. Using this abstraction every container can run its own OS distribution regardless the OS of the hosting machine given that the underlying distribution runs a Linux Kernel (Scheepers, 2014). In that way, the booting a container results in the same experience regardless of the computing environment. This cross-platform compatibility is essential to today's cloud environments, that usually consist of heterogenous systems and operating systems.

Containerized applications have significantly reduced booting times, since there is no need to boot an operating system, except for some binaries which are included in the container image. This means that deploying and destroying containers on demand, is much cheaper in terms of resources and elapsed time. The aforementioned characteristics allow for greater flexibility, when scaling cloud applications. If we also consider the fact that the most common approach to cloud native application development is *microservices*, where the application is divided in distinct software artifacts, that operate and are scaled independently and communicate only through APIs, it is apparent that containerization is the most suitable isolation technique to facilitate this software development paradigm.

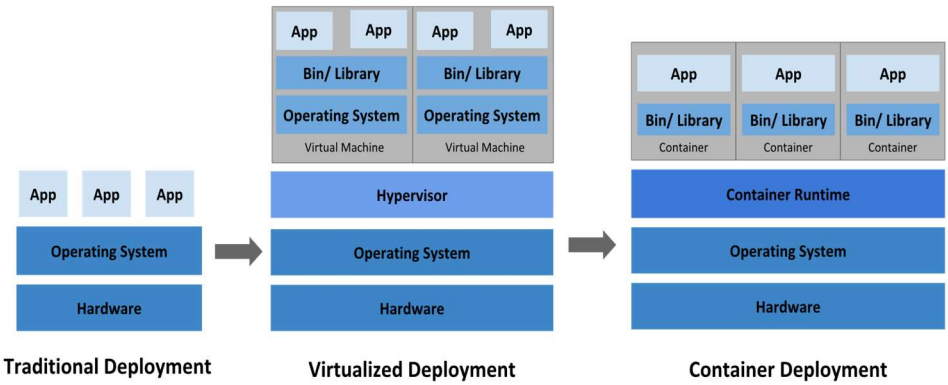

Figure11 : Application Deployment History

3.2 Kubernetes

Figure 12: Kubernetes Logo

Docker provides a framework to create containers that are reproducible and run in the same way regardless of the underlying architecture. Also, by providing efficient ways of storing and executing the containers, it promotes and supports the software development paradigm of microservices. Ideally, each container runs a single software component of the application that runs and can be scaled independently. However, as applications become more complex and software components increase in number and variety, performing DevOps becomes a tedious task and performing rolling updates, rollbacks and scaling is not a trite procedure. The above tasks created the necessity for an orchestration mechanism to deploy, monitor, maintain and scale containerized applications. Kubernetes, initially launched in 2014 is an attempt to create such a mechanism.

Some of the core utilities of Kubernetes are:

- **Service discovery and load balancing** Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.
- **Storage orchestration** Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more.
- **Automated rollouts and rollbacks** You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, you can automate Kubernetes to create new containers for your deployment, remove existing containers and adopt all their resources to the new container.
- **Automatic bin packing** You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources.
- **Self-healing** Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.
- **Secret and configuration management** Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration. (https://kubernetes.io/, 2022)

Overall Kubernetes provides tools to ensure resilience and fault tolerance of the application and also minimum or no downtime at all during updates in production environments.

3.3.1 Core Components

Kubernetes consists of two types of nodes, worker nodes and master nodes. Kubernetes nodes can reside in the same physical or virtual machine, for example when deploying a vanilla cluster using Minikube, or in separate machines for larger deployments.

Worker nodes host the Pods of the application (Pods encapsulate the containers used by Kubernetes and is the smallest piece of software Kubrnetes can execute). Master nodes run the control-plane, which monitors and controls the application Pods as well as the worker nodes of the cluster. In production environments, usually more than one master nodes run the control-plane for fault tolerance and high availability.

The software components of the control plain are:

- Kube-apiserver
- Etcd
- Kube-scheduler
- Kube-controller-manager
- Cloud-controller-manager

Kube-apiserver: It is the frontend of the Kubernetes cluster. It validates data and exposes the cluster's shared state to all the other components of the cluster, for example pods, services and controllers. It also provides an API that the manager of the cluster can use monitor and perform manual changes to the cluster configuration. It can be scaled horizontally to offer higher availability when traffic is increased.

Etcd: It is a persistent key-value database that the Kubernetes uses to store the cluster state as well as the previous states. Every key created in the database is immutable. This means that subsequent updates to the value of the key create new versions of the key rather than mutating the data of the existing key. That said, it is apparent that the versions of every key increase monotonically throughout the lifecycle of the cluster. For storage efficiency reasons, older versions are compacted. When a key-value pair is deleted, its version is reset to 0 and then in the next compaction all keys with version 0 are not included in the compaction. In terms of implementation, etcd uses a persistent b+ tree to store information. It also preserves an in-memory b+ tree index to speed up range queries over the keys. The keys of the in-memory tree point to the most recent mutation of the data. During compactions, the structure is updated and dead pointers are removed (https://etcd.io/, 2022).

Kube-scheduler: As stated by the name, the scheduler is responsible for watching for new Pods that are not assigned to a worker node and then choose the node to run them on. To chose the most appropriate node for each Pod the scheduler performs two operations on existing nodes, *filtering* and *scoring.*

Every pod comes with a set of requirements. These requirements can be in terms of hardware resources available, workload interference, data locality or node/pod affinity/anti-affinity. For example, a pod that runs a stateful software component must run on a node that contains the correct state information on an ephemeral or persistent storage. Also, affinity or anti-affinity scheduling rules that are included in the pod specification, explicitly declare the nodes and/or pods that the new pod can or cannot be paired with. By

considering the above restrictions the scheduler creates a list of all the *feasible* nodes, the nodes that the pod can be scheduled on without violating any of the restrictions. If the list of feasible nodes is empty then the Pod remains in the unscheduled state until an appropriate node is observed. This can be either a new node or an existing node, whose workload was changed. After the list of feasible nodes is created, the nodes are sorted based on scoring functions. Scoring ensures a local optimum that is likely to lead to a better cluster state consecutive pod assignments to nodes. There are many factors that are considered during node scoring. For instance, a node that is going to reach its full resource capacity after assigned the new pod is less likely to be chosen. A node that already has the docker image stored is more likely to be chosen. Also, the scheduler ranks nodes, that already run instances of pods that belong to the same service, lower to ensure a more uniform distribution of the workload of service within the cluster. In that way, if a given node fails it is less likely to cause disruption to the service. After the scheduler calculates the scores of each node the highest ranked node is chosen and the scheduler binds the pod to that node and updates the kube-apiserver.

Kube-scheduler is highly configurable in different ways. First of all, there are a plethora of options to restrict scheduling in order to optimize scheduling performance and decision latency. The parameters of the algorithm can also be configured to tune performance. For example, in large cluster that many nodes are feasible it is beneficial in terms of performance to do a partial rather than an exhaustive search of the available nodes, so the scheduler can be parametrized to only search for a fixed number of feasible nodes. Scoring can also be set so that all nodes above a threshold are considered optimal and the scoring procedures terminates prematurely when such a node is discovered (https://kubernetes.io/, 2022). Finally, Kubernetes offers the flexibility to replace the default scheduler with a custom controller.

Kube-controller-manager: The kube-controller-manager is the “heart” of the Kubernetes cluster. It is a non-terminating loop that communicates with the kube-apiserver and monitors the shared state of the cluster. When the current state is not the same as the desired state it periodically performs action to attempt approaching the desired state. The desired state is such that all the pods are scheduled and healthy, all replicasets, statefullsets and deployments deployed have the number of pods specified in their –replicas attribute in running state and all jobs are in completed state (the above terms will be covered in more detail in the following sections).

The kube-controller-manager consists of several controllers that monitor smaller parts of the cluster independently but they are integrated in a single process for efficiency. Some of the controllers inside the kube-controller-manager are:

- Node controller: Monitors nodes and when a node stops responding waits for a grace period and then evicts pods from the non-responding node and reschedules them.
- Replica controller: It monitors existing replicasets and ensures that a sufficient number of pods run for each deployment.
- Service Account & Token controller: Create accounts and API access tokens for new namespaces. In that way more than one applications can reside in the same cluster and be isolated from one another.

Cloud-controller-manager: Kubernetes is created with the intent to be integrated with the cloud infrastructure of major cloud service providers. When the cluster is deployed using one of the supported cloud vendors then the cloud manager is deployed and runs some controls to provide additional functionalities. For example, the cloud manager can understand if a non-responding node is deleted or temporarily not responding and act accordingly. The controller can also leverage the load balancing functionalities of the vendor. In bare-metal or local machine installations the cloud-controller-manager does not exist.

Except for the components that run on the master node there are some components that must run on every worker node to enable cluster functionality. The first is the kube-proxy. The kube-proxy is responsible for the low-level implementation of the services created in a Kubernetes cluster. Services, are a way to create static connections between groups of pods with each or expose a group of pods outside the cluster. This is not trivial, since every pod created is assigned a new IP address that persists for the entirety of its lifecycle. As pods are created and destroyed, the IP addresses of a group of pods changes over time. Services provide an immutable way to connect to these pods regardless of their IP addresses at any given time. The kube-proxy implements this functionality by monitoring the nodes running on the node and updating the iptables of the hosting machine when changes occur.

The second component is the kubelet. The kubelet is a container manager and its responsibility is to start the execution of Pods as well as monitor their state and try to ensure that they are healthy. To know which Pods to run it receives a PodSpec yaml file from the kube-apiserver. It also interacts with the etcd database to update information about the state of the pods. It also periodically checks the liveness(the pod is running) and the readiness(the pod is set up and ready to execute its function).

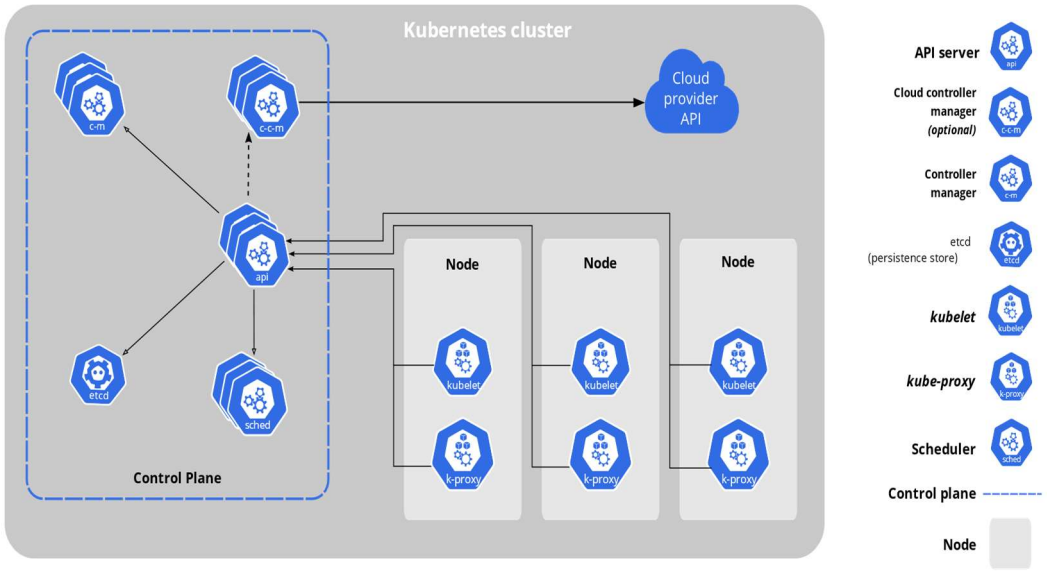

Figure 13: Overview of a Kubernetes Cluster and its core components

3.3.2 Core Concepts

The workload we wish to execute in a Kubernetes cluster is described by a collection of objects. There are three different ways to declare an object to Kubernetes. The first is by using imperative commands. This type of declaration is suitable for development purposes and offers a quick way to create a new object. However, it provides no template to create similar objects in the future and also it creates no trail of the changes applied to the object, only the live state is preserved. The second and third declaration techniques use yaml files (Yet Another Markup Language) to create new objects or update existing ones. Imperative object declaration uses a .yaml file and executes an action explicitly stated (create, delete, update) using that file. The main advantage is that the file acts as a reference point for future deployments of instances of the same object. However, a .yaml file has to be created which requires a certain level of knowledge. Also, if an update is executed, changes must be merged in the new version of the configuration file or they will be lost during the next replacement. Declarative object configuration uses .yaml files and the apply command of kube-controller. Kubernetes then detects all objects to be created or updated and automatically performs the appropriate actions to every object declared in the yaml files. This way of declaration makes changes persistent even if they are not explicitly merged and is also suitable to perform on directories of yaml files. However, it can create a complex sequence of changes to the objects, which makes the procedure less transparent and harder to debug in cases of unexpected behaviour.

Kubernetes objects act as a “record of intent” for the cluster. This means that from the moment an object is declared the cluster will continuously perform actions to ensure that the object exists. Essentially objects, describe how we want our cluster workload to look like, so the collection of the declared objects represents the *desired state of the cluster.* Kubernetes includes a plethora of available objects in its documentation that offer very complex and expressive ways of workload orchestration. Describing all of them is out of the scope of this thesis, so we will only mention the objects that are used for the deployment of our application.

Pods:

Pods are the smallest unit of computing that can be deployed in the Kubernetes ecosystem. A pod is a group of containers that are always co-scheduled on the same node and share the same computational resources, filesystem and storage. The shared environment within a pod consists of a shared set of cgroups and namespaces, analogous to a Docker container isolation setting. Pods represent a cohesive unit of service of our application and these containers must always be run and scaled together. Pods are rarely created as standalone objects because a Pod is an ephemeral structure, meaning that if no additional objects are declared that use a certain Pod then its lifecycle can be terminated without further notice, due to lack of resources for example.

Deployments:

Deployments are objects that enable to perform rolling updates or rollbacks to existing groups of pods or create new ones. It is an important feature of Kubernetes because one of the major issues that it claims it can solve it the durability issues and minimal downtime during updates. During a rolling update, pods that run the latest configuration of the provided specs, are gradually scheduled to the nodes and are started. As more pods running the latest version are reaching the Ready state, pods running the old version are gradually terminated and replaced. By default, deployments ensure that no more than 25% of the desired number of pods are unavailable at any given time and no more than 125% of the desired number of pods are started simultaneously. Of course, these parameters are configurable. If for some reason a scale operation occurs during a rolling update, for instance if horizontal autoscaling is enabled, then the cluster performs proportionate scaling. This means that the cluster creates pods running the latest version as well as pods running the old one, at a proportion that preserves the ratio before the scaling started. In that way it ensures that scaling is less likely to disrupt the rolling update procedure.

ReplicaSets:

A ReplicaSet is a Kubernetes object that monitors Pods belonging to a group declared in its specs and tries to ensure that at any given time the desired number of these pods run on the cluster. In the ReplicaSet specification additional parameters can be configured, to specify the node selection process and the image the pods must run. When changes are applied to the ReplicaSet manifest (by declarative of imperative configuration) then the ReplicaSet ensures that the old pods are terminated and new ones, complying with the updated specifications are running. This object is suitable for stateless workload application components because when an instance is terminated or replaced no state information is preserved.

DaemonSets:

DeamonSets are similar to ReplicaSets but they ensure that an instance of the pods defined in their specification are running on every node of the cluster or on every one of the nodes explicitly or implicitly declared in the specification. Pods controlled by DaemonSets are not scheduled by the kube-scheduler but instead use a dedicated controller that deploys these pods to nodes independently. In some cases, this can cause scheduling problems especially when the workload running on a node is close to its resource limit. In such cases it is possible to disable the dedicated controller and force daemonset pods to follow the normal scheduling process. This object is suitable for pods that perform node specific functionalities for example log or metric scraping and exposing them to the cluster, node monitoring or persistent storage handling.

StatefulSets:

Originally Kubernetes was developed with intent to support stateless applications primarily. However, as its popularity grew and the desire to support more complex applications rose, it was inevitable that *statefullness* of workloads had to be supported. StatefulSets fulfill this purpose. As all previous objects, StatefulSets monitor a group of objects and ensure that the desired number is running on the cluster. Unlike the previous categories however, every pod created as part of a StatefulSets has a sticky identity that persists as long as the number of replicas of the set is not changed. This means that even if a pod is terminated or evicted, after a new instance is recreated it will preserve the same identifier. Also, the pod receives a unique and stable network identifier.

Preserving the state of the pods also introduces some restrictions in the deployment and scaling of StatefulSets. During scale up or scale down, only a certain number of pods can be created or terminated simultaneously. By default, only one at a time is created or terminated unless the StatefulSet is divided in subgroups. In that case only one per subgroup is created or terminated. In stateful applications, during scale up or scale down some information must be transmitted between the Pods of the set, to update their states about the change that is about to happen, so that the state of workload is consistent. For this reason, a new pod belonging to the set can only start if all the predecessors are in running state and a pod can be terminated if all of its successors have already been terminated. The restrictions introduced can create delays or even block the scaling procedure if one of the pods fails and cannot reach the running state again. This creates the need for more advanced controllers that are application specific, to handle internal errors accordingly so that pods can recover from failure.

StatefulSets provide the basis for implementing stateful applications but the pods still remain ephemeral. If the application demands that the state is preserved and cannot or should not be recovered completely by other pods, then the StatefulSet must use some form of persistent storage to permanently store the necessary information.

PodDisruptionBudget:

Kubernetes runs on a distributed infrastructure and in order to ensure robustness of the system to faults, it must account for unexpected failures and perform actions to mitigate their impact on the workload. So far, we covered how Kubernetes handles rolling updates on sets of pods to ensure robustness. Other reasons of disruption can be the draining of a node (removing all running pods from a node) to perform upgrades, repairs or scale down the cluster and removing a pod from a node to change the scheduling configuration of the cluster for optimal performance. These types of pod disruptions are called *voluntary,* whereas disruptions called from nodes becoming unavailable either due to physical system failure or VMs being destroyed (by the cloud provider for example) or network partitioning, are called *involuntary.*

Kubernetes offers an object that provides an additional layer of durability when voluntary and involuntary disruptions occur simultaneously. They are called PodDisruptionBudgets and they state the minimum number of pods of a deployment that

must be running at any given time. It can be an absolute number or a percentage of the replicas available. So, if during a rolling update an involuntary disruption occurs, leading to the number of available pods dropping under the defined PodDisruptionBudget threshold, then the update process will be paused until a sufficient number of pods reach the ready and running state, to prevent application downtime.

Services:

As we mentioned before every pod on creation is assigned a unique IP address so it can be accessed from inside the cluster. Pods however, are ephemeral and a group of pods performing a certain function does not have static IP addresses. To support the microservices software paradigm there must be a mechanism that allows pods to communicate with each other while their IP addresses alter over time. Services are this exact mechanism. In essence a service is an abstraction that defines a set of pods and a policy on how to forward traffic to the pods. The most common way to define the group of pods is using a pod selector. A pod selector is just a key-value label that is stored in the pod template when it is created. The service then uses this key-value selector to monitor which IP addresses belong in the group and updates the endpoints.

Services can also be used to forward traffic to or from external addresses for example to query an external database or to make the running application available outside the cluster. If the Kubernetes cluster is deployed using on of the major cloud services vendors (AWS, Azure, GKE etc.) then services can leverage the load balancing features they provide. By setting the type of the service as *LoadBalancer* the service is connected to a load balancing resource of the cloud provider so that traffic can be balanced between nodes.

Whenever a new service is created, it allocates a port of the node in the range 30000-32767. In large scale deployments where multiple users create services, potentially from different namespaces, there is the risk for overlapping node assignments. Kubernetes solves this problem by preserving a persistent data structure that stores all existing assignments and updates its mapping prior to every new service creation. It also automatically collects IP addresses that are no longer used by any service.

Volumes:

On-disk files in a container are ephemeral so during crashes the files are lost. This creates problems since for Kubernetes the smallest computing unit is the pod, which can potentially consist from multiple containers and when one of them crashes the pod should not crash as well, so data should be preserved. Also, the filesystem should be accessible to all containers inside a pod if required. That is why Kubernetes defines the volume objects that extend the functionality of volumes defined by Docker to support the need for ephemeral or persistent storage in pods. Ephemeral volumes are deleted once a pod is terminated but persistent volumes are not cleared unless explicitly declared so. Both ephemeral and persistent volumes are preserved during container restarts. At its core, a

volume is a directory, with or without data that can be accessed by containers inside a pod. The content of the directory, the medium that supports the directory and how this directory is created, depend on the type of volume used. Kubernetes offers a plethora of volume types that create volumes using the volume features of most major cloud services vendors.

Persistent Volumes:

Persistent volumes are used to store data that are preserved between pod terminations until the user decides to delete them. Volumes can be backed by various cloud infrastructure technologies, each with its own features and implementation details. Kubernetes however, aims to separate dev from ops in cloud applications. In the same way that users deploy their application components without concerning themselves with allocating computing resources, they should be able to provide persistent storage to them without concerning themselves with the implementation specifics of each backing technology. That is why Kubernetes offers an additional level of abstraction for persistent volumes to separate how volumes are provided from how they are consumed. To achieve this, it uses three additional resources, *PersistentVolumes, PersistentVolumeClaims* and *StorageClasses.*

PersistentVolumes (PV) is the resource that represents a persistent volume that is available in the cluster. In other words, it's a volume that has already been configured by a cluster administrator and is ready to be used. To use available volumes, pods create PersistentVolumeClaims (PVC) before they initiate the execution of their containers. These are claims for a persistent volume that complies with a set of specifications explicitly described inside the claim. When a new PVC is created, a controller inspects its specifications and if it finds a PV that satisfies them, it binds the PVC with the PV in a one-to-one relation. The pod then receives information about the location of the bound PV and starts its execution using the PV. To match a PVC with a PV, the volume must provide at least the requested specifications, for example storage size and access writes, or an excess of the requested specifications. If a suitable PV does not exist, the PVC remains unbound and the pod stuck until an appropriate PV is provided to the cluster.

StorageClasses are used for two reasons. First to relieve application developers from explicitly stating all the characteristics of a PV or PVs they want to request for a pod. We could say that StrorageClasses describe different profiles of volumes and each PersistentVolume is an instance of a StorageClass (a PV that is not explicitly declared as an instance of a StorageClass is an instance of the DefaultStorageClass). In that way, developers can request for a specific StorageClass and then the controller will try to allocate an appropriate volume instance. The second reason is to facilitate dynamic provisioning. When the underlying technology supports such a feature, Kubernetes can dynamically provision volumes on demand. To do so, the pod must request a StorageClass instance that supports this feature. Then when the controller inspects the created PVC and finds no matching volumes, it notifies the cloud-controller-manager to request a new volume from the specified endpoint. The new PersistentVolume is automatically created and bound with the PVC. In that way the available persistent storage in the cluster can grow or shrink according to the demands of the application at different times, leading to more efficient pricing.

Ephemeral Volumes:

Ephemeral volumes support the need of some applications to store temporary data before or during execution without the need to preserve them across restarts. For example, some applications need some read only data that provide initialization parameters (ConfigMaps). Other application might need to store temporary results during intensive computations to alleviate memory pressure and reduce the risk of being terminated because of exceeding resource limits. Kubernetes provides this type of volumes to simplify deployment and management in these cases, since there is no concern for data perseverance.

CustomResources:

Kubernetes offers several resources like the ones described above but in some scenarios there might be a need for additional monitoring of resources. For this purpose, Kubernetes allows to define custom resources that provide additional layers of control and extend the standard installation. The most common custom resource are custom controllers. Just like built-in controllers, they monitor the state of a part of the cluster and periodically perform actions to help the cluster reach the desired state in term of the resources monitored by the custom resource. This can be useful in cases where the application demands monitoring that cannot be done from inside the pod because the state of the cluster is not visible. For instance, when a pod or a resource goes through different states there might be the need to execute additional actions inside some pods to ensure the proper function of the application as a whole.

3.4 Cassandra

Cassandra is a distributed open-source database management system. It is designed to handle very large amounts of data spread across many servers, offering high availability and no single point of failure. It is a NoSQL solution and was initially developed by Facebook to support the Inbox Search feature until 2010 (Avinash Lakshman, 2014). Its main features are:

Decentralized:

Every node in the cluster has the same role. There is no single point of failure. Data is distributed across the cluster (so each node contains different data), but there is no master as every node can service any request.

Supports replication and multi data center replication:

Replication strategies are configurable. Cassandra is designed as a distributed system, for deployment of large numbers of nodes across multiple data centers. Key features of Cassandra's distributed architecture are specifically tailored for multiple-data center deployment, for redundancy, for failover and disaster recovery.

Scalability:

Read and write throughput both increase linearly as new machines are added, with no downtime or interruption to applications.

Fault-tolerant:

Data is automatically replicated to multiple nodes for fault-tolerance. Replication across multiple data centers is supported. Failed nodes can be replaced with no downtime.

Tunable consistency:

Writes and reads offer a tunable level of consistency, all the way from "writes never fail" to "block for all replicas to be readable", with the quorum level in the middle.

MapReduce support:

Cassandra has Hadoop integration, with MapReduce support. There is support also for Apache Pig and Apache Hive.

Query language:

Although it is NoSQL database at its core, CQL (Cassandra Query Language) was introduced, an SQL-like alternative to the traditional RPC interface.

3.4.1 Data Model

Column:

It is the atomic unit of information in Cassandra and is represented in the form *name: value.*

Super Column:

Super Columns group together columns and provide a common name. In that way more complex data structures can be modeled inside Cassandra.

Rows:

Rows are the uniquely identifiable data stored inside Cassandra. They group together column and super column values and link them with a unique key. Cassandra performs queries based on this unique key.

Column Families:

Column Families are analogous with a Relational Database Table. They group together keyed rows that consist of similar columns and super columns. The core difference with relational databases is however, that this is not a strong restriction and the schema of column families is not predefined or strictly the same. The user is free to include as many of the columns and super column values as they wish for each row. Column families just provide an abstraction to group together rows that are likely to be queried together, in order to improve performance.

Keyspaces:

Keyspaces are the highest level of information representation in a Cassandra cluster. Each column family belongs to exactly one keyspace. Also, the consistency levels and replication factor are defined on a per keyspace basis, meaning that different keyspaces can have varying consistency levels and replication strategies inside the same cluster.

(Featherston, 2010)

3.4.2 Cluster Architecture

A Cassandra cluster works with a Peer to Peer (P2P) architecture, meaning that every node is connected to all other nodes. Also, every node is aware of the data distribution on all other nodes. As a result, every node is capable of serving clients and perform all database operations. All of the above contribute to the fact that there is no single point of failure in a Cassandra cluster and this promotes increased fault tolerance. Also, performance scales almost linearly with the amount of nodes available in the cluster.

When we use the terminology of nodes, we refer to the physical of virtual machines that the cluster comprises of. In Cassandra there is also the term of virtual nodes. According to the data model of Cassandra, all data rows are uniquely identified by a key. The keys are hashed in order to form tokens, that are integer values in the range $[-2^{63}+1, 2^{63}-1]$. The resulting token range offers a one-to-many mapping of the data rows to the tokens. We can conceptualize the range of tokens as an ordered ring. In the case of a single physical node all of the information (token range) is stored in the same node, but as more nodes are added

the token range has to be distributed among nodes, so that each node is responsible for a part of the cluster data. To simplify token calculation and data distribution challenges, Cassandra divides the token range to a large number of virtual nodes. Then each physical node is assigned an equal number of virtual nodes at random, to ensure uniform data distribution. When new nodes enter or existing nodes exit the cluster, the virtual nodes are redistributed among the new set of physical nodes, instead of recalculating the token range for each physical node.

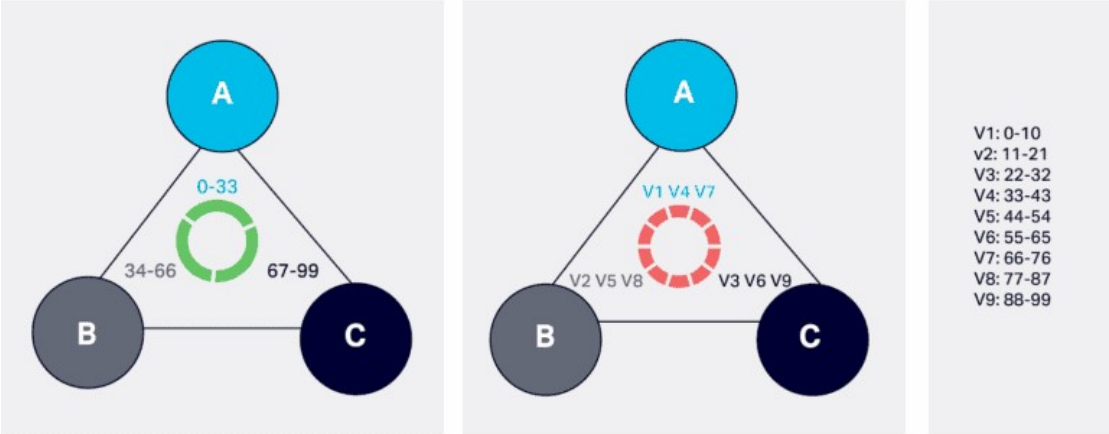

Figure 14: Virtual Node Distribution Example

The Cassandra cluster is also logically organized in racks and datacenters. In physical datacenters a rack is a group bare-metal servers sharing resources like power supply, networking etc. In the same manner racks inside the Cassandra cluster form a group of nodes that potentially can fail altogether. When multiple racks are available in a cluster, Cassandra opts to distribute data replicas to different racks. In that way, it guarantees higher availability and fault tolerance, because it is certain that more than one nodes will be queried each time and that all information is replicated in more than one physical nodes. A group of racks can be grouped to form a datacenter. Dividing the cluster to datacenters helps mitigate the impact of different workloads to one another, by assigning data of different workloads to reside inside different datacenters. Datacenters can have different consistency levels and can be scaled independently and simultaneously, further isolating the performance interference.

Cassandra offers tunable consistency levels. The consistency level options are a tradeoff between Consistency and Availability in accordance with the CAP theorem (Simon, 2000). The higher the consistency levels chosen, the more request acknowledgments have to be received to complete a query which leads to higher latency and limits availability. In general, read operations have significantly higher latency than write operations. The reason behind this is how data storage is implemented. Writes in Cassandra are immutable. Every new record is written in a read-ahead log. The on-disk structure is called SSTable. Updates or deletes on an existing record, create additional entries to the log with new version of the record. When Cassandra performs a write operation, it only waits for confirmation that the record has been committed from the nodes responsible for the key, whereas during read operations the node must consolidate all versions of a stored record and retrieve the most recent. Read operations also perform minor compactions and consistency checks, by removing deleted entries if they encounter them during version control and updating inconsistent entries with the latest value of a record.

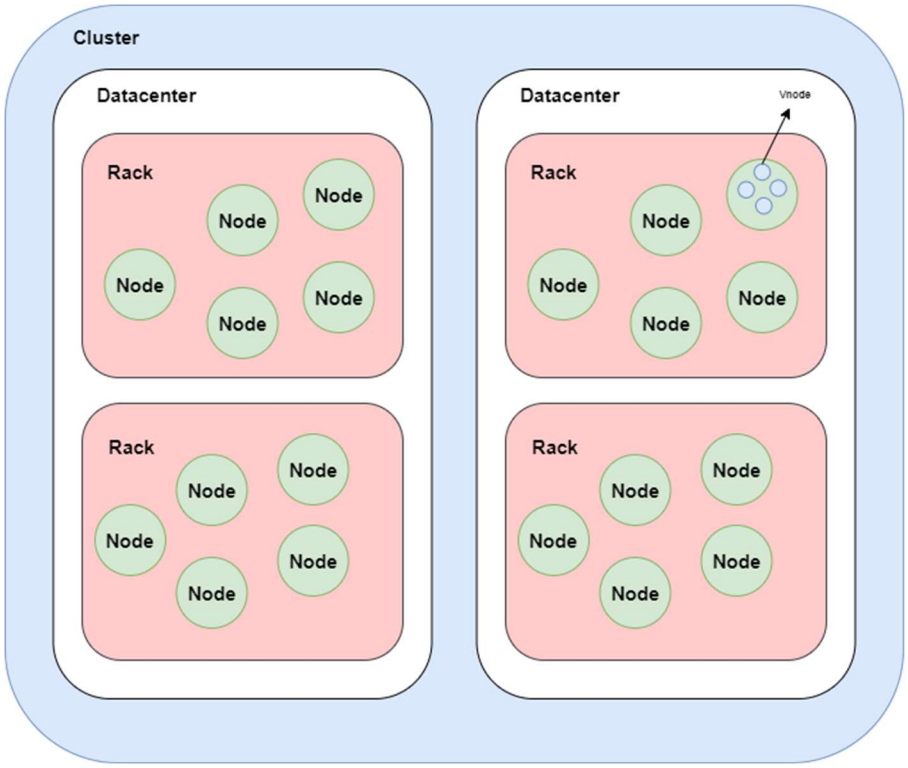

Figure 15: Logical Representation of a Cassandra Cluster

One final set of operations that are crucial for the consistency of data are *anti-entropy mechanisms.* As we mentioned before, reads perform some partial anti-entropy functions. They are however opportunistic, meaning that not all read operations perform anti-entropy checks to avoid performance decrease. Another anti-entropy mechanism is hinted handoffs. When a node becomes briefly unavailable, all replication information that should be stored in the node, is stored in a peer node instead. When the unavailable node reenters the cluster, it receives this information (called “hints”) from the peer node and catches up with the latest state of the cluster. Obviously, there are time and size limitations for hinted handoffs and consequently it cannot be considered a primary anti-entropy mechanism. The most reliable tool for anti-entropy are repairs. Repairs create special data structures called Merkel-trees that hash existing data entries in a replica and are then distributed to other nodes to detect inconsistencies. Finally, the latest version of all entries is streamed between nodes. Although, partial repairs are performed sporadically during low traffic periods, a full repair has to be scheduled manually because it is a computationally intensive operation that might make the cluster temporarily unavailable.

3.5 K8ssandra

Figure 16: K8ssandra Logo

When Kubernetes was initially launched, its primary focus was to support stateless applications. Although, several organizations have migrated their cloud applications in the Kubernetes ecosystem, the initially limited support for stateful applications resulted in computing infrastructure to mature at a greater rate than data infrastructure inside this ecosystem. Before K8ssandra, the most common practice was to deploy computing resources inside the Kubernetes cluster and deploy an external database and connect it with the cluster. This approach is suboptimal because at least to stacks of monitoring are needed to ensure the health of the application, development productivity is limited because competency is needed in at least two environments and each one can be a bottleneck for the other and finally cloud infrastructure is not utilized optimally leading to greater costs (K8ssandra, 2022).

The obvious solution is to migrate the data infrastructure inside the cluster. In that way the application can be monitored more effectively and scaled with reduced cost. K8ssandra offers a flexible deployment of a Cassandra cluster inside Kubernetes that is suitable both for developers looking for a scalable data solution as well as developers looking for a simple and ready to run solution using preconfigured parameters. The K8ssandra deployment provides additional tools for monitoring, repairing and restoring data except for the Cassandra implementation. The core components of a K8ssandra installation are the k8ssandra-operator, stargate, cass-operator, medusa, reaper and a monitoring service that uses Prometheus. For the scope of this thesis, we will only use and analyze the components used for deploying and monitoring a single Cassandra cluster, namely cass-operator, stargate and Prometheus.

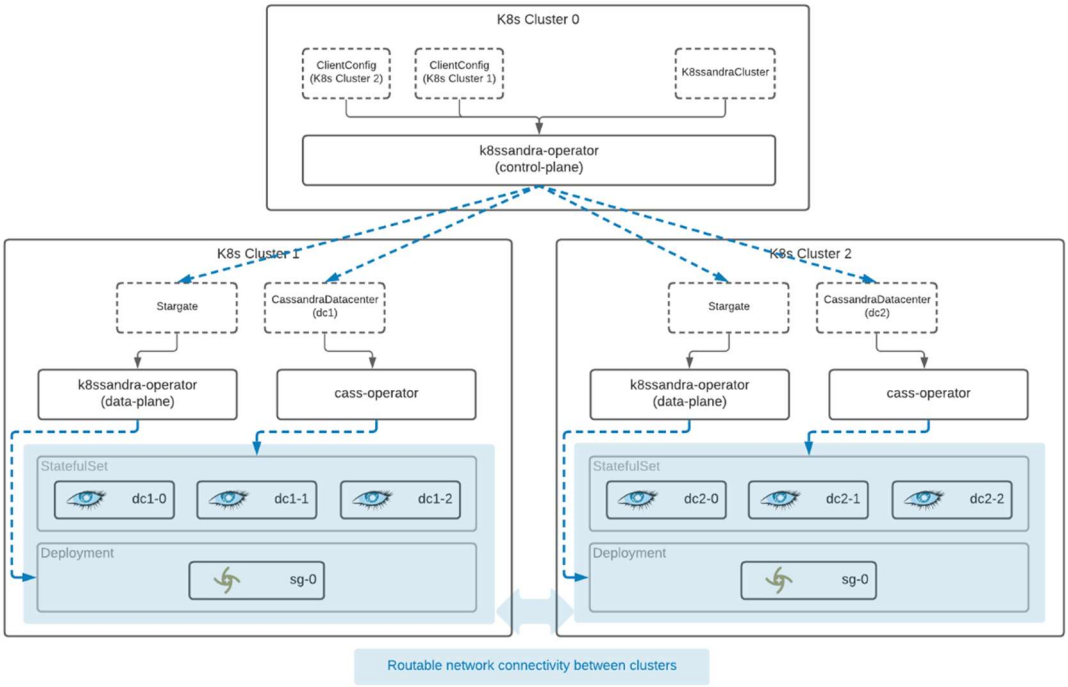

Figure 17: K8ssandra overview

3.5.1 Components

Cass-operator:

Initially, the cass-operator was a standalone project that was used to deploy a Cassandra cluster inside a Kubernetes cluster. It provided limited monitoring capabilities and also orchestrated node bootstrapping and cluster scaling. Soon after the release of the K8ssandra project the cass-operator was migrated inside of it and now its development and maintenance are transferred to the K8ssandra community.

In terms of Kubernetes resources cass-operator is a custom controller that monitors all the resources that are used to deploy the Cassandra cluster. On deployment, it receives a yaml file that contains parameters about the Cassandra cluster. This approach allows for great flexibility in terms of configuration, because every component of the Cassandra cluster can be configured to great extends. For example, we can explicitly define resource limits for each individual component, configure networking connections between components and also create privacy settings to increase security of the cluster. We can also opt to not install a specific component of K8ssandra default installation if it does not suit the needs of our application. After deploying the resource, cass-operator creates a StatefulSet with size equal to the declared size of our Cassandra cluster. It then monitors the bootstrapping of nodes, ensuring that only one node enters the cluster at a time to ensure stable deployment. Since the operator is aware of the network topology in term of Kubernetes resources, it organizes nodes on racks in a way that ensures that there is the least probability if any at all, of data being lost due to node failure. More specifically, it opts to organize nodes running on the same Kubernetes node in the same rack. In that way if the node becomes unavailable, data are not lost if the replication factor is greater than one, due to the replication protocol of Cassandra to store replicas in different racks.

Cass-operator also creates several services to allow network connections between Cassandra nodes. First it creates a seed service so that nodes entering the cluster can finds seed nodes in order to bootstrap. The amount of seed nodes varies, from three per datacenter to one per logical rack and seed nodes are automatically replaced by cass-operator in case a seed node is removed from the cluster or becomes unavailable. It also creates an all-pod service that allows Cassandra nodes to connect with each other once they have entered the cluster. Again, endpoints are updated by cass-operator as nodes enter or leave the cluster.

Overall, the cass-operator is the most reliable point of observation of the Cassandra cluster. The clustered state can be monitored by the yaml of json file that can be produced by Kubernetes at any time and describes the current state of the cass-operator resource and consequently the cluster. This is the part of the deployment that the agent we created uses to make changes to the cluster and receive updates on when these changes are applied.

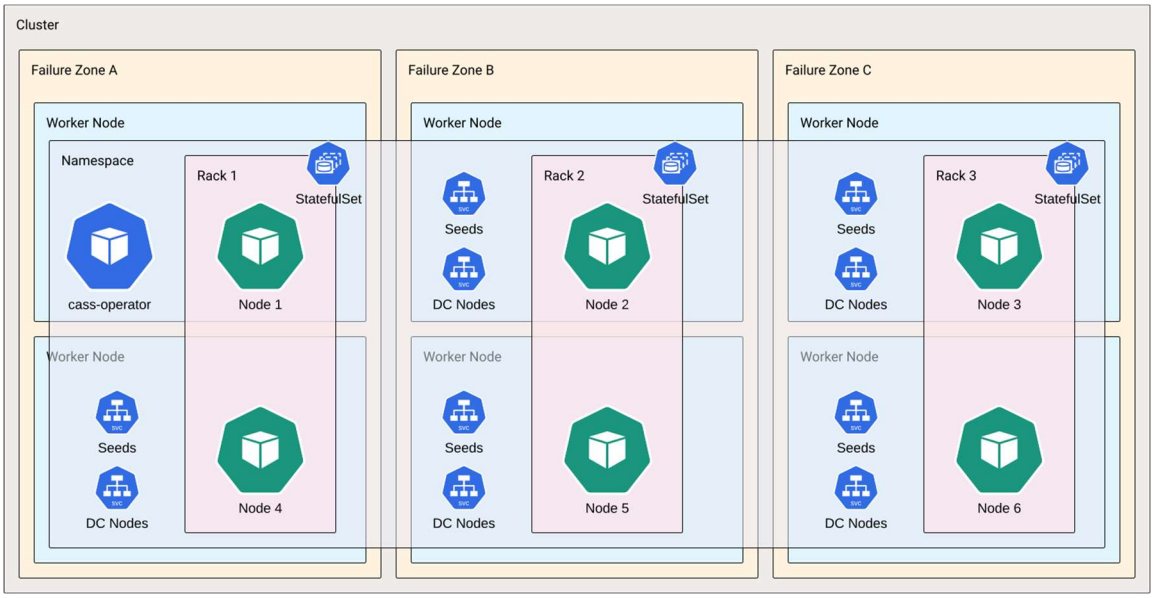

Figure 18: K8ssandra Cluster Overview

Cassandra Pods:

Cassandra pods consist of three containers. A Cassandra container that runs the Cassandra image, a logger container that stores log information about Cassandra operation within the pod and is useful for troubleshooting and an init-container that handles configuration settings received from cass-operator as the pod is initialized. The Cassandra container does not start the Cassandra JVM process immediately. It only starts a management REST API used by cass-operator to send lifecycle operations to the container (Start, Drain, Stop etc.). Then when the cass-operator decides it is safe it triggers the initialization of Cassandra and the container starts running. As we mentioned, Cassandra pods are part of a StatefulSet monitoring the whole Cassandra cluster. In that way, every pod is unique and if a pod fails it is restarted with the same identifiers. Also, since there is the obvious need for persistent storage, a PersistentVolumeClaim is made when the pod is created that binds the pod to a persistent volume. A pod can only start running if a suitable volume is bound to it. The specifications about the volume features are declared in the

values of the cass-operator and transferred to the StatefulSet and each individual pod (K8ssandra, 2022).

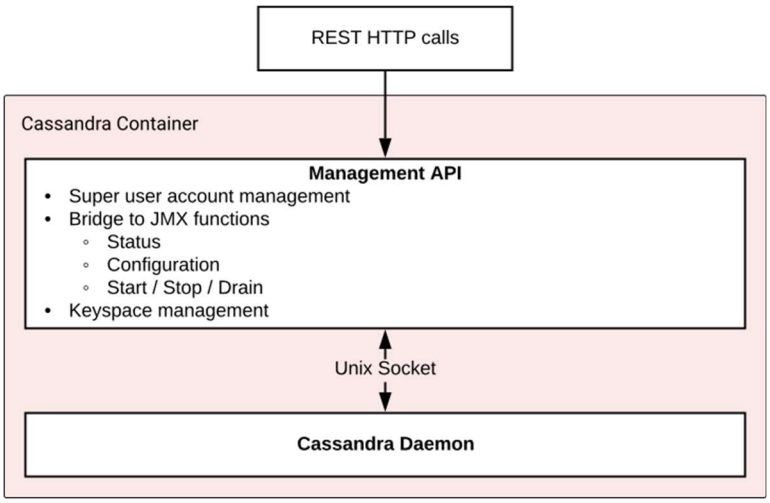

Figure 19: Cassandra Pod

Stargate:

We can say that the K8ssandra installation is not a pure form of Cassandra installation, as it would be deployed in a set of virtual machines and the Stargate pods are the reason for this. The stargate project is an open-source data gateway that provides a layer of abstraction between a database and client requests. In that way, client request can be configured to suit the application needs and then be transformed to fit the database interface.

The motivation behind the integration of stargate to the K8ssandra project is due to two main reasons. The first is that although Cassandra offers the CQL interface to create SQL-like queries, such an approach would be suitable only for applications that already used a Cassandra solution. In the general case, especially in a microservices environment, the common approach is that different components interact using API requests. Stargate satisfies this need for a more uniform way to interact with the Cassandra cluster.

Another purpose served by Stargate nodes is to perform a part of the coordination of data requests towards the cluster. Each stargate pod runs a stateless instance of Cassandra inside it. This means that when a new instance is created it bootstraps in the cluster and considered a part of the ring but it does not receive any data. It only receives and preserves keyspace distribution and topology information, so that it is aware of the data distribution. When requests arrive, it is responsible for forwarding them to the appropriate Cassandra nodes and also verify the returned responses according to the consistency protocols of Cassandra. In that way, Stargate pods alleviate some of the computational pressure from Cassandra since the computations for serving requests is distinguished from computations to retrieve data. The two parts of the clusters can be scaled independently which leads to additional robustness.

Although Stargate is considered a core component of the K8ssandra cluster and is configured for optimal performance it is not mandatory to include it in the installation. A completely viable alternative is to disable the Stargate pods during installation and deploy

only the Cassandra pods. In that case however, the user must manually create an additional service to expose Cassandra to the rest or the cluster or outside of the cluster, depending on the business needs.

Monitoring:

The monitoring stack that is installed with K8ssandra is based on two components, Prometheus and Grafana. Prometheus consists of a database and a server. The server periodically scrapes metrics from available resources (pods or nodes) by making API requests to the endpoints provided to it and then stores the data locally in the form of time series. Grafana is used to visualize the data stored in Prometheus by performing queries on them and then presenting them in diagrams via a web browser. For the scope of this thesis, we will only analyze Prometheus, which is used by our agent to extract metrics that describe the state of the cluster.

Prometheus stores all data in a key-value time series format. This means that all raw information is stored as a pair of value and the timestamp that this value was observed. Then data are organized in metrics using distinct names as keys. Prometheus also allows data to have dimensionality by using additional labels. In that way, datapoints belonging to the same metric can be distinguished further, by using the set of labels each one has. This allows for a more expressive representation of information when performing queries on data. Although all information is stored as pairs of values and timestamps, Prometheus offers some additional metric types through the client libraries that are installed to application interacting with Prometheus. These are counts, gauges, histograms and summaries. Counts are used for metrics that are monotonically increasing, for example the requests served by an http server. Gauges are used to represent values that can increase or decrease over time, for example memory or CPU usage. Histograms, samples observed values during a configurable timeframe and then counts them in configurable buckets. Then the user can query on that metric based on quantile percentages. This can be useful for metrics like latency where we are usually interested for the latency experienced by a specific percentage of users. The summary metric is similar to a histogram but it also sums values instead of counting them and offers the functionality of querying this metric using percentiles. This additional metrics extend the queries we can perform on these metrics, increasing the information we can extract from source information (Prometheus, 2022).

As we mentioned, Prometheus supports its own query language PromQL. PromQL can support 4 types of queries. The first two are string and numeric literals that offer additional functionality when creating complex queries, for example to subtract a constant from a value. Instant vectors return a set of values from a metric, that correspond to the latest timestamped recorded. Although only one metric is queried, more than one values can be returned based on the filter o labels applied. It is useful to note that PromQL supports this application of regular expressions on labels so that a more generic matching can be created. Finally range vectors support the same functionality as instant vectors but also a range is defined and all values with timestamps in range [present -range, present] are returned. The user can also apply aggregation functions on the returned values (sum, avg, rate etc.) and numeric operators. Using the above language, the raw data can be

transformed to very expressive timeseries that can describe metrics ranging from resource usage observation to very complex and application specific metrics.

Prometheus can discover the endpoints to scrape metrics from either by being provided the endpoints explicitly or by being provided a service endpoint and then discovering all endpoints that are connected to this service. This is very similar to the pod connectivity in Kubernetes which makes the integration of Prometheus inside the Kubernetes cluster very natural. We have to mention here that Prometheus have some disadvantages too. It provides no shared storage architecture, meaning each Prometheus server is autonomous. This of course allows increased availability and shorter query latency but provides no fault tolerance in case a server becomes permanently unavailable. For this reason, Prometheus is more suitable for short term monitoring in order to create graphs or alerts based on present observations and also provides a very expressive query language. If the primary goal is to create long-term event-logging then it is better to seek an alternative. That said, in our case Prometheus perfectly suits our needs because it provides low latency and high availability, so our agent has negligible downtime waiting for metrics.

So far, we only described how Prometheus handles metrics once they are exported by the various components of an application. This implies that each component is responsible for exporting a set of metrics at a specific interval. In the case of K8ssandra, each pod of the Cassandra includes a container that runs Metrics Collector for Apache Cassandra as a Java agent. This is an open-source metrics collector built on the collectd, a well known and supported metrics collector for Unix systems. Every metrics agent then exposes the metrics at the 9103 port to be collected by Prometheus.

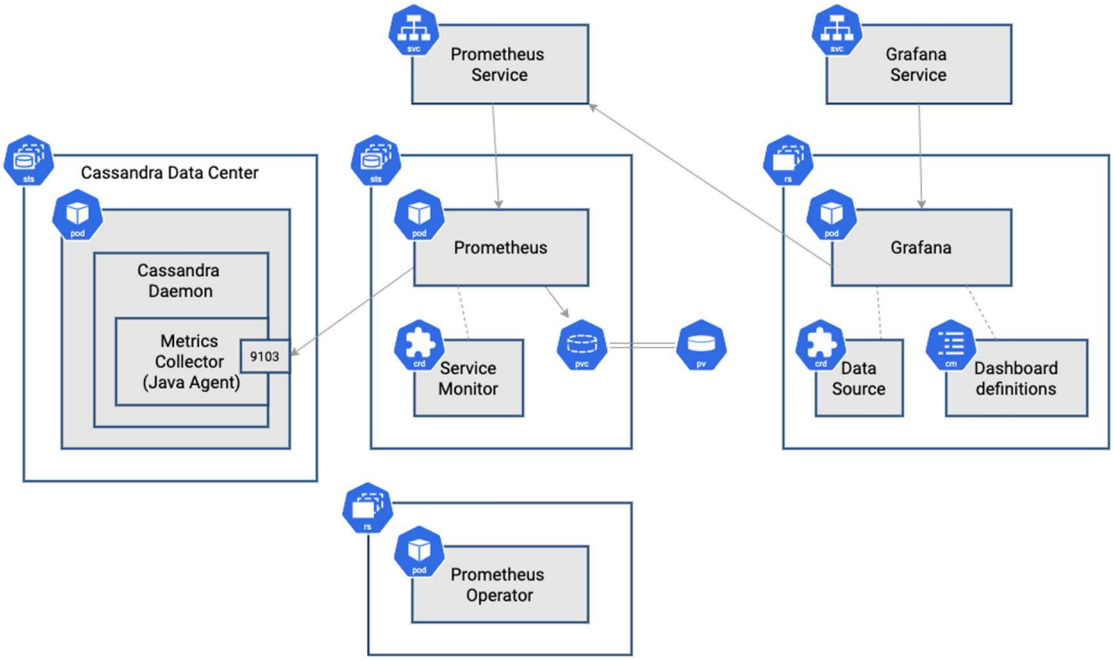

Figure 20: Monitoring Stack Overview

3.6 Reinforcement Learning

3.6.1 Introduction

Artificial Intelligence refers to the development of machines that imitate human intelligence. In that way, machines can perform functions as learning and problem solving in a similar manner as the human brain performs these tasks. Artificial intelligence can be divided in two main domains, Artificial General Intelligence (AGI) and Artificial Narrow Intelligence (ANI). AGI describes the theoretical of a single agent being able to apply artificial intelligence to any domain and solve any problem. ANI describes any agent that is able to perform a single task that is narrowly defined and structured task. These systems are designed to perform a single task and outperform humans in that task using a set of rules, parameters and contexts to be trained with. This type of Artificial Intelligence is also referred as "weak AI". Although AGI agents are still not achievable by current research, narrow AI agents have been created for a plethora of tasks.

ANI agents are becoming increasingly popular because as mentioned before, they can outperform humans in certain tasks and consequently increase efficiency but also Artificial Intelligence systems have been able to solve problems that traditional algorithmic methods have not been able to due to their complexity. It is one of the main reasons that Artificial Intelligence is becoming increasingly popular, because it is a promising alternative to problem solving.

3.6.2 Machine Learning

Machine Learning (ML) is a subdomain of Artificial Intelligence that has been gaining increased attention since 2010. This domain studies algorithms and models that are able to learn from data in order to solve problems, similarly to humans learning through examples. The most commonly accepted definition of Machine Learning was given by Tom M. Michell: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks T, as measured by P, improves with experience" (Michell, 1997). From the above definition, we can understand that ML aims to create algorithms that focus on providing the way to utilize experiences efficiently to reach the problem solution, rather than providing strictly defined instructions on how to solve a problem.

Machine Learning can be categorized, according to the tasks it is implemented to and the nature of data used as experiences, in three categories:

Supervised Learning:

In this category the agent is provided with a dataset of labeled data, where each label represents the class that each datapoint belongs to. Then the agent, processes the data and learns the correlations between datapoint features and labels. After sufficient training has taken place, the agent then can classify unseen datapoints belonging to the classes represented by the labels and classify them correctly. Of course, for a successful training the data must be correctly labeled and the dataset must provide a complete and unbiased representation of the distribution of data points.

Unsupervised Learning:

In this category the agent takes unlabeled data as input. Its task is to cluster data to groups that display similarities, without the human intervention of labeling the data. After the data has been processed the agent can be used to either produce imaginative datapoints that reflect the distribution of the experience data or categorize unseen data to one of the categories it produced during training.

Reinforcement Learning:

This type of agents takes as input a vector the represents a state in the environment it is deployed to interact with, a set of actions and a reward function that provides a numerical value to each state. The agent then performs actions and observes the rewards of the states its actions lead to. Gradually it tries to converge to a policy and choose actions that maximize the overall reward it is expected to gain. These types of agents are useful for playing games and robotics applications.

3.6.3 Reinforcement Learning

As we mentioned before the goal of Reinforcement learning is to maximize the overall rewards an agent can gain by interacting with an environment. The maximization infers the need for an optimality criterion. More specifically, we must define how the agent evaluates the rewards in order to modify its behavior. There are three approaches that are most commonly used for this purpose.

The *finite-horizon* model, where the agent is expected to maximize the expected rewards for a finite number of steps h.

$$E\left[\sum_{t=0}^{h} r_t\right]$$

This approach is suitable when the agent can only perform a fixed number of steps in the environment, for example in a game that ends after a fixed number of turns. For this approach we can either opt for a *fixed or receding horizon*. Fixed horizon means that the agent does not have a stationary policy. On the first step it is going to choose the *h-step optimal action.* On the second step the (h-1)-step optional action and so on. In the receding horizon case, the agent has a fixed policy and it just alters the *h* number of steps ahead it will look to decide on an action. This approach can be problematic since it limits the number of steps ahead the agent must consider and this information is not always known beforehand.

Another approach is the *average-reward model,* in which the agent is supposed to maximize the long-term average reward:

$$\lim_{h\to\infty} E\left(\frac{1}{h}\sum_{t=0}^{h} r_t\right)$$

Again, this approach is problematic because it does not distinguish between a policy that prioritizes short term large rewards to and agent that prioritizes long term larger rewards. To avoid these two problematic cases, an alternative form of reward model is used the *infinite-horizon discounted model.* The equation to maximize is (where 0 ≤ γ ≤ 1):

$$E\left(\sum_{t=0}^{\infty} \gamma^t r_t\right)$$

In this case, the long-term rewards are considered but they are discounted by a factor γ for each additional step it takes to receive them. The discount factor serves more than one purposes. First of all, it is effectively setting a frame of effective rewards. The smaller the discount factor is, the faster future rewards diminish to zero and are not affecting the result. It also conveniently ensures that the sum will converge to a finite number given that the rewards are themselves a finite number. This mathematical tractability is the dominant reason this type of reward is has received the most attention (L. P. Kaelbling, 1996).

3.6.4 Markov Decision Process

Markov Models are stochastic models created to describe non deterministic processes. Markov models are described by a number of states and a transition probability between states. One of the properties that make them widely adopted, is that they are *memoryless,* meaning that the behavior of the system only depends on the current state. If our problem can be transformed into a Markov Model, then by observing the current state we have sufficient information to decide on the next states. A *Markov decision process* is an extension of the Markov chain, where actions are added to each state and rewards for executing these actions. These processes a suitable to describe reinforcement learning problems because they are expressive enough to describe the process of an agent taking deliberate actions and not only transition to states stochastically and also receive rewards. Using these models, we can calculate optimal policies for agents.

A Markovian Decision Process consists of the following:

- A set of states S
- A set of actions A
- A reward function $R: S \times A \rightarrow \mathbb{R}$
- A transition function $S \times A \times S \rightarrow \Pi(S)$

Before discussing algorithms for finding MDP models, we will first explore techniques for finding an optimal policy, given that we already have the MDP model available. We call *optimal value* of a state, the expected infinite discounted reward the agent will gain if it starts from this state and executes the optimal policy.

$$V^*_{(s)} = \max E\left(\sum_{t=0}^{\infty} \gamma^t r_t\right)$$

This optimal value is unique and is the solution to the equations:

$$V^*_{(s)} = \max\left(R(s,a) + \gamma \sum_s T(s,a,s')V^*(s')\right) \qquad (1)$$

$$\pi^*_{(s)} = \arg\max\left(R(s,a) + \gamma \sum_s T(s,a,s')V^*(s')\right) \qquad (2)$$

The optimal values can be calculated using the *value iteration* algorithm. At every step of the algorithm, we iterate over all states and all actions and calculate the next estimation about function V. Then we update our next step estimation as the max calculated value for each state. This algorithm is shown to converge to the optimal policy (Bellman, 1957). There is no obvious termination criterion of the algorithm. It has been proven however, that if the difference between two successive value functions is less than ϵ then value of the greedy policy differs from the optimal value by no more than $2\epsilon\gamma \,/\, (1-\gamma)$. This provides an effective stopping criterion for the algorithm. By greedy policy we mean taking the max value at each iteration as an update for the value function. It is apparent that the policy can be arbitrarily close to the optimal, depending on the value of ϵ we choose.

3.6.5 Exploration Strategy

In a reinforcement learning context, since the only way the agent learns the environment is by performing actions, it is necessary that a strategy is implemented that forces the agent to perform all actions available. If that is not the case, the agent can get locked in attempting the first action that yielded a positive result again and again, missing out on opportunities to obtain better rewards. It is no surprise therefore that the greedy strategy of always choosing the most optimistic option is not very effective in practice.
To overcome this difficulty, a number of techniques have been proposed. One such useful heuristic is known as optimism in the face of uncertainty, in which actions are selected greedily, but strongly optimistic prior beliefs are put on their payoffs so that strong negative evidence is needed to eliminate an action from consideration. Of course, using this technique it is still possible to eliminate an optimal but unlucky action, but the risk can be made arbitrarily small.

Another simple approach to exploration strategies is to not always perform the best available action, but instead with a probability *e* perform an action at random. This probability can easily be adjusted throughout the life of the agent, so that at the start of the training where little is known about the system and a lot of exploration opportunities are available the probability of random actions is high, while as the agent learns the world and becomes more certain about its behavior the probability to perform random actions diminishes. This is strategy is known as *ε-greedy* strategy.

3.6.6 Learning an Optimal Policy

In the previous chapter when we started describing an MDP and how to calculate the optimal value function of a model, we assumed that we already had the model parameters available to us. This is not always the case though. In many problems we do not have prior knowledge of the model that describes the system the agent interacts with. Moreover, even if we do have such knowledge, it is not always desirable to provide this knowledge to the agent because it inserts some form of bias based on our perception of this system. It is preferable that the agent is able to learn the dynamics of the system on its own and still be able to find an optimal policy.

The agent interacts with the environment and observes the states these actions lead to and the rewards it receives. This is the only means by which the agent can observe and consequently gain information about the environment. There are two approaches to reach an optimal policy:

- *Model Free Systems:* Learn the action controller without learning a model of the environment.
- *Model Based Systems:* Learn a model of the environment and then derive the controller.

(L. P. Kaelbling, 1996)

The question of which approach is better still remains a bone of contention in the academic community. The only thing that is certain is that both approaches have been used to provide optimal solutions for different problems. For the scope of this thesis, we will attempt to solve the proposed problem using model free systems and specifically Q Learning so we are not going to analyze the model based methods further.

3.6.7 Q Learning

Q Learning is a widely adopted method that has been able to solve many reinforcement learning problems due to the ease of implementation compared to other methods. To understand Q Learning we must define the function Q^* as a function of states and actions. Q represents the expected discounted reinforcement of taking action a in state s and then continuing to make optimal options of action for the successive states. Considering the above $V^*_{(s)}$ is the value of s assuming we perform the optimal action from step one so essentially $V^*(s) = \max_a Q^*(s,a)$. Using this equation along with equation (1) the Q function can be written recursively as:

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s \in S} T(s,a,s') \max_{a'} Q^*(s',a')$$

$Q^*(s,a)$ also provides us with the optimal policy $\pi^*_{(s)}$ as $\pi^*_{(s)} = \arg\max Q^*(s,a)$ using equation (2) and substituting Q for V. The recursive definition of Q and the fact that it provides an explicit way of deciding on an action on each step allows us to estimate Q values

online and also use them to define the optimal policy, by taking the maximum Q value for the current state at each step. The Q Learning rule is:

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$$

Where $\langle s, a, r, s' \rangle$ is an experience tuple. It is proven that if every action is executed an infinite number of times in each state and α is decayed appropriately then the Q values will converge with probability 1 to Q^*. The only problem of trading an agent with this method is that there is not clear exploitation vs exploration strategy. In practical applications the ε-greedy strategy is adopted where ε decreases sufficiently slow so that the agent can experience all state-action pairs enough times to be able to converge (L. P. Kaelbling, 1996).

3.7 Deep Reinforcement Learning

Reinforcement learning algorithms have existed more than two decades in academic literature. Nevertheless, their applications were limited. The main problem is that every reinforcement learning algorithm needs at some point an approximator function to estimate the value function of the policy distribution of the actions it performs. Explicitly defining such a function is impossible and using simple approximators, like linear ones, greatly impacts performance and limits the scope of the problems it can be applied to. Deep neural networks provided an effective non linear approximator that is able to fit to very high dimensionality non-linear functions given enough samples and time to converge. The increase in attention neural networks have received since 2010 has also revitalized Reinforcement Learning applications that leverage this new powerful tool to learn the estimators they need.

3.7.1 Neural Networks

Neural Networks imitate the function of brain neurons. Each neuron receives a number of input electrical signals by other neurons it is connected to and outputs an activation signal. The intensity of the output signal depends on the mixture of the input signals but not in a linear way. Artificial Neural Networks (ANN) work in the same manner. In their simplest form ANNs consist of multiple layers of neurons. The output of the neurons of each layer is then forwarded as input to each layer. The initial layer takes as input the datapoints of the dataset and is called *input layer*. Similarly, the final layer produces the output that is task specific and is called *output layer.* The rest layers are called *hidden layers.*

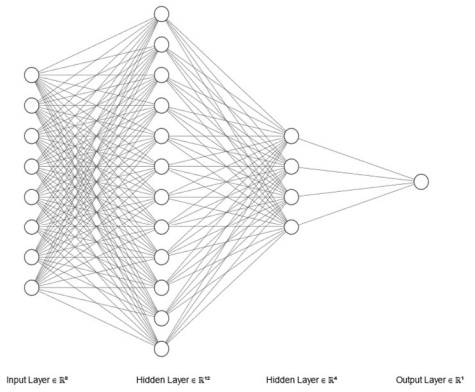

Figure 21 : Example of a Fully Connected NN

The fully connected ANN is the simplest form of neural network architecture. Its approximation capabilities extending greatly. However, as the layers increase, so do the learnable parameters and this introduces instability in the problem and the need for

exponentially more samples, for the network to approximate the desired function. In his study Hughes proved that for a fixed number of training samples, the performance of a model initially improves with the addition of trainable parameters but after a threshold it rapidly deteriorates. The curse of dimensionality is also known as the Hughes phenomenon (Hughes, 1968).

3.7.2 Artificial Neuron Model

As we mentioned, Artificial Neural Networks consist of several neurons that are organized in layers. Each layer receives the outputs of the previous layers as input and each neuron creates a new output using them. In essence, the neuron is the atomic computational unit of an ANN. In its generic form an artificial neuron takes m input signals from the previous layer. Each signal – called synapse – is multiplied by a weight and then all the multiplied synapses are summed. The result is modified by a non-linear function and the result of the function is the output of the neuron. In most applications, it is also useful to add an additional constant to the sum of the weights, called *bias.*

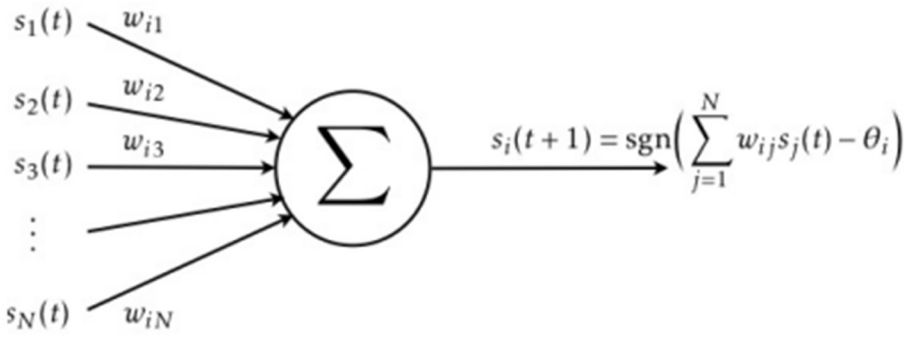

Figure 22: Artificial Neuron using sgn(x) as activation function

The learnable parameters of each neuron are the weights and the bias. The set of all learnable parameters of all the neurons, are the learnable parameters of the network.

3.7.3 Activation Functions

Activation functions are the components of the network that introduce non-linearity and are thus vital for the approximation abilities of an ANN. The simplest activation function is the $sgn(x)$. Its range is $[-1,1]$ and essentially works as a threshold of activation.

Although, this function can be useful in some applications it is problematic for two reasons. Firstly, due to the sudden change happening around 0, it can introduce fluctuations to the output of the neuron when the sum of the inputs takes values around 0. Secondly, the derivative of $sgn(x)$ with respect to any of the input signals is $\delta(x)$. This function, is not ideal for applying the training algorithms that update the weights of the network. That is because these algorithms need to first order derivative of the activation function and $\delta(x)$ is mostly useful for theoretical calculations, not arithmetic. A better approach is to use the $\tanh(x)$ function instead. It is very similar to the $sgn(x)$ but it is a continuous function and so is its derivative, which leads to better weight training.

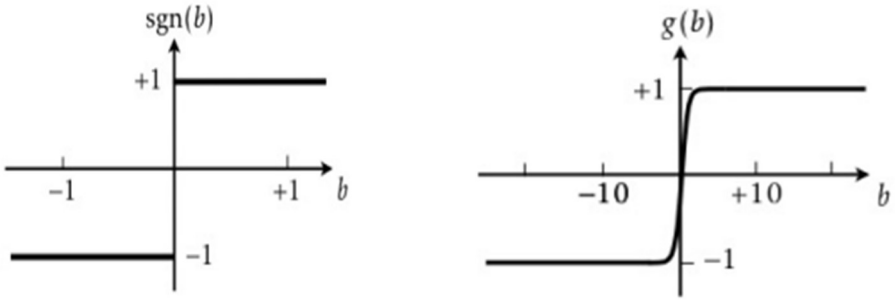

Figure 23: signum function and sigmoid function

Although $\tanh(x)$ solves the problem of values around 0, it still is problematic in terms of weight updating. The reason is that the first order derivative of $\tanh(x)$ is $\tanh'(x) = \tanh(x)(1 - \tanh(x))$. We can observe that for any value of x that $\tanh(x)$ is less than 1, so does the derivative. As a result, as values less than 1 are propagated throughout the network the resulting gradients are diminishing rapidly to 0, leading to negligible updates of the weight parameters. This problem is known as *vanishing gradients* and is quite severe as the size of the network grows, because bigger networks require training with smaller arithmetic values to ensure training stability.

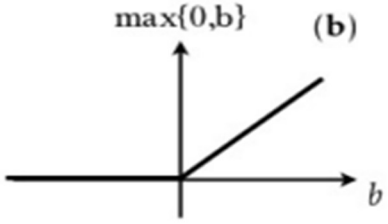

Figure 24: ReLU function

An alternative choice of activation function, is the *rectified linear units or ReLU function.* This function is defined as $r(x) = \max(0, x)$ and solves the diminishing gradients problem while still remaining a continuous function. Deep neural networks using this activation function have been able to reach better training performance levels and for that reason it is the most commonly used activation function in modern applications. ReLU NNs are useful for RBMs (Nair, 2010) (Maas, 2013), outperformed sigmoidal activation functions

in deep NNs (Glorot X. B., 2011), and helped to obtain best results on several benchmark problems across multiple domains (Krizhevsky, 2012) (Dahl, 2012).

3.7.4 Weight Initialization

A topic in Deep Neural Network training that receives less attention than it should is weight initialization. The authors Boris Hanin and David Rolnick provide rigorous proof about how the variance of the distribution from which the initial weights are drawn from, determines whether a network will be able to train on a dataset and also provide criteria on if a deeper architecture will still be able to train on the same data compared to a shallower one, again regarding the initial weight distribution (Rolnick, 2018). Analyzing the paper's findings further is out of the scope of this thesis but they provide evidence that weight initialization is crucial for successful training and its impact increases with network depth.

For the needs of this thesis let us consider the most common cases in weight initialization. First let us assume that all weights are initialized to 0 or any arbitrary constant. In that case, the derivative of the activation function with respect to any weight in the network is identical. Since the learning algorithms use the derivative of the activation function to calculate the update of each parameter, all updates will be identical as well. This will continue to happen for every iteration of learning, leading to a symmetric hidden architecture. This makes the network degrade to a linear approximator.

A better approach is to randomly initialize the values of the weights. In that way, the symmetry in the weight updates is broken. However, this approach although it works for shallow networks can still be problematic depending on the distribution and scale of the features. If initialized with high values, it is possible that the activation function will be saturated (meaning it will reach either the -1 or 1 state for bounded activation functions or very high numeric values in unbounded cases). If this happens, then the gradients provide no effective training for the neuron and no learning can occur for this neuron from that point onward. If we opt to initialize with values close to 0, then we can avoid saturation but the updates are very small arithmetic values, which decreases the convergence speed significantly.

To account for the problems of saturation and also provide faster convergence, more advanced initialization techniques were introduced. These techniques are driven by the observation that as an input is propagated through the hidden layers of the network, the impact of initial weights on the output is magnified with each additional layer. To avoid this phenomenon, it must be ensured that the variance of the input weights is the same for all layers of the network. This idea was used first for the Xavier initialization (Glorot X. B., 2010), were the variance of the weights is set in order to satisfy the uniform distribution:

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, -\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right] where\ n_j\ is\ the\ number\ of\ neurons\ of\ the\ j^{th} layer$$

This choice of initialization is not activation function agnostic and consequently the proof of constant weight variance is based on the assumption that the sigmoid activation function is used. In the case of ReLU activation function, a slightly different initialization satisfies the constraint for constant variance across layers, called He initialization (K. He, 2015):

$$W \sim N\left[0, \frac{2}{n^l}\right] where\ n^l\ is\ the\ number\ of\ neurons\ of\ the\ l^{th}\ layer$$

The proof of this initialization choice is straight forward. For a layer of the network:

$$y_i = W_i x_i + b_i$$
$$where\ y_i\ is\ the\ input\ to\ the\ activation\ funtionn\ at\ layer\ i,$$
$$W_i\ is\ the\ weight\ matrix\ of\ the\ layer\ at\ layer\ i,$$
$$x_i\ is\ the\ output\ of\ the\ previous\ layer\ x_i = f(y_{i-1})$$
$$b_i\ is\ the\ bias\ matrix\ of\ the\ layer\ i$$

We assume that the weights are mutually independent random variables with 0 mean and that the variances of x_i is constant so we can write that:

$$Var[y_i] = Var[W_i x_i] = Var[W_i]Var[x_i] + E[W_i]^2 Var[x_i] + Var[W_i]E[x_i]^2$$
$$= n * Var[W_i]E[{x_i}^2]$$

When the activation function is the ReLU then $E[{x_i}^2] = \frac{1}{2}Var[y_{i-1}]$ because:

$$E[x^2] = \int_{-\infty}^{+\infty} x^2 P(x)dx =$$
$$\int_{-\infty}^{+\infty} \max(0, y)^2\, P(y)dy =$$
$$\int_{0}^{+\infty} y^2 P(y)dy = \frac{1}{2}\int_{-\infty}^{+\infty} y^2 P(y)dy = \frac{1}{2}Var[y]$$

When we consider the above the variance of a single layer is:

$$Var[y_i] = \frac{1}{2} n_i Var[W_i]Var[y_{i-1}]$$

Finally, since the output of any layer is the input to the next the total variance of the network outputs is:

$$Var[y_L] = Var[y_1]\left(\prod_{l=2}^{L} \frac{1}{2} n_l Var[W_l]\right)$$

The requirement for constant variance can be satisfied if the product does not magnify or diminish the total variance. In other words, if every term of the product equals to 1 then we can satisfy our requirement. Setting every term of the product to 1 provides the weight distribution of He initialization.

3.7.4 Training

Parameter Updates/Optimization:

The goal of an ANN is to approximate the desired output for all training samples. To achieve that the agent is provided with a loss function $l(\hat{y}, y)$ that measures the cost of predicting $\hat{y}$ when the output is y. The aim is to minimize the function f that minimizes $l(f_w(x), y)$ averaged on all samples x. Ideally, we would like to average over the whole distribution *z* that is the underlying distribution of the samples *x.* But realistically we must settle for an average over the samples, since the distribution is unknown in the common case. The *empirical risk* $E_n(f) = \frac{1}{n}\sum_i^n l(f(x_i), y_i)$ is the measure of the training set performance and $E(f) = \int l(f(x), y)dP(z)$ is the generalization performance, meaning the expected performance on future unknown samples deriving from the same distribution. When f is sufficiently restrictive then it is justifiable to minimize the empirical instead of the expected risk (Vapnik, 1971).

A common approach to minimize the empirical risk is using *Gradient Decent.* Each iteration updates the weights of the network using the following rule.

$$w_{t+1} = w_t - \gamma \frac{1}{n} \sum_{i=1}^{n} \nabla_w Q(z_i, w_t) \text{ where } Q(z_i, w_t) = l(f_w(x), y)$$

The γ parameter is called learning rate and is a very important hyperparameter of machine learning as different values impact convergence times and overall convergence of a model greatly. Applying *Gradient Decent u*sing the above equation as is, is computationally inefficient in ANN applications because each data point is used individually leading to many calculations that must be executed sequentially. Also, the algorithm needs to preserve information about the datapoints already used and pick the next sample accordingly, a fact that makes this approach inapplicable for certain datasets. A better approach is to use *Stochastic Gradient Decent.* In this case at every iteration only a randomly selected subset of the samples is used to update the weights. It is proven that if the learning rate satisfies the conditions $\sum_t \gamma^2 < \infty \; and \sum_t \gamma = \infty$ then stochastic gradient decent is converging with almost 1 probability (Bottou, 2022).

Although stochastic gradient decent indeed is applicable to ANNs there have been even more sophisticated learning algorithms that provide increased speed of convergence and also are able to overcome problems that arise when the objective function is noise and the data sparse, both of which often are true in deep neural network problems. The one that is most commonly utilized for state-of-the-art models is the *Adam optimizer.* Adam stands for Adaptive Moment Estimation and uses properties of two other popular optimizers, RMSProp (Tieleman, 2012) and Adadelta (Duchi, 2011). The algorithm works as presented below:

Adam Algorithm

Require: α: Step size or learning rate
Require: $\beta_1, \beta_2 \in [0,1)$ **:** Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$
$u_0 \leftarrow 0$
While ϑ not converged **do:**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients with respect to the stochastic obj fun)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$(Update biased first moment estimate)
    $u_t \leftarrow \beta_2 \cdot u_{t-1} + (1 - \beta_2) \cdot g_t^2$(Update biased second moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$(Compute bias corrected first moment estimation)
    $\hat{u}_t \leftarrow u_t / (1 - \beta_2^t)$ (Compute bias corrected second moment estimation)
    $\theta_t \leftarrow \theta_{t-1} - a \cdot \widehat{m}_t / (\sqrt{\hat{u}_t} + \epsilon)$ (Update parameters)
End while
return

Except for the abilities to converge is sparse spaces, another desirable property is the fact that the updates are invariant of the scale of gradients. This is a very important property that limits the possibility of exploding gradients, a severe problem in ANN training. We can see that if the gradients are scaled by any constant *c* then due to the fact that the update factor is $\widehat{m}_t / (\sqrt{\hat{u}_t} + \epsilon) \leq \widehat{m}_t / \sqrt{\hat{u}_t}$ the contribution of the scaling factor is canceled out (Diederik P. Kingma, 2015).

Backpropagation:

Backpropagation is an efficient algorithm for adjusting the weights of an ANN, in order to gradually minimize the difference between the network output and the desired output for a given sample. The idea was formally introduced in 1986 by (David E. Rumelhart, 1986). This algorithm can calculate the gradient with respect to each parameter of weight of then network, so one of the optimization methods like the ones mentioned before, can be applied to update the weights.

For any given unit of the network the total input x_j is equal to the sum of outputs y_i of the previous layer units that are connected to it, multiplied by their corresponding weights.

$$x_j = \sum_i y_i w_{ji} \quad (1)$$

The output of each neuron is a non-linear function that is known beforehand and so is its derivative with respect to its input. The total error in performance can be measured in several ways according to the task but let us assume that in this case we are using the Mean Squared Error that is the loss function that we are going to use for our agent.

$$E = \frac{1}{2}\sum_c \sum_j \left(y_{j,c} - d_{j,c}\right)^2 \quad (2)$$

$d_{j,c}$ is the desired output for the given sample. The partial derivatives we need to calculate can be calculated with one forward and one backward pass of the network. The calculations are shown below:

- $\frac{\partial E}{\partial y_j} = y_j - d_j$
- $\frac{\partial E}{\partial x_j} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial x_j}$ $where$ $\frac{\partial y_j}{\partial x_j}$ $is\ the\ derivative\ of\ the\ activation\ function\ and\ its\ analytical\ formula\ is\ known\ beforehand.$
- $The\ contribution\ of\ the\ output\ of\ unit\ i\ to\ \partial E/\partial y_i\ from\ the\ effect\ of\ i\ on\ j\ is$ $\frac{\partial E}{\partial x_j}\frac{\partial x_j}{\partial y_i} = \frac{\partial E}{\partial x_j} W_{ji}$
- $Finally\ taking\ into\ account\ all\ connections\ from\ unit\ i$: $\frac{\partial E}{\partial y_i} = \sum_j \frac{\partial E}{\partial x_j} W_{ji}$

Using the partial derivatives for every y_i we can proceed with the optimization. Since during the backward pass the gradients of every layer depend only on the gradients of the previous layer, each gradient needs to be calculated only once and also the calculations for every layer can be executed in parallel. These properties make the algorithm very efficient and also allow for increased parallelism that is the norm in modern ANN training.

3.8 Deep Q Learning

3.8.1 Deep Q Learning (DQN)

As we mentioned in the chapter of Q Learning, this algorithm attempts to find the sequence of actions that maximize total discounted rewards by trying to estimate the Q for any given pair of state-action and then make greedy choices by opting for the action that has the highest Q value every time. It is apparent, that the better the estimation of the Q values, the closer the action sequence to the optimal will be. Attempting to solve a problem using the definition of Q Learning is inefficient, because according to the original algorithm every sequence is evaluated independently and no form of generalization can take place. To enable generalization, a parametrized Q function is used $Q(s, a; \theta) \approx Q^*(s, a)$, where θ is a set of trainable parameters. In deep Q Learning the approximator is a neural network that attempts to estimate optimal Q values for all action sequences using the same function.

Deep Q Learning models are trained using tuples of $\langle s, a, r, s' \rangle$ (state, action, reward, next state). The difference in implementation compared to the original algorithm and other Reinforcement Learning implementations is that the tuples are not used for training in the same order as the agent observes them. In DQL a structure known as *replay memory* is used. The *replay memory* is a fixed size buffer that stores the N latest tuples observed by the agent. At every step first an observation of the state is made. Then an action is chosen based on the observed state. Finally, the state that the action leads to is observed and the reward is calculated. The new tuple is stored in the memory and then *m* random tuples are chosen from the memory to train the agent. The process is then repeated.

Using a *replay memory* instead of just using the samples as they are received for training provides three important advantages. The first is that this method of training leads to more efficient sample utilization. Every sample is almost certainly used more than once for weight updates. Combined with the fact that neural networks demand small learning rates to converge it is very likely that using a training tuple only once to update the network weight is not sufficient to gain all the information possible from this training sample. This method increased the chances that a tuple will be used more than once so that more information can be extracted from the sample. The second advantage is that this training technique breaks the strong correlations between successive samples. This is important because randomized samples reduce the variance of the updates, which leads to faster convergence. Additionally, learning on-policy is prone to get stuck in local minima. The reason is that at every step the network makes a choice based on its parameters and then trains based on the choice it made. If the choice is locally optimal, the network is going to repeat the choice and ignore other options that potentially lead to greater overall reward (Martin, 2013).

3.8.2 Double Deep Q Learning (DDQN)

The original Q Learning algorithm is known to overestimate the Q values of the model. This phenomenon is not necessarily harmful for the performance of the algorithm and the policy it will result to. A common exploration technique called *optimism in the face of uncertainty,* is bases on this idea. Every unexplored Q value is assigned a high numeric value, so that the algorithm is incentivized to sufficiently explore the state space before making greedy locally optimal actions. Moreover, if the overestimation of the values is uniform then the dynamics of the action preferences are preserved, leading to the optimal policy. In their paper however, Hado et al. (Hado van Hasselt, 2016) claim that in several applications of DQN the overestimation is not uniform and it indeed harms performance.

In their paper, they propose an adaptation to the original DQN model, where a second model is added to the agent. Their implementation is named Double Deep Q Learning. The idea is that one of the main reasons of overestimation is the fact that the same network is used to select the optimal action and then evaluate its value. This creates an unwanted feedback loop, where a slightly increased Q value is more likely to be chosen again in the future without being the optimal solution in reality, increasing long term overestimation. This problem is more likely to occur when the approximator is a neural network because at the early stages of training Q values are arbitrary due to randomized weight initialization. Even slight initial overestimations, can lead to noticeable long-term deviation from the optimal policy (Hado van Hasselt, 2016).

To alleviate the overestimation, the authors propose to decouple the action selection from the action evaluation. The two networks of the agent are called online and target network accordingly. The online network is updated normally using the training samples. The target network is a lagging copy of the online network, meaning that every N steps, the parameters of the online network are copied to the target network. The greedy policy or action selection is evaluated using the online network. Then the target network is used to estimate the Q value of the state action pair and the weight update of the online network is performed using this estimation. The Q estimation rule for DDQN is shown below:

$$L^{DDQN} = R_{t+1} + \gamma Q\left(S_{t+1}, arg\max_{a} Q(S_t, a; \theta_t); {\theta'}_t\right)$$
$$where\ \theta_t\ are\ the\ parameters\ of\ the\ online\ network\ and$$
$${\theta'}_t\ the\ parameters\ of\ the\ target\ network$$

The frequency of synchronization between target and online network is not specific and in reality, is a tunable hyperparameter. Small values degrade the model to a simple DQN agent and introduce overestimation. Large values slow down learning because the updated Q values are not known to the evaluator for longer timeframes. According to state-of-the-art implementations, it seems that a reasonable decision is to synchronize the weights after every episode. DDQN indeed solves the overestimation problem and leads to higher performance in most tasks and for that reason DDQN has become the default implementation for solving problems using the Q Learning algorithm.

3.8.3 Return Based Scaling

Scaling issues in Reinforcement Learning models is a tedious task but also a necessary one because when errors scales vary across different stages of training, it can hinder or obstruct the convergence of the model. Especially in model-free algorithms, where the agent has to accurately estimate the value function that describes the underlying dynamics of the problem, scaling issues are even more severe.

There are various factors that can affect the error scales during training and each one of them can be detrimental for the convergence of the model. The most common is the reward function. Every Q value is the discounted sum of the current and future rewards the agent expects to accumulate. The greater the variance of the rewards is, the greater the disparity of the Q values can be during training. This can lead to error scales that vary in many orders of magnitude due to the cumulative nature of the Q values. Even if the reward function does not display high variance, it is possible that during an update the estimated and observed Q value vary greatly, leading to a high numeric value of error. Using such a value for an update can distort the convergence of the weights because neural networks are smooth function approximators. This phenomenon is more likely to happen during the early stages of training, where the agent explores the state space and it possible that it had not acted optimally around a specific part of the state space until that point. It is also possible to happen when at some point in the training the agent discovers new possibilities for higher rewards that became available after the policy started to change due to the training. Finally, the discount factor also greatly affects the arithmetic values of Q values. Trying different discount factors for the same problem may demand different scaling of the rewards, adding to the struggles, reinforcement learning practitioners face when they try to parametrize their models.

To overcome the aforementioned problems, reinforcement learning practitioners resorted to empirical solutions. These solutions may be efficient depending on the dynamics of a specific problem and the distribution of the reward function but they are not widely applicable and also not supported by academic literature. Some examples are reward clipping and reward or return normalization. These methods can effectively address the scaling problem but they are problematic because they hide certain aspects of the dynamics of the problem, that are expressed by the variance in the values of the reward function. As a result, these methods can make it impossible for an agent to reach the optimal policy due to the distortion introduced. Tom et al. (Schaul, 2021) propose an alternative approach, to the scaling problem that preserves the dynamics of the problem while alleviating the numerical fluctuations between updates of the Q values and consequently the weights. They propose to apply the scaling directly in the temporal difference, meaning the input of the loss function of the neural network. The scaling factor is adaptive and is updated during each training step. The scaling problem of the updates is more noticeable during the early stages of the training because as the model approaches convergence, errors should approach 0 asymptotically. The derivation of the scaling factor is shown below:

$$\delta = R_t + \gamma V'_{t+1} - V_t$$
$$where\ R_t\ is\ the\ reward\ observed\ at\ step\ t$$
$$and\ V_t\ the\ estimatedQ\ value\ of\ the\ model\ at\ step\ t$$

To find an approximation of the scaling factor we must estimate $V[\delta]$. At the early stages of training, we can assume that the rewards are independent from the Q values since the agent performs random actions almost always. Using this assumption, we can express $V[\delta]$ as:

$$V[\delta] = V[R] + V[\gamma(V' - V)] + V[(1-\gamma)V]$$
$$= V[R] + \gamma^2 V[V' - V] + V[\gamma]E[(V' - V)^2] + (1-\gamma^2)V[V] + V[\gamma]E[V^2]$$

At every step the Q values are updated with the rule $Q = R + \gamma Q'$ and Q values estimate the overall gain of the agent. It is reasonable to substitute V for G and for the one step difference $\mathrm{G'} - \mathrm{G} = \mathrm{R} + (1-\gamma)\mathrm{G}$. Also $G = \sum_t \gamma^t R_t$ so $E[G] \approx (1-\gamma)E[R]$. Using these equations and substituting in $V[\delta]$ we can write:

$$V[\delta] = V[R] + (1-\gamma)^2 V[G] + V[\gamma]E[G^2]$$

The term $(1-\gamma)^2 V[G]$ can be neglected because it is dominated by the other two. Also, when γ is constant or when the training is continuous and not divided in episodes, the term $V[\gamma]E[G^2]$ is also neglected leading to $V[\delta] = V[R]$. The scaling factor is σ where $\sigma^2 = V[\delta]$. The difference between this method is that $V[\delta]$ is calculated online at every step of the training. The authors also account for some edge cases that can occur during training. The most notable is the case where batch training is used and the variance of the rewards of the batch is greater than the overall variance. To avoid detrimental updates the scaling factor must be altered to $\sigma^2 = \max(V[\delta], V[\delta_{batch}])$

3.9 Offline Reinforcement Learning

3.9.1 Introduction

Offline Reinforcement Learning is a lucrative research field that has been drawing increasing attention over the latest years. The reason is that in theory, offline reinforcement learning can leverage the immense datasets that exist and effectively train agents based on these static datasets without further interaction. Currently, reinforcement learning is an active learning process, where the agent performs an action observes the results and then reiterates. This approach has limited applicability because first of all the quantities of data that can be generated are limited compared to offline training. State-of-the-art models of machine learning owe a major part of their success to the immense amounts of the training datasets they are presented. Except for the dataset limitations, interactions with the environment can be costly and/or catastrophic in several applications such as robotics or medical applications. For the reasons stated above, effectively applying offline reinforcement learning is a key challenge for the adoption of reinforcement learning in real-world environments.

The problem of most value based off-policy offline Reinforcement Learning methods is that they display poor performance in reality. The main reasons of failure are overfitting and out-of-distribution actions (OOD). These problems usually manifest themselves as erroneous overestimations of the value function at certain states. More specifically, the problem lies in the fact that the Bellman optimization algorithm tries to sample actions from the learned policy that is created as the model is trained but the Q values can only be trained on values sampled from the policy that generated the offline dataset. Since the algorithm is created to use the learned policy, it often leads to OOD actions. When these actions have erroneously high values, they lead to overestimations. Typical offline Reinforcement Learning applications mitigate this effect by restraining the algorithm from opting for unobserved states. These attempts however result in over-restrictive policies that limit the performance of the agent during testing.

3.9.2 Conservative Q-Learning

The aim of Conservative Q-Learning is to estimate the value function $V^{\pi}(s)$ of a target policy $\pi(s)$ given a static dataset D that is generated by a behavior policy $\pi_{\beta}(a|s)$. Ideally, the target policy is identical to the optimal policy. To learn in a conservative manner, an additional term is added to the minimization equation alongside the standard Bellman objective that is used in online Reinforcement Learning. The intuition behind this additional term is that since the values of the dataset D are generating by the behavior policy $\pi_{\beta}(a|s)$, actions that are more likely according to this policy are more likely to be overestimated and so this additional term acts as a penalty for these actions. The objective function of conservative Q- Learning is provided by the equation:

$$Q^{k+1} = \min_{Q} \; a \cdot \left(E_{s\sim D, a\sim\mu(a|S)}[Q(s,a)] - E_{s\sim D, a\sim\pi_{\beta}(a|S)}[Q(s,a)]\right) + \frac{1}{2}L \qquad (1)$$

where L is the standard Bellman objective function and $\mu(\alpha|s)$ is the desired distribution action-states after training. The authors that propose the solution prove that for $\mu = \pi$ the resulting estimation of the value function and the Q values satisfies the restriction

$$\hat{V}^{\pi}(s) < V^{\pi}(s) \forall s \in D$$

Meaning that every Q value that can be estimated using the static dataset, is bounded by the actual value of the Value function, so overestimation is eliminated. The constant α is a hyperparameter of the optimization problem. In reality this constant needs to be sufficiently big for a dataset of fixed size. In other words, the larger the size of the dataset, the smaller α can be. Asymptotically, for a large enough dataset α can take very small numeric values and the objective function is dominated by the Bellman objective term.

Observing (1) it is apparent that the minimization involves *a priori* knowledge of the distribution $\mu(\alpha|s)$. However, μ is a part of the training process and after a sufficient number of training steps we want $\mu = \pi$. Since μ is a part of the optimization problem we can include a maximization over μ in the conservative-learning term so that at every iteration the objective function is:

$$Q^{k+1} = \min_{Q} \max_{\mu} a \cdot \left(E_{s\sim D, a\sim\mu(\alpha|s)}[Q(s,a)] - E_{s\sim D, a\sim\pi_{\beta}(a|s)}[Q(s,a)]\right) + \frac{1}{2}L + R(\mu)$$

where $R(\mu)$ is a regularizer term. A reasonable choice of $R(\mu)$ is the Kullback-Liebler divergence (KL). KL-divergence $D_{KL}(P||Q)$ is a type of statistical distance that expresses the additional surprise or uncertainty introduced because of our choice to use as a model a distribution Q when the actual distribution is P. In our case P=μ and Q is a prior distribution of action-states. When the distribution of actions is almost uniform at every state, then the maximization over μ results in a soft-max of the Q-values at any given state and the objective function is transformed to:

$$Q^{k+1} = \min_{Q} a \cdot E_{s\sim D}\left(log \sum_{a} \exp\left(Q(s,a)\right) - E_{s\sim D, a\sim\pi_{\beta}(a|s)}[Q(s,a)]\right) + \frac{1}{2}L \quad (2)$$

Transforming equation (2) to a Loss function that can be used to calculate gradients for a neural network is straightforward:

$$L = a \cdot E_{s\sim D}\left(log \sum_{a} \exp\left(Q(s,a)\right) - E_{s\sim D, a\sim\pi_{\beta}(a|s)}[Q(s,a)]\right) + \frac{1}{2}L$$

At every step of the training we randomly pick a subset of the dataset and calculate *L* and then update the parameters of the network using the chosen optimizer (Aviral Kumar, 2020).

4. Experimental Results

4.1 Setup

In this section we briefly describe the way we coordinated the different components used to perform our experiments. We used a K8ssandra deployment that runs inside a distributed Kubernetes cluster. The clients that generated the traffic load ran an instance of the YCSB service and a remote script monitored the number and nature of the generated requests. For the collection of metrics, we used a Prometheus instance that was deployed inside the Kubernetes cluster as well. The VMs were hosted in the Okeanos cloud environment.

The Kubernetes cluster consists of 10 VMs, one of them acted as a Master Node and the rest as Worker Nodes. Each of the worker nodes had 4GB of RAM, 30GB of storage space and 2 virtual CPU cores. The master node had 8GB of RAM, 30GB of storage space and 4 virtual CPU cores. From the 30GB of available storage of every node, 15GB were allocated as a virtual disk and provided to the Kubernetes cluster as a Persistent Volume. Every worker node had an instance of the Kubernetes local volume static provisioner running on it. Its role is to manage the PersistentVolume lifecycle for pre-allocated disks by detecting and creating PVs for each local disk on the host, and cleaning up the disks when released. Every K8ssandra node needs at least 2GB of RAM to operate flawlessly. For this reason, only one K8ssandra node could run at a time per worker node. The resource limits that performed best in our setup were to use 2GB of RAM and 1 CPU core per K8ssandra node. We also allowed the nodes to exceed RAM usage to a margin of 0.5GB. This allowed the K8ssandra cluster to perform scaling operations even under head traffic and high percentage of resource utilization. Finally, we deployed 3 instances of Stargate nodes to coordinate incoming requests.

The role of the client generating the queries against our database was carried out by the YCSB framework. YCSB is a benchmark tool written in Java that can generate traffic for several database systems. The workloads can be configured in terms of target loads (in requests per second), the number of operations to be executed, time limits of total execution time and the percentage of reads and writes among others. To generate the traffic needed we created 4 additional VMs with the YCSB installed in them. The generated traffic was monitored by a remote script that sent commands to the VMs to execute partitions of the total workload and ensured that the total workload was evenly distributed among client machines.

In our setup all of the monitoring was performed on the server-side. The metrics were periodically scraped by the Prometheus instance and stored in its database. Then the script that executed the monitoring agent, performed PromQL queries against the database over HTTP to collect the metrics. After that the metrics were normalized and formatted as a *numpy* array and finally provided as input to the decision-making module.

4.2 Results

For the training of our models we used 17 parameters to describe the state of the cluster:

- The size of the cluster
- The average 98th percentile latency measured by the stargate nodes
- The average 99th percentile latency measured by the stargate nodes
- The average 999th percentile latency measured by the stargate nodes
- The throughput or requests per second measured
- The throughput measured at the previous decision step
- The total free memory of the cluster as a percentage of the total available memory
- The total cached memory of the cluster as a percentage of the total available memory
- The average CPU utilization of the cluster
- The minimum CPU utilization of the cluster
- The maximum CPU utilization of the cluster
- The average CPU that is idle in the cluster
- The average CPU time spent waiting for IO
- The average IOPS in the cluster
- The average disk read throughput of the nodes in the cluster
- The average disk write throughput of the nodes in the cluster
- The percentage of reads in the incoming load

All the metrics are scraped by Prometheus every 10 seconds and the measurements are averaged over a 5 minutes interval.

The reward function used to evaluate every state is:

$$R = 0.01 * throughput - (vms - B) \; where \; B \; is \; the \; minimun \; cluster \; size$$

Regarding the network architecture of the online agent, we used a fully connected network with 2 hidden layers. The first hidden consisted of 48 nodes and the second layer consisted of 24 nodes. The replay memory buffer was set to store the last 300 experiences and the weights were updated with a learning rate of *α=0.001*. The discount factor was set to *γ=0.99.* To avoid weight updates that could lead to the divergence of the system, we used the return-based scaling technique described in the previous chapter to normalize the loss at every update step according to the running variance of the rewards of the past experiences. The training starts with a replay memory of 300 random experiences. The agent then performs 500 annealing steps, with *epsilon* decaying from 1 to 0.1 linearly over the course of the 500 steps. We preserve a small *epsilon* value over the rest of the training to preserve the potential of the agent to explore higher reward states at later stages of the training. The agent at every step observes the current state of the cluster and chooses between 3 actions. Increment the cluster size by 1, decrement the cluster size by 1 or do nothing. In the cases where the size of the cluster changed, the agent periodically observed the state of the

cluster as described by the variable *status.cassandraOperatorProgress.* When the value of this variable was set from “Updating” to “Ready” the agent waited for 5 minutes and then collected the metrics from Prometheus to perform the next action. When the cluster remained unchanged, the agent waited for 2 minutes and then collected the metrics to perform the next action. The batch size for the training after every decision was set to 32 randomly sampled memories from the replay buffer.

For the network architecture of the offline agent, we again used a fully connected network with 2 hidden layers. Given the fact that we had access to an offline dataset and training can be performed in a matter of minutes or hours, we had greater freedom to tune the hyperparameters of the model. For every checkpoint that we compare our agents, we use a different number of nodes for the hidden layers to optimize the performance of the model. Also, the hyperparameter *a* of the CQL loss function ranges from 5 for the smallest dataset to 1 for the largest dataset.

Finally, we utilized an additional optimization that is described in literature as *initial value offset* (Schaul, 2021). Although the effectiveness of this method is not proven theoretically, empirical results show that in certain cases it speeds up convergence dramatically. The intuition behind this method is that in the output layer of the algorithm in Deep Q Learning tries to estimate the numeric value of the Q function for a pair of state-action. When the nature of the reward function is such that it is significantly offset from 0, then initializing the biases of the output layer to 0 as per usual, can be problematic. The reason is that the agent will spend a great amount of training time to increment the biases to reach the order of magnitude of the Q function, given that the Q values are discounted sums of the observed reward values, before it can start to effectively learn the dynamics of the problem. To overcome this delay in learning we can initialize biases with an estimation of the mean value of the overall gain of the agent $E[G]$ based on some initial statistics. Our experiments have shown that this bias initialization indeed helps the training to take off sooner than the zero initialization approach.

We now present the comparative performance of the online and offline agent at specific training checkpoints.

Minimal Dataset(300 random experiences):

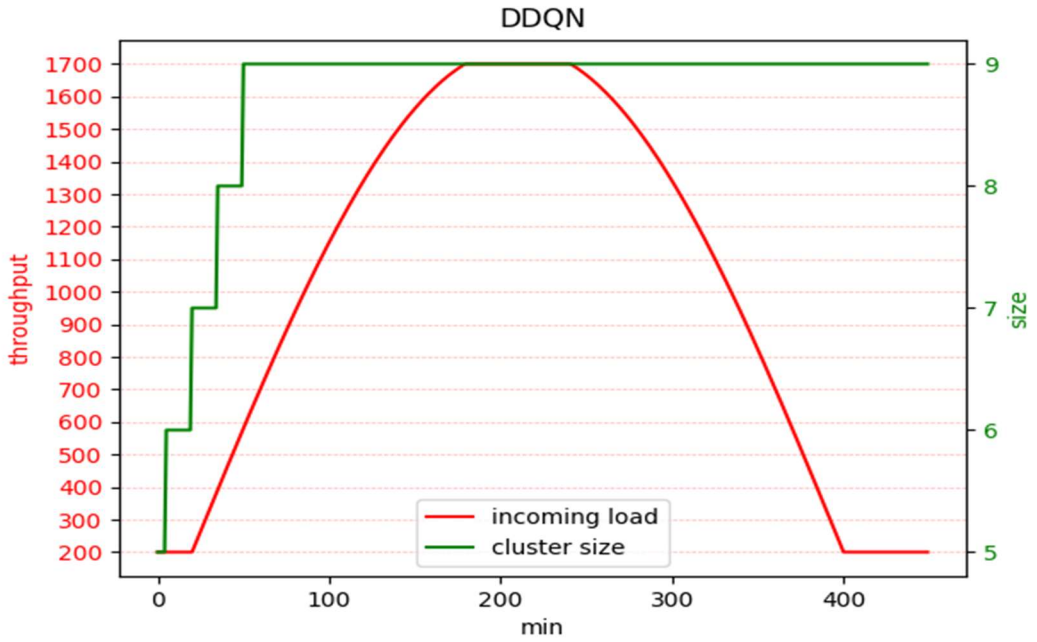

Figure 25: CQL behavior under sinusoidal load (minimal dataset)

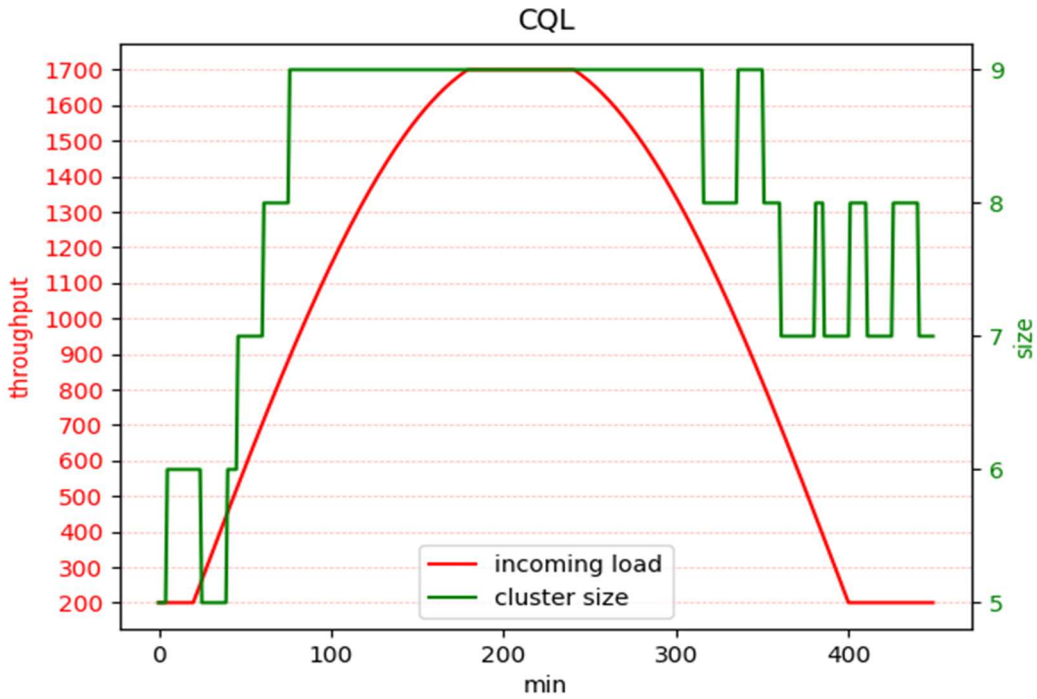

Figure 26: DDQN agent behavior under sinusoidal load (minimal dataset)

Observing the behavior of the two models it is apparent that the CQL agent can already extract some knowledge from the minimal dataset of 300 observations about the dynamics

of the problem. On the contrary, the DDQN agent has only learned that higher cluster sizes can potentially lead to higher rewards.

Small Dataset (800 experiences):

Next, we compare the performance of the two models after the DDQN agent has completed the annealing steps, meaning the part of the training that the agent performs mainly exploratory actions. Again, we can observe that the offline agent is able to scale the cluster more drastically. The DDQN agent is still biased toward higher cluster size states.

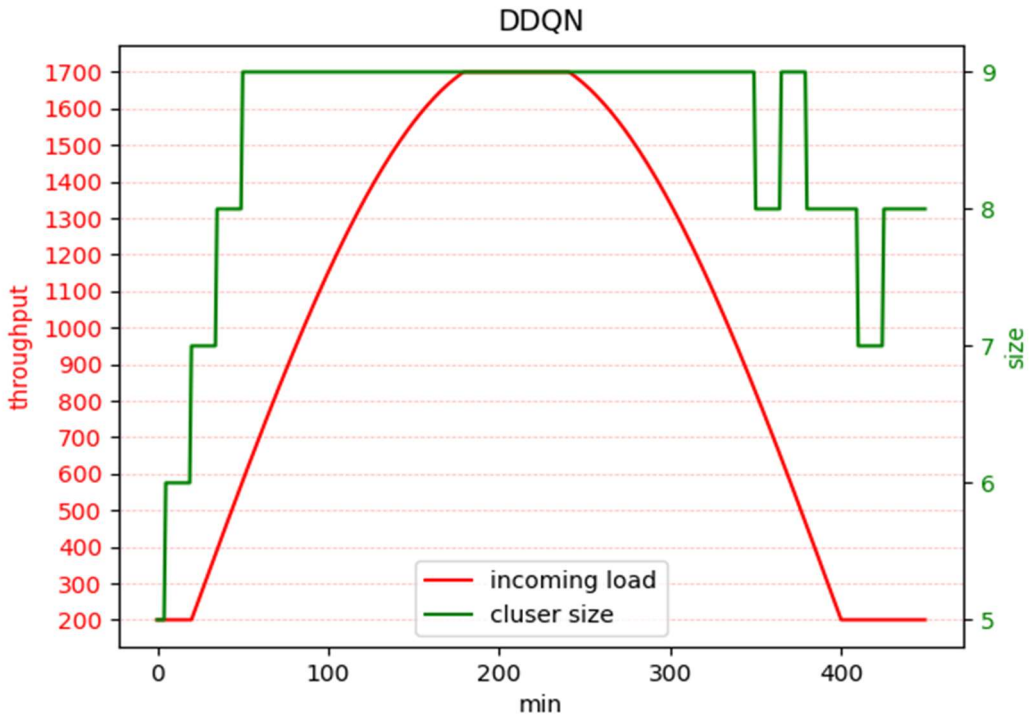

Figure 27: DDQN agent performance under a sinusoidal load (small dataset)

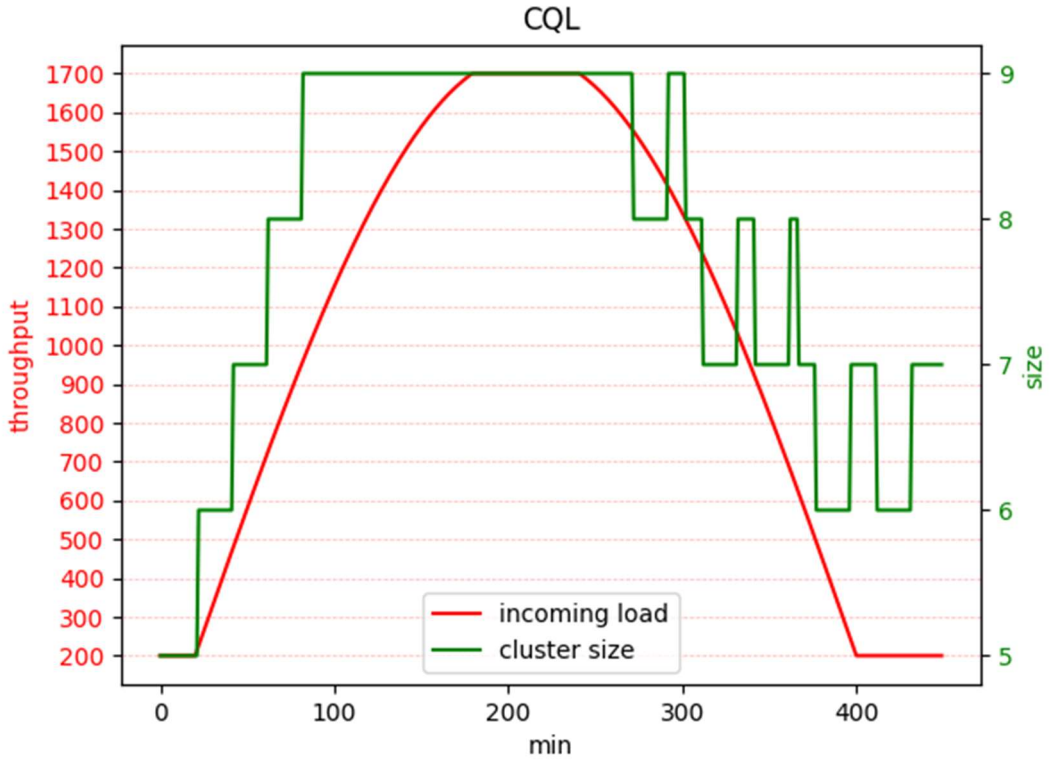

Figure 28: CQL agent behavior under a sinusoidal load (small dataset)

Medium Dataset (1800 experiences):

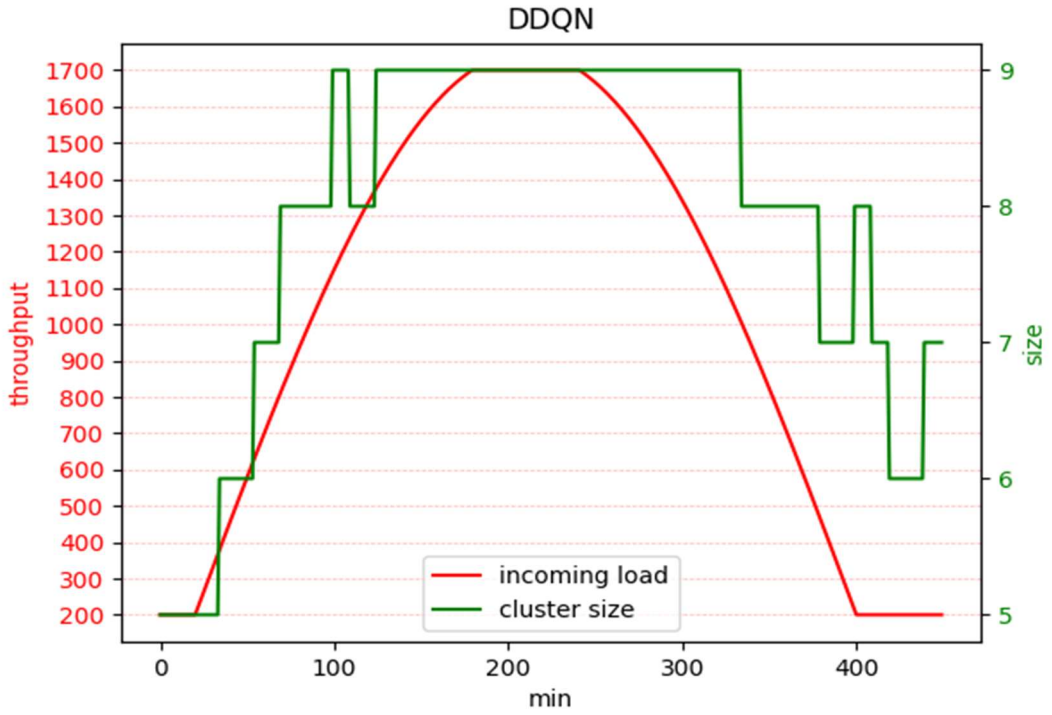

Figure 29: DDQN agent behavior under sinusoidal load (medium dataset)

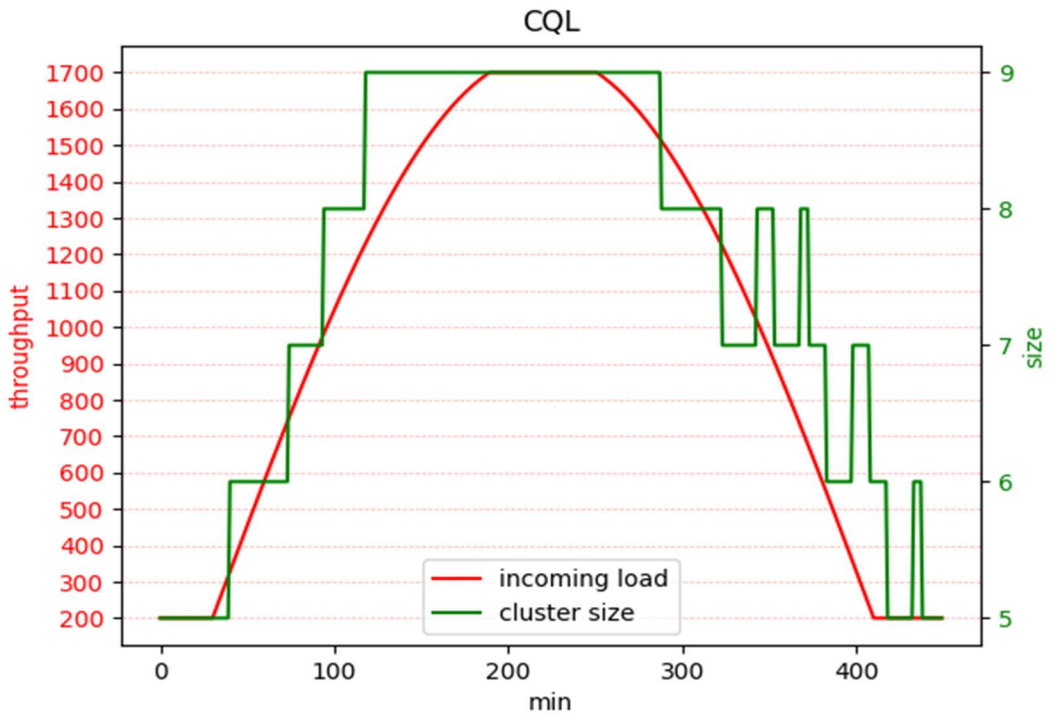

Figure 30: CQL agent behavior under sinusoidal load (medium dataset)

Final Dataset (3300 experiences):

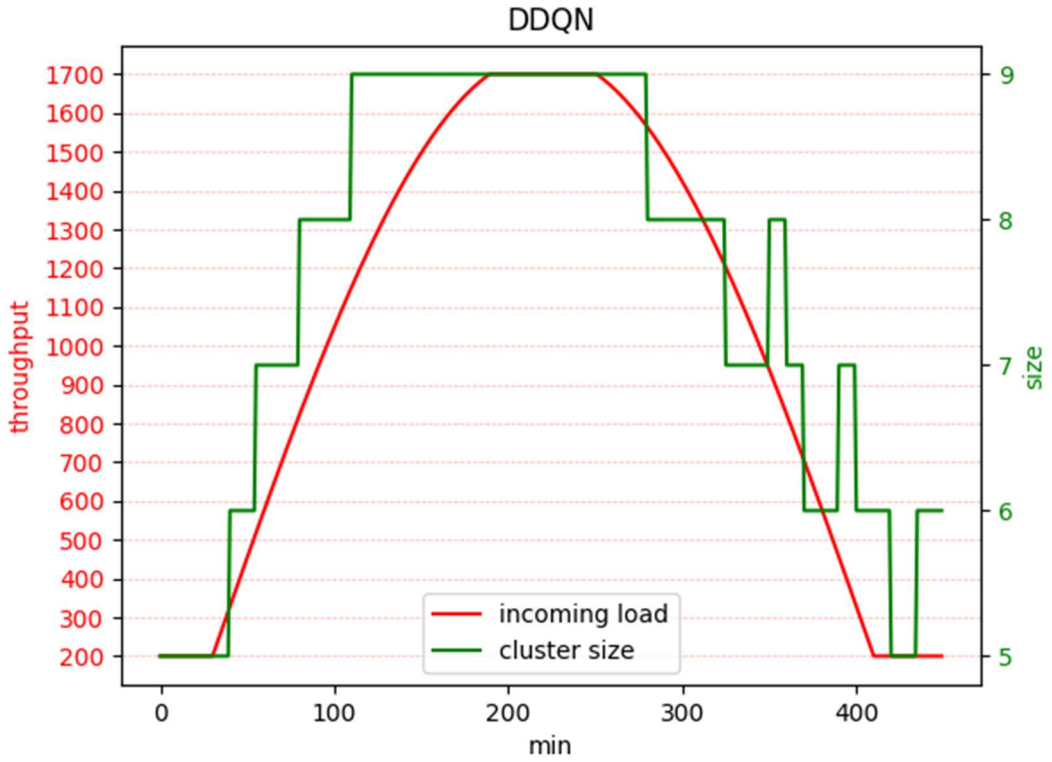

Figure 31: DDQN agent behavior under sinusoidal load (final dataset)

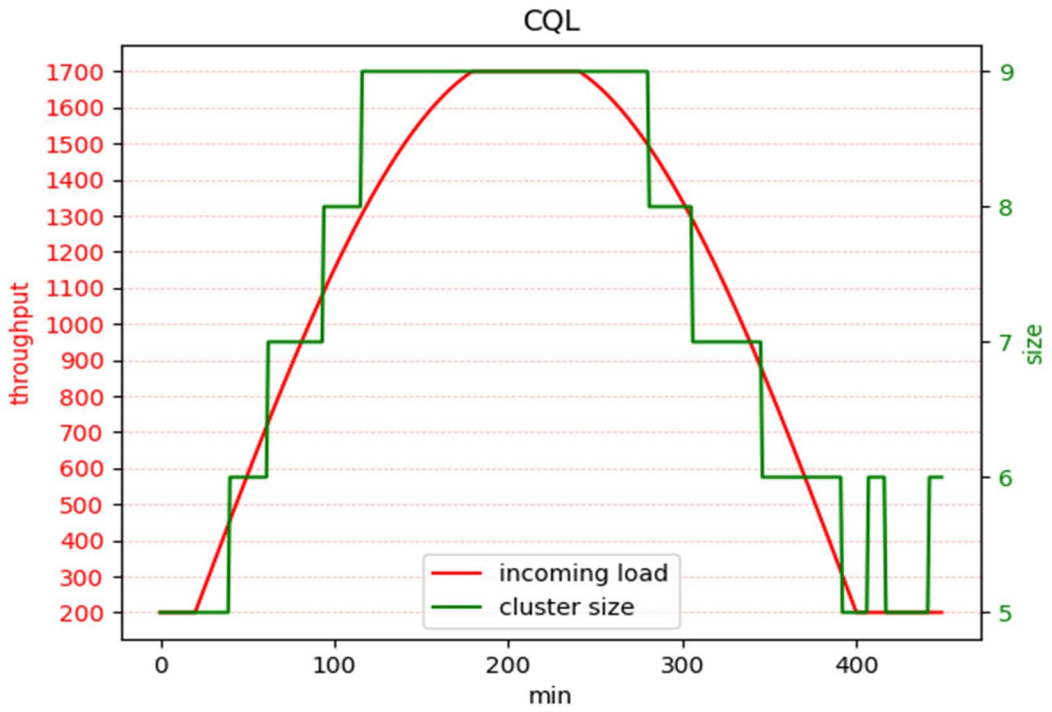

Figure 32: CQL agent behavior under sinusoidal load (final dataset)

Increasing the size of the dataset from that point onwards has shown no significant improvement for the offline agent and as a result we terminate the training at this point. The online agent continues to improve as it interacts with the environment but the progress slows down greatly.

Dataset	**DDQN**	**CQL**	**Improvement**
Minimal (300 exp)	581.43	642.73	10.5%
Small (800 exp)	601.74	670.05	11.3%
Medium (1800 exp)	659.71	698.36	6.1%
Final (3300 exp)	690.42	714.62	3.5%

As a final benchmark of our model, we monitor our best offline agent performance over some unseen workloads to observe its generalization capabilities.

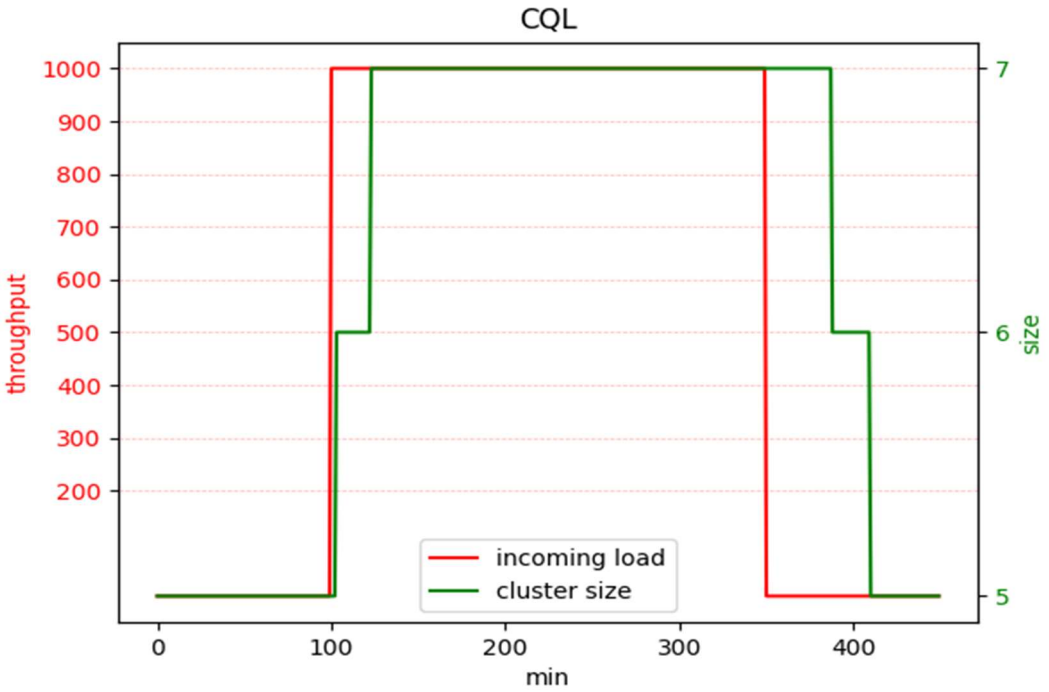

Figure 33 CQL agent behavior under a constant load

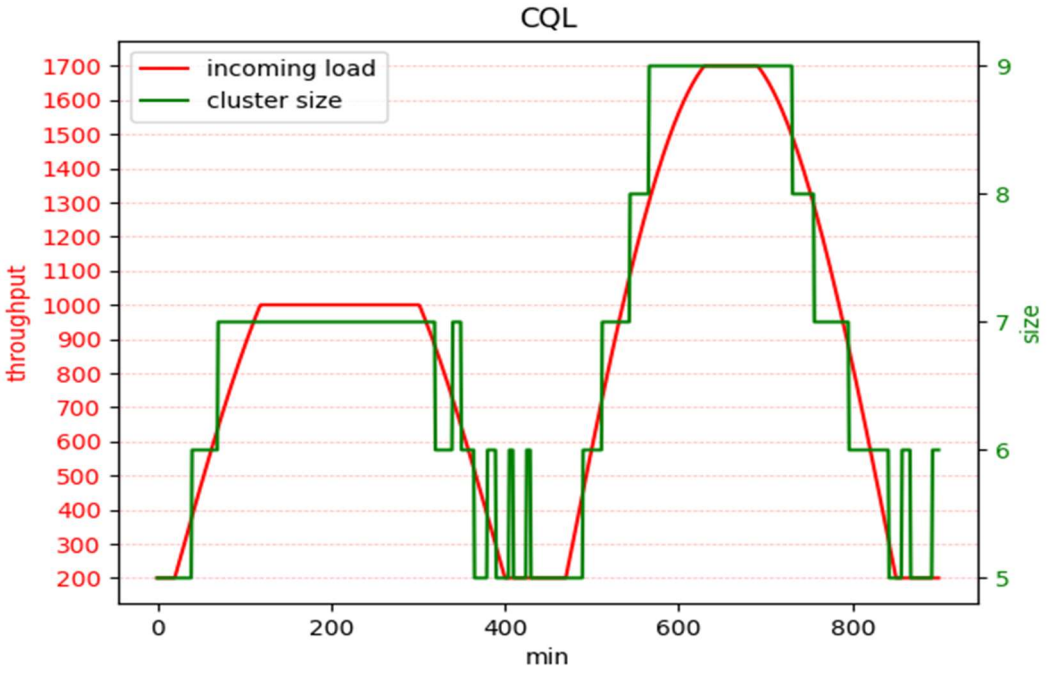

Figure 34 CQL agent behavior under sinusoidal loads of different aptitudes

4.3 Conclusion

During this thesis, we had the opportunity to experiment with contemporary deep reinforcement learning techniques to enhance the capabilities of an already powerful tool for containerized applications. Our proposed model provides a monitoring agent that can effectively auto-scale complex applications in order to maximize resource utilization without compromising performance. Moreover, the agent is capable of discovering they dynamics of the monitored system even from very small datasets.

Using a containerized version of the application enhanced the training process because it allowed to accumulate an increased number of diverse experiences in the same amount of time compared to previous attempts that relied on VMs setups. The process can be accelerated further if we opt to compromise resource utilization, so more resources can be dedicated for the scaling tasks rather than executing the workloads. Nevertheless, our deployed application showed high level of resilience and was able to perform scaling operations even under severe resource pressure. This is due to the efficient scheduling algorithms of the Kubernetes, that reorganize the deployed resources to ensure minimum interruptions to the deployed workloads.

The experiments with online Deep Reinforcement Learning algorithms highlighted the practical challenges that occur when applying these models to realistic scenarios. The most important is the fact that these models need constant and extensive interaction with the environment they monitor. This means that in order to perform successful training, the system must be configured meticulously, to avoid unexpected behaviors due to the agent's actions that may lead the system towards destructive states. The second challenge is the limitation in accumulated experiences. Typical Deep Reinforcement Learning application require millions of experiences and this may be unfeasible is realistic applications. Finally, due to the limitation in experiences, performing hyperparameter tuning is a very time-consuming task.

The offline model we propose tackles all these challenges effectively. First of all, since the problem is essentially transformed to an unsupervised learning problem, we are able to perform hyperparameter tuning to derive the optimal model for the problem. Moreover, since the model is trained without any interaction with its environment, it is much less probable that after deployment it will lead the system to destructive states, if the problem is defined correctly. The offline agent is able to extract significantly more information from the provided dataset, compared to the online agent. As a result, it is able to converge to a solution much faster than the online equivalent. Finally, we observe that at every checkpoint of the training, the offline model is able to mitigate the bias of the online agent towards higher cluster size states, a fact that supports the claim that our agent is able to systematically derive a better decision-making policy than the one provided by the dataset.

Although limited to certain cases, our offline agent showed some capabilities of generalization over unseen workloads. Extensive generalization with Deep Reinforcement Learning still remains an unsolved issue. Nevertheless, these results are encouraging for further experimentation, especially with offline Reinforcement Learning techniques that tackle the generalization problem directly.

Citations

Avinash Lakshman, P. M. (2014). Retrieved from http://planetcassandra. org/what-is-apache-cassandra.

Aviral Kumar, A. Z. (2020). Conservative q-learning for offline reinforcement learning. *Advances in Neural Information Processing Systems, 33*, pp. 1179-1191.

AWS, Amazon Autoscaling. (n.d.). Retrieved from " https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-scale-based-on-demand.html

Bellman, R. (1957). *Dynamic Programming.* Princeton, NJ.: Princeton University Press.

Bottou, L. (2022, 7 5). *Large-Scale Machine Learning.* Retrieved from https://leon.bottou.org/publications/pdf/compstat-2010.pdf

citrix.com. (2022, 6 26). Retrieved from citrix.com/solutions/app-delivery-and-security/what-is-containerization.html

Constantinos Bitsakos, I. K. (2018). DERP: A Deep Reinforcement Learning Cloud System for Elastic Resource Provisioning. *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom).*

D. Tsoumakos, I. K. (2013). Automated, Elastic Resource Provisioning for NoSQL Clusters Using TIRAMOLA. *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium* (pp. 34–41). IEEE.

Dahl, G. Y. (2012). Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *Audio, Speech, and Language Processing, IEEE (20)*, pp. 30–42.

David E. Rumelhart, G. E. (1986). Learning Representations by back-propagating errors. *Nature 323*, pp. 533-536.

Diederik P. Kingma, J. B. (2015). Adam: A Method for Stochastic Optimization. *International Conference for Learning Representations*.

Dirk, M. (2014). Docker:lightweight linux containers for consistent development and deployment. *Linux j* .

Duchi, J. H. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, pp. 12:2121–2159.

Featherston, D. (2010). Cassandra: Principles and Application.

Glorot, X. B. (2010). *Understanding the difficulty of training deep feedforward neural networks.* Retrieved from http://proceedings.mlr.press/v9/glorot10a.html

Glorot, X. B. (2011). Deep sparse rectifier networks. *AISTATS, volume 15*, pp. 315–323.

Hado van Hasselt, A. G. (2016). Deep reinforcement learning with double Q-Learning. *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI'16)*, (pp. 2094–2100).

https://etcd.io/. (2022, 6 1). Retrieved from https://etcd.io/docs/v3.4/learning/data_model/

https://kubernetes.io/. (2022, 6 1). Retrieved from https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

https://kubernetes.io/. (2022, 6 2). Retrieved from https://kubernetes.io/docs/concepts/scheduling-eviction/scheduler-perf-tuning/

Hughes, G. (1968, January). On the mean accuracy of statistical pattern recognizers. *IEEE Transactions on Information Theory*, pp. 55–63.

I. Giannakopoulos, N. P. (2014, 10). Celar: Automated application elasticity platform. *2014 IEEE International Conference on Big Data*, pp. 23–25.

K. He, X. Z. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *IEEE International Conference on Computer Vision (ICCV)*, pp. 1026-1034.

K. Lolos, I. K. (2017). Elasitc management of cloud applications using adaptive reinforcement learning. *2017 IEEE International Conference* , (pp. 203–212).

K8ssandra. (2022, 7 4). Retrieved from https://k8ssandra.io/blog/articles/why-k8ssandra/

K8ssandra. (2022, 7 5). Retrieved from https://docs.k8ssandra.io/components/cass-operator/

Krizhevsky, A. S. (2012). Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems (NIPS 2012)*, p. 4.

L. P. Kaelbling, M. L. (1996). Reinforcement learning: A survey. *Journal of Articial Intelligence Research 4*, pp. 237-285.

Lucia Schuler, S. J. (2021). AI-based Resource Allocation: Reinforcement Learning for Adaptive Auto-scaling in Serverless Environments. *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)* (pp. 804-811). IEEE.

Maas, A. L. (2013). Rectifier nonlinearities improve neural network. *In International Conference on Machine Learning (ICML).*

Martin, M. V. (2013). Playing Atari with Deep Reinforcement Learning. *NIPS Deep Learning Workshop*.

Michell, T. M. (1997). *Machine Learning p2.* McGraw-Hill.

Microsoft's Azure. (n.d.). Retrieved from https://docs.microsoft.com/en-us/azure/monitoring-and-diagnostics/insights-autoscale-common-metrics

Nair, V. a. (2010). Rectified linear units improve restricted Boltzmann machines. *International Conference on Machine Learning (ICML).*

Prometheus. (2022, 7 6). Retrieved from https://prometheus.io/docs/concepts/metric_types/

R. Taft, N. E.-S. (2018). P-Store: An Elastic Database System with Predictive Provisioning. *Proceedings of the 2018 International*, (pp. 205–219). New York.

Rolnick, B. H. (2018). How to Start Training: The Effect of Initialization and Architecture. *32nd Conference on Neural Information Processing Systems* .

Schaul, T. O. (2021). *Return-based Scaling: Yet Another Normalisation Trick for Deep RL.* Retrieved from ArXiv: https://arxiv.org/abs/2105.05347

Scheepers, M. J. (2014). Virtualization and Containerization of Application Infrastucture: A Comparison. *21st twente student conference on IT*, (p. Vol 21).

Simon, S. (2000). Brewer's CAP Theorem. *Symposium on Principles of Distributed Computing*, (p. 2).

Tieleman, T. a. (2012). *Lecture 6.5 - RMSProp.* Retrieved from http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

Turnbull, J. (2019). *The Docker Book: Containerization Is the New Virtualization.*

Vapnik, V. N. (1971). On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities. *Theory of Probability and its Applications*, pp. 264-280.