Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων
Και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης Και Υπολογιστων

Επεκτάσεις για Χρονοδρομολόγηση και Αυτόματη Κλιμάκωση σε Συστοιχίες Kubernetes με Τοπική Αποθήκευση Δεδομένων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γρηγόριος Π. Θανάσουλας

Αθήνα, Ιούλιος 2022

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

Επεκτάσεις για Χρονοδρομολόγηση και Αυτόματη Κλιμάκωση σε Συστοιχίες Kubernetes με Τοπική Αποθήκευση Δεδομένων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γρηγόριος Π. Θανάσουλας

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

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

. . . . .	. . . . .	. . . . .
Νεκτάριος Κοζύρης	Γεώργιος Γκούμας	Διονύσιος Πνευματικάτος
Καθηγητής ΕΜΠ	Αν. Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ

Αθήνα, Ιούλιος 2022

National Techincal University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Extensions for Scheduling and Autoscaling on Kubernetes Clusters with Local Storage Considerations

DIPLOMA THESIS

Grigorios P. Thanasoulas

Athens, July 2022

NATIONAL TECHINCAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
DIVISION OF COMPUTER SCIENCE

Extensions for Scheduling and Autoscaling on Kubernetes Clusters with Local Storage Considerations

DIPLOMA THESIS

Grigorios P. Thanasoulas

Supervisor Nectarios Koziris
Professor NTUA

Approved by the three-member examination committee on the 14th of July 2022.

. . . . .	. . . . .	. . . . .
Nectarios Koziris	Georgios Goumas	Dionysios Pnevmatikatos
Professor NTUA	Associate Professor NTUA	Professor NTUA

Athens, July 2022

. . . . . .

Γρηγόριος Π. Θανάσουλας
Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

στους γονείς μου,

στα αδέλφια μου,

στα παιδιά που ονειρεύονται

Η στοιχειοθεσία του κειμένου έγινε με το XeTeX 0.999992.

Χρησιμοποιήθηκαν οι γραμματοσειρές Minion Pro, Myriad Pro και `Consolas`.

Περίληψη

Ο Kubernetes είναι η de facto επιλογή ενορχηστρωτή containers για κάθε εταιρεία που χρησιμοποιεί cloud native εφαρμογές, καθώς δύναται να ενορχηστρώνει αποδοτικά ένα μεγάλο αριθμό containers μέσω μιας ισχυρής δηλωτικής διεπαφής διαχείρισης, μειώνοντας έτσι τις λειτουργικές επιβαρύνσεις για τους διαχειριστές των συστοιχιών.

Η διεπαφή αποθήκευσης του Kubernetes επιτρέπει την ενσωμάτωση διαφορετικών αποθηκευτικών συστημάτων, τα οποία εν συνεχεία μπορούν να χρησιμοποιηθούν ως μόνιμοι τόμοι από το φορτίο εργασίας. Η χρήση τοπικών μόνιμων τόμων έναντι απομακρυσμένου μόνιμου αποθηκευτικού χώρου προσφέρει το πλεονέκτημα των υψηλών επιδόσεων: οι τοπικοί δίσκοι προσφέρουν υψηλότερο αριθμό IOPS, μεγαλύτερους ρυθμούς μετάδοσης και χαμηλότερη καθυστέρηση σε σύγκριση με τα απομακρυσμένα συστήματα αποθήκευσης.

Επί του παρόντος, ο Cluster Autoscaler δεν υποστηρίζει την αυτόματη κλιμάκωση σε συστοιχίες με τοπικούς μόνιμους τόμους, ενώ ο Scheduler δεν λαμβάνει υπόψη του την ελεύθερη χωρητικότητα στους τοπικούς δίσκους των κόμβων κατά τη χρονοδρομολόγηση των Pods.

Η απρόσκοπτη λειτουργία του Cluster Autoscaler και του Scheduler σε συστοιχίες Kubernetes με τοπική αποθήκευση είναι πολύ σημαντική για τις εταιρείες, καθώς τους επιτρέπει να έχουν συστοιχίες, που αξιοποιούν τα πλεονεκτήματα της τοπικής αποθήκευσης, ενώ προσαρμόζεται δυναμικά το μέγεθός τους και το φορτίο εκτελείται αδιάλειπτα.

Στην παρούσα διπλωματική εργασία προτείνουμε και υλοποιούμε επεκτάσεις για τον Cluster Autoscaler και τον Scheduler, προκειμένου να λειτουργούν απρόσκοπτα με τον τοπικό αποθηκευτικό χώρο. Κατά τη διάρκεια αυτής, λοιπόν, εγκαταστήσαμε με επιτυχία τις επεκτάσεις του Cluster Autoscaler και Scheduler στις συστοιχίες διαφόρων εταιρειών. Επιπλέον, ξεκινήσαμε να συνεισφέρουμε τμήματα του προτεινόμενου σχεδιασμού upstream.

Λέξεις-Κλειδιά

Kubernetes, Cluster Autoscaler, Scheduler, Χρονοδρομολόγηση, Τοπική Αποθήκευση, Συστοιχία

Abstract

Kubernetes is the de facto container orchestrator choice for every company going cloud-native. It can efficiently orchestrate a large number of containers via a powerful declarative management interface, reducing operational burdens for the cluster admins.

The Kubernetes storage interface allows the integration of different storage systems, which can get used as persistent volumes by the workload. The use of local persistent volumes over remote persistent storage offers the benefit of performance: local disks offer higher IOPS and throughput and lower latency compared to remote storage systems.

Currently, the Cluster Autoscaler does not support autoscaling with local storage. Moreover, the Scheduler does not consider the available capacity of local storage when scheduling Pods.

Enabling the seamless operation of the Cluster Autoscaler and Scheduler on Kubernetes clusters that use local storage systems is crucial for enterprises to reduce costs. The local storage will enable their disk-intensive workload to complete faster, and the Scheduler will ensure that the workload units run on nodes that have the requested storage capacity. At the same time, the Cluster Autoscaler will scale the cluster at an appropriate size for the workload to run without any excess resource waste.

In this diploma thesis, we propose and implement extensions for the Cluster Autoscaler and the Scheduler to seamlessly operate with local storage. During this thesis, we deployed the proposed extended Cluster Autoscaler and Scheduler to enterprises, and they used it successfully at large production clusters. Moreover, we started contributing parts of the proposed design upstream.

Keywords

Kubernetes, Cluster Autoscaler, Scheduler, Local Storage, Logical Volume Manager, Capacity Tracking

Αντί Προλόγου

Πριν προχωρήσω, θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα της διπλωματικής μου εργασίας, καθηγητή κ. Νεκτάριο Κοζύρη, ο οποίος με ενέπνευσε να ασχοληθώ περαιτέρω με τον τομέα τον Υπολογιστικών Συστημάτων.

Θα ήθελα ακόμη να εκφράσω τις ευχαριστίες μου στον διδάκτορα κ. Βαγγέλη Κούκη, για την ευκαιρία που μου έδωσε να εκπονήσω τη διπλωματική στο περιβάλλον της Arrikto, για την καθοδήγηση και την ενθάρρυνση να εμβαθύνω στο συγκεκριμένο θέμα.

Η Arrikto υπήρξε καταλυτικός παράγοντας για τη διαμόρφωση του τρόπου σκέψης μου ως μηχανικός και γι' αυτό θα ήθελα να ευχαριστήσω τα άτομα της εταιρείας για το υπέροχο κλίμα, την αλληλοϋποστήριξη, την παρότρυνσή τους και τη διάθεση για μάθηση. Επιπροσθέτως, θα ήθελα να αναφερθώ ξεχωριστά στους Ηλία Τσιτσιμπή και Νίκο Τσιρώνη, με τους οποίους συνεργάστηκα στενά στο πλαίσιο της παρούσας διπλωματικής και υπήρξαν επάξιοι καθοδηγητές, με συμβουλές σε κάθε κρίσιμο σημείο.

Τέλος, θέλω να ευχαριστήσω τους γονείς μου για τη διαρκή υποστήριξη και την αγάπη τους, καθώς και τ' αδέλφια μου και τους φίλους μου, που υπήρξαν η πιο ευχάριστη συντροφιά στην πορεία μου όλα αυτά τα χρόνια.

Ιρηγόριος Θανάσουλας
Ιούλιος 2022

Contents

List of Algorithms

List of Acronyms

CA Cluster Autoscaler

HPA Horizontal Pod Autoscaler

VPA Vertical Pod Autoscaler

K8S Kubernetes

CR Custom Resource

CRD CustomResourceDefinition

PVC PersistentVolumeClaim

PV PersistentVolume

PDB PodDisruptionBudget

LVM Logical Volume Manager

VG Volume Group

LV Logical Volume

List of Figures

1 Εισαγωγή

Σε αυτό το κεφάλαιο περιγράφουμε συνοπτικά το αντικείμενο της εργασίας μας. Παρουσιάζουμε μία σύντομη επισκόπηση της δουλειάς μας και εντοπίζουμε τις όποιες αδυναμίες υπάρχουν στις τρέχουσες προσεγγίσεις. Συνεχίζοντας, δίνουμε μία βασική σύνοψη του μηχανισμού που χτίσαμε. Τέλος, παρουσιάζουμε τη δομή αυτής της διπλωματικής εργασίας.

1.1 Κίνητρο

Ο Kubernetes είναι η de facto επιλογή ενορχηστρωτή containers για κάθε εταιρεία που αναπτύσσει cloud native εφαρμογές. Η δημοτικότητα του Kubernetes προκύπτει από το γεγονός ότι κάνει τη διαχείριση του κύκλου ζωής των εφαρμογών, που βασίζονται σε containers, πολύ πιο εύκολη για τα DevOps, μέσω ενός δηλωτικού API. Οι προγραμματιστές μπορούν να χρησιμοποιήσουν αυτό το API, για να περιγράψουν την επιθυμητή κατάσταση μιας εφαρμογής με όρους Pods, Services, κλ.π., και οι ελεγκτές του Kubernetes εκτελούν άμεσα ενέργειες, για να φέρουν την παρατηρούμενη κατάσταση τους συστήματος στην επιθυμητή κατάσταση. Ο Kubernetes μπορεί να εκτελεστεί τόσο σε φυσικές εγκαταστάσεις, όσο και στους περισσότερους παρόχους cloud υπηρεσιών, όπως οι Amazon Web Services (AWS), Google Cloud και Microsoft Azure. Το γεγονός αυτό αποτελεί καθοριστικό παράγοντα για την επιτυχία του.

Με το packaging του κώδικα και των εξαρτήσεων του σε containers, κάθε ομάδα ανάπτυξης λογισμικού μπορεί να χρησιμοποιήσει τυποποιημένες μονάδες κώδικα για την εκτέλεση λειτουργιών, ενώ διευκολύνεται η κλιμάκωση των εφαρμογών σε οποιοδήπο-

τε μέγεθος. Τα containers μπορούν να χρησιμοποιηθούν για το packaging ολόκληρων εφαρμογών, έτσι ώστε να μπορούν να μεταφερθούν για εκτέλεση στο υπολογιστικό νέφος, χωρίς να χρειάζεται να υποστεί αλλαγές ο κώδικας. Ο Kubernetes λειτουργεί ως πλατφόρμα ενορχηστρωτή containers και επιτρέπει σε μεγάλο αριθμό containers να εκτελείται και να συνεργάζεται αρμονικά, ελαττώνοντας τις λειτουργικές επιβαρύνσεις για τους διαχειριστές της συστοιχίας.

Ο Kubernetes Scheduler και ο Cluster Autoscaler λειτουργούν από κοινού, για να εξασφαλίσουν ότι το φορτίο εργασίας της συστοιχίας εκτελείται αδιάλειπτα. Ο Scheduler διασφαλίζει ότι οι μονάδες φορτίου εργασίας, τα Pods, χρονοδρομολογούνται σε κόμβους της συστοιχίας που έχουν επαρκείς πόρους, όπως μνήμη, CPU, αποθηκευτικό χώρο, κ.λπ., ενώ ο Cluster Autoscaler διατηρεί τον κατάλληλο αριθμό κόμβων στη συστοιχία, κλιμακώνοντάς την προς τα πάνω (προσθήκη κόμβων) ή προς τα κάτω (αφαίρεση κόμβων), επιτρέποντας στις μονάδες φορτίου εργασίας να εκτελούνται απρόσκοπτα, χωρίς να ξοδεύονται τυχόν πλεονάζοντες πόροι. Από το τελευταίο συνεπάγεται ότι ο Cluster Autoscaler είναι ένα στοιχείο, που επιτρέπει στις εταιρείες να βελτιστοποιήσουν το κόστος της υπολογιστικής υποδομής, κλιμακώνοντας δυναμικά τον αριθμό των κόμβων στη συστοιχία με βάση το φορτίο εργασίας, ανταποκρινόμενος δηλαδή στη δυναμική ζήτηση των πόρων. Χωρίς τον Cluster Autoscaler, οι εταιρείες θα ήταν υποχρεωμένες να χρησιμοποιούν μία συστοιχία σταθερού πλήθους κόμβων, που θα είχε σαν συνέπεια είτε την ύπαρξη υποχρησιμοποιούμενων κόμβων στη συστοιχία, με αποτέλεσμα τη χρέωση των αδρανών κόμβων, είτε το φορτίο εργασίας δεν θα είχε τους απαραίτητους πόρους για να εκτελεστεί.

Ο συνδυασμός του Kubernetes με τοπικούς μόνιμους τόμους επιτρέπει στους χρήστες να έχουν πρόσβαση στον τοπικό αποθηκευτικό χώρο ενός κόμβου στη συστοιχία, μέσω της διεπαφής `PersistentVolumeClaim` του Kubernetes με έναν απλό και φορητό τρόπο. Το πρωταρχικό πλεονέκτημα των τοπικών μόνιμων τόμων σε σχέση με την απομακρυσμένη μόνιμη αποθήκευση (π.χ., network-attached volumes) είναι η επίδοση: οι τοπικοί δίσκοι προσφέρουν συνήθως υψηλότερα IOPS και χαμηλότερη καθυστέρηση σε σύγκριση με τα απομακρυσμένα συστήματα αποθήκευσης. Ενδεικτικά, με τη χρήση δίσκων μη πτητικής μνήμης (NVMe) σε έναν κόμβο, ο τελικός χρήστης μπορεί να επωφεληθεί από μία τεράστια αύξηση των επιδόσεων κατά την εκτέλεση εφαρμογών, που απαιτούν μεγάλη χρήση του αποθηκευτικού χώρου. Στην περίπτωση των εταιρειών, ο τοπικός αποθηκευτικός χώρος τους επιτρέπει να βελτιστοποιήσουν την ταχύτητα με

την οποία εκτελούν εργασίες υψηλής απαίτησης σε operations δίσκου, όπως πειράματα μηχανικής μάθησης, εργασίες ανάλυσης μεγάλων δεδομένων κ.λπ, το οποίο αποτελεί κρίσιμο παράγοντα για τα κέρδη τους.

Ωστόσο, αυτή τη στιγμή, η εργασία με τοπικό μόνιμο αποθηκευτικό χώρο έχει ορισμένα προβλήματα, που δεν έχουν επιλυθεί: ο Scheduler δεν χρονοδρομολογεί το φορτίο εργασίας λαμβάνοντας υπόψη τον διαθέσιμο τοπικό αποθηκευτικό χώρο και επίσης, ο Cluster Autoscaler δεν μπορεί να κλιμακώσει προς τα κάτω ή προς τα πάνω συστοιχίες, όταν γίνεται χρήση τοπικών μόνιμων τόμων. Η απρόσκοπτη λειτουργία του Cluster Autoscaler σε συστοιχίες Kubernetes, που χρησιμοποιούν τοπικό αποθηκευτικό χώρο, είναι πολύ σημαντική για τις εταιρείες. Η τοπική αποθήκευση θα τους επιτρέψει να εκτελούν φορτίο εργασίας απαιτητικό σε operations δίσκου αποτελεσματικά και γρήγορα, ο Cluster Autoscaler θα διατηρεί το κατάλληλο μέγεθος της συστοιχίας και ο Autoscaler θα διασφαλίσει ότι κάθε μονάδα του φορτίου εργασίας εκτελείται σε κατάλληλο είδος κόμβου. Αυτός ο τριπλός συνδυασμός λειτουργεί εμφανώς προς την κατεύθυνση της μείωσης του κόστους υποδομής, πράγμα το οποίο είναι επιδίωξη κάθε εταιρείας.

Με κίνητρο τη σημασία και τον αντίκτυπο της χρήσης τοπικής μόνιμης αποθήκευσης σε συστοιχίες Kubernetes, στην παρούσα διπλωματική εργασία προτείνουμε επεκτάσεις για τον Scheduler και τον Cluster Autoscaler, ώστε να καταστεί δυνατή η απρόσκοπτη λειτουργία τους με τοπική αποθήκευση. Πιο συγκεκριμένα, η εργασία μας είναι διττή:

- Προτείνουμε έναν μηχανισμό, έτσι ώστε ο Cluster Autoscaler να μπορεί να κλιμακώσει προς τα κάτω και προς τα πάνω συστοιχίες με τοπικό μόνιμο αποθηκευτικό χώρο.
- Επεκτείνουμε τον Scheduler, για να διασφαλίσουμε ότι το φορτίο εργασίας χρονοδρομολογείται σε κόμβους, που διαθέτουν την απαιτούμενη ποσότητα τοπικού αποθηκευτικού χώρου.

1.2 Διατύπωση Προβλήματος

Όπως εξηγήσαμε στην ενότητα 1.1, η παρούσα διπλωματική εργασία έχει ως κίνητρο τα οφέλη που προσφέρει η τοπική μόνιμη αποθήκευση σε συνδυασμό με τον Kubernetes.

Ωστόσο, η λειτουργία του Kubernetes Scheduler και του Cluster Autoscaler, με την τοπική αποθήκευση παρουσιάζει ορισμένα προβλήματα, τα οποία δεν έχουν επιλυθεί ακόμη:

- Ο Scheduler δεν χρονοδρομολογεί το φορτίο εργασίας λαμβάνοντας υπόψη τον διαθέσιμο αποθηκευτικό χώρο. Λόγω αυτού, ένα Pod μπορεί να κολλήσει χωρίς ποτέ να χρονοδρομολογηθεί σε κατάλληλο κόμβο, αφού ο Scheduler αναγκάζεται να επιλέξει στα τυφλά (όσον αφορά στην αποθηκευτική χωρητικότητα) κόμβο, στον οποίο, ενδεχομένως, ο τόμος δεν μπορεί να διατεθεί, επειδή το υποκείμενο σύστημα αποθήκευσης δεν έχει επαρκή χωρητικότητα.
- Ο Cluster Autoscaler δεν κλιμακώνει προς τα κάτω συστοιχίες με τοπικό μόνιμο αποθηκευτικό χώρο, καθώς θεωρεί πως τα τοπικά δεδομένα ζουν σε κάθε κόμβο και η αφαίρεση ενός κόμβου θα οδηγούσε στην απώλεια πρόσβασης στα δεδομένα αυτά.
- Ο Cluster Autoscaler δεν κλιμακώνει προς τα πάνω συστοιχίες, όταν δεν υπάρχει αρκετός τοπικός αποθηκευτικός χώρος για την εκτέλεση του φορτίου εργασίας, καθώς δεν διαθέτει μηχανισμό να ενημερωθεί για την αποθηκευτική χωρητικότητα ενός κόμβου, που θα προσθέσει.

Στην παρούσα διπλωματική εργασία κάνουμε σχεδιαστικές προτάσεις και τις υλοποιούμε, προκειμένου να ξεπεράσουμε αυτά τα προβλήματα και να καταστεί δυνατή η απρόσκοπτη λειτουργία του Scheduler και του Cluster Autoscaler με τοπικό μόνιμο αποθηκευτικό χώρο.

1.3 Υπάρχουσες Λύσεις

Επί του παρόντος, ο Cluster Autoscaler δεν υποστηρίζει αυτόματη κλιμάκωση για κόμβους που χρησιμοποιούν τοπικό μόνιμο αποθηκευτικό χώρο. Ο Cluster Autoscaler θεωρεί ότι κάθε τόμος που βασίζεται σε τοπική μόνιμη αποθήκευση είναι προσβάσιμος μόνο από αυτόν τον κόμβο και, ως εκ τούτου, δεν θα αφαιρέσει τον κόμβο όπου ζουν τα δεδομένα. Με άλλα λόγια, η κλιμάκωση προς τα κάτω με τοπικούς τόμους την παρούσα στιγμή είναι αδύνατη. Επιπλέον, δεν υπάρχει δυνατότητα κλιμάκωσης προς τα πάνω όταν δεν υπάρχει αρκετός τοπικός αποθηκευτικός χώρος για το φορτίο εργασίας

που τον ζητά, καθώς δεν διαθέτει μηχανισμό να ενημερωθεί για τη χωρητικότητα ενός κόμβου που θα προσθέσει. Την παρούσα στιγμή δεν υπάρχουν λύσεις για τη διόρθωση αυτών των προβλημάτων.

Όσον αφορά τον Scheduler, η κοινότητα του Kubernetes έχει υλοποιήσει μια αρχική υποστήριξη για χρονοδρομολόγηση με τοπικό αποθηκευτικό χώρο. Το χαρακτηριστικό αυτό ονομάζεται "*Παρακολούθηση Αποθηκευτικής Χωρητικότητας*" και επιτρέπει στους οδηγούς συστημάτων αποθήκευσης να δημοσιεύουν πληροφορίες σχετικά με την εναπομένουσα χωρητικότητα σε κάθε τμήμα τοπολογίας της συστοιχίας. Στη συνέχεια, ο Scheduler χρησιμοποιεί αυτές τις πληροφορίες για να επιλέξει έναν κατάλληλο κόμβο για κάθε Pod που ζητάει την παροχή τόμων. Ωστόσο, η τρέχουσα προσέγγιση έρχεται με ορισμένους περιορισμούς:

- Δεν επιχειρεί να μοντελοποιήσει τον τρόπο με τον οποίο οι αποφάσεις χρονοδρομολόγησης επηρεάζουν την αποθήκευση χωρητικότητα. Αυτή είναι μια σχεδιαστική απόφαση της ομάδας του Kubernetes με στόχο να διευκολύνει την ανάπτυξη του feature, δεδομένου ότι η επίδραση των αποφάσεων χρονοδρομολόγησης στη διαθέσιμη χωρητικότητα μπορεί να διαφέρει σημαντικά ανάλογα με τον τρόπο που το σύστημα αποθήκευσης χειρίζεται την αποθήκευση. Ως συνέπεια αυτής της απόφασης, με τον τρέχοντα σχεδιασμό, ένα Pod που ζητά την παροχή πολλαπλών τόμων μπορεί να ανατεθεί σε έναν κόμβο όπου υπάρχει αρκετός χώρος μόνο για καθένα από τους τόμους ξεχωριστά, χωρίς να λαμβάνεται υπόψη η συνολική ποσότητα του αποθηκευτικού χώρου που απαιτείται για τη φιλοξενία όλων των τόμων. Αυτό μπορεί να οδηγήσει στο σενάριο όπου η χρονοδρομολόγηση αυτού του Pod μπορεί να αποτύχει μόνιμα: ένας τόμος μπορεί να δημιουργηθεί με επιτυχία σε ένα τμήμα τοπολογίας, στο οποίο στη συνέχεια δεν θα απομείνει αρκετή χωρητικότητα για τον άλλο τόμο. Τότε, κάθε μελλοντική προσπάθεια χρονοδρομολόγησης του Pod θα περιορίζεται από τον τόμο που δημιουργήθηκε ήδη. Σε αυτή την περίπτωση, είναι απαραίτητη η χειροκίνητη παρέμβαση για την ανάκτηση από αυτή την κατάσταση, για παράδειγμα αυξάνοντας τη χωρητικότητα ή διαγράφοντας τον τόμο που είχε ήδη δημιουργηθεί.
- Η λειτουργία Παρακολούθησης Αποθηκευτικής Χωρητικότητας μπορεί να ενεργοποιηθεί μόνο σε συστοιχίες που εκτελούν έκδοση Kubernetes 1.21 ή νεότερη. Αυτή είναι μια απαίτηση που δεν ικανοποιούν όλες οι συστοιχίες σε περι-

βάλλον παραγωγής. Αρκετές εταιρείες χρησιμοποιούν προηγούμενες εκδόσεις Kubernetes για λόγους σταθερότητας. Στην περίπτωσή μας, οι πελάτες μας χρησιμοποιούν Kubernetes 1.19 και 1.20, όπου η λειτουργία είναι μη διαθέσιμη.

1.4 Προτεινόμενη Λύση

Όπως έχει ήδη εξηγηθεί σε προηγούμενες ενότητες, ο στόχος της παρούσας διπλωματικής είναι να επιτρέψει την απρόσκοπτη αυτόματη κλιμάκωση και χρονοδρομολόγηση σε συστοιχίες που αξιοποιούν τοπική μόνιμη αποθήκευση. Για να το επιτύχουμε αυτό, ο προτεινόμενος σχεδιασμός μας περιλαμβάνει την επέκταση διαφόρων μερών: του Scheduler, του Cluster Autoscaler και του οδηγού τοπικής αποθήκευσης Rok CSI της Arrikto.

Συνοπτικά, προτείνουμε τον ακόλουθο σχεδιασμό:

- Επεκτείνουμε τον οδηγό τοπικής αποθήκευσης Rok CSI ώστε να αναφέρει τη διαθέσιμη χωρητικότητα αποθήκευσης σε κάθε κόμβο της συστοιχίας. Το πρόγραμμα οδήγησης θα υπολογίζει τον διαθέσιμο αποθηκευτικό χώρο υποβάλλοντας εντολές στον υποκείμενο Local Volume Manager, ενώ θα λαμβάνει υπόψη του και τον χώρο που χρειάζεται να δεσμευθεί για την αποθήκευση των μεταδεδομένων του τόμου.

- Επεκτείνουμε τον Scheduler ώστε να λαμβάνει υπόψη την αναφερόμενη διαθέσιμη χωρητικότητα αποθήκευσης κάθε κόμβου κατά την χρονοδρομολόγηση ενός Pod που ζητά την παροχή τόμων τοπικού αποθηκευτικού χώρου του Rok.

- Εγκαθιστούμε στη συστοιχία τον επεκταμένο Scheduler, (τον οποίο εφεξής αποκαλούμε "*Rok Scheduler*") μαζί με ένα mutating webhook, το οποίο θα μεταλλάσσει τα Pods ώστε να αναφέρουν ότι πρέπει να χρονοδρομολογηθούν από τον Rok Scheduler.

- Αξιοποιούμε τον μηχανισμό του οδηγού αποθήκευσης Rok CSI για τη δημιουργία snapshots και για την προστασία των τοπικών τόμων ενός κόμβου όταν ένας κόμβος της συστοιχίας πρόκειται να αφαιρεθεί, ώστε να μπορούμε να κλιμακώσουμε προς τα κάτω τη συστοιχία.

- Επεκτείνουμε τον Cluster Autoscaler ώστε κλιμακώνει προς τα κάτω την συστοιχία με τα τοπικά δεδομένα και να συντονίζεται με τον μηχανισμό του Rok CSI που τα προστατεύει και λαμβάνει τα αντίγραφα ασφαλείας τους.

- Επεκτείνουμε τον Cluster Autoscaler ώστε να λαμβάνει υπόψη τη χρησιμοποίηση του αποθηκευτικού χώρου όταν αποφασίζει να αφαιρέσει έναν κόμβο, καθώς και να ελέγχει αν υπάρχει αρκετός αποθηκευτικός χώρος σε άλλους κόμβους για να φιλοξενήσει τα Pods των κόμβων που σκοπεύει να αφαιρέσει.

- Επεκτείνουμε τον Cluster Autoscaler ώστε να λαμβάνει υπόψη τον διαθέσιμο αποθηκευτικό χώρο στους κόμβους που προσθέτει κατά το την κλιμάκωση προς τα πάνω, δεδομένου ότι μια συστοιχία μπορεί να έχει διαμορφωμένες ομάδες κόμβων με διαφορετικό αποθηκευτικό χώρο.

- Επεκτείνουμε τον Cluster Autoscaler ώστε να μην αφαιρεί κόμβους μιας συστοιχίας που μπορεί να βρίσκονται σε κατάσταση Unready μετά από κάποιο σφάλμα, δεδομένου ότι τα τοπικά δεδομένα εξακολουθούν να ζουν εκεί και ενδεχομένως απαιτούνται ενέργειες από το διαχειριστή της συστοιχίας για την ανάκτηση από αυτή την κατάσταση.

1.5 Δομή Διπλωματικής Εργασίας

Το ελληνικό τμήμα της διπλωματικής αυτής εργασίας αποτελεί μία σύντομη περίληψη των αντίστοιχων αγγλικών κεφαλαίων. Ενώ προσπαθούμε να δώσουμε μία επαρκή εικόνα για το σύνολο της εργασίας μας, κάποιες λεπτομέρειες καθώς και οι πλήρεις αλγόριθμοι παραλείπονται, και παρατίθενται μόνο στο αγγλικό τμήμα της εργασίας.

Τα επόμενα κεφάλαια του ελληνικού τμήματος της εργασίας, οργανώνονται ως εξής:

- Στο **Κεφάλαιο 2** παρέχουμε το θεωρητικό υπόβαθρο που είναι απαραίτητο για να κατανοήσει ο αναγνώστης την εργασία μας.

- Στο **Κεφάλαιο 3** αναλύουμε τον σχεδιασμό του Kubernetes Scheduler και του Cluster Autoscaler, εκθέτουμε τμήματα του μηχανισμού τους και προτείνουμε σχεδιαστικές αλλαγές για να καταστεί δυνατή η απρόσκοπτη λειτουργία τους με τοπική μόνιμη αποθήκευση.

- Στο **Κεφάλαιο 4** αναλύουμε την υλοποίηση του σχεδιασμού μας.
- Στο **Κεφάλαιο 5** παρέχουμε μια περίληψη των συνεισφορών μας καθώς και πιθανές μελλοντικές κατευθύνσεις.

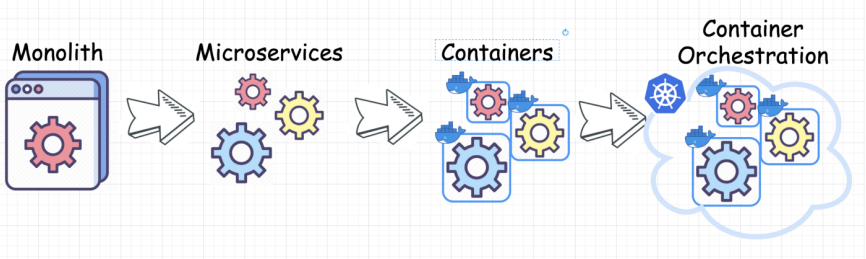
2 Υπόβαθρο

Σε αυτό το κεφάλαιο παραθέτουμε το απαραίτητο θεωρητικό υπόβαθρο για την κατανόηση των κεντρικών ιδεών της διπλωματικής εργασίας.

2.1 Η Εξέλιξη της Αρχιτεκτονικής των Εφαρμογών

Η δημοτικότητα των μικροϋπηρεσιών και του containerization έχει εκραγεί τα τελευταία χρόνια. Η ανάγκη για κλιμακούμενες εφαρμογές που αναπτύσσονται εύκολα, έχει οδηγήσει στην εγκατάλειψη των μονολιθικών αρχιτεκτονικών, όπου όλες οι διαδικασίες είναι στενά συνδεδεμένες και εκτελούνται ως μια ενιαία υπηρεσία, προς όφελος των μικροϋπηρεσιών. Οι μικροϋπηρεσίες είναι μια αρχιτεκτονική και οργανωτική προσέγγιση στην ανάπτυξη λογισμικού, όπου το λογισμικό αποτελείται από μικρές ανεξάρτητες υπηρεσίες που επικοινωνούν μέσω καλά ορισμένων APIs.

Η τρέχουσα τάση της βιομηχανίας αναφορικά με τις μικροϋπηρεσίες είναι η χρήση του containerization για την παροχή μικρότερων, ενιαίων λειτουργικών μονάδων, οι οποίες συνεργάζονται μεταξύ τους για τη δημιουργία ευέλικτων επεκτάσιμων εφαρμογών. Το containerization είναι μια μορφή εικονικοποίησης όπου οι εφαρμογές εκτελούνται σε απομονωμένους χώρους χρηστών, που ονομάζονται container, ενώ χρησιμοποιούν το ίδιο κοινόχρηστο λειτουργικό σύστημα. Ένα container είναι ουσιαστικά ένα πλήρες συσκευασμένο και φορητό υπολογιστικό περιβάλλον. Όλα όσα χρειάζεται μια εφαρμογή για να εκτελεστεί –ο δυαδικός κώδικας, βιβλιοθήκες, αρχεία ρυθμίσεων και εξαρτήσεις– είναι ενσωματωμένα και απομονωμένα στο container.



Σχήμα 2.1: *Από μονολιθικές εφαρμογές, στις μικροϋπηρεσίες, στα containers τα οποία διαχειρίζεται ένας Container Orchestrator*

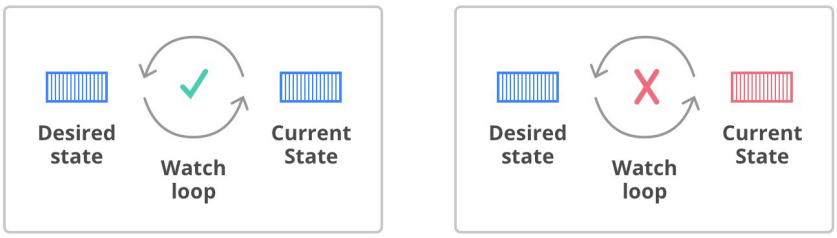
Η εκτεταμένη χρήση των containers, έχει οδηγήσει στην ανάγκη αυτοματοποίησης της ανάπτυξης και της διαχείρισης τους. Η ενορχήστρωση containers είναι η αυτοματοποίηση της προσπάθειας που απαιτείται για την εκτέλεση φορτίων εργασίας σε container και υπηρεσιών. Αυτό περιλαμβάνει ένα ευρύ φάσμα ενεργειών που χρειάζονται οι ομάδες λογισμικού για να διαχειριστούν τον κύκλο ζωής ενός container, συμπεριλαμβανομένης της παροχής, της ανάπτυξης, της κλιμάκωσης, της δικτύωσης, της εξισορρόπησης φορτίου κ.λπ.

Μεταξύ των διαφόρων ενορχηστρωτών containers, ο Kubernetes είναι η πιο δημοφιλής λύση και πλέον αποτελεί το βιομηχανικό πρότυπο. Έχοντας αρχικά ξεκινήσει από την Google, ο Kubernetes είναι μια φορητή, επεκτάσιμη, ανοιχτού κώδικα πλατφόρμα για τη διαχείριση των containerized φόρτων εργασίας και υπηρεσιών, που διευκολύνει τόσο τη δηλωτική παραμετροποίησή τους, όσο και την αυτοματοποίηση των διαδικασιών.

Ο Kubernetes επενεργεί σε μία συστοιχία υπολογιστών. Μια συστοιχία είναι ένα σύνολο από μηχανές - κόμβους για την εκτέλεση εφαρμογών που περιέχονται σε containers. Η συστοιχία είναι η καρδιά του Kubernetes και σε αυτήν οφείλεται το βασικό του πλεονέκτημα: η δυνατότητα προγραμματισμού και εκτέλεσης containers σε μια ομάδα μηχανών, είτε αυτές είναι φυσικές, είτε εικονικές, είτε on-premise ή σε κάποιο υπολογιστικό νέφος.

2.2 Kubernetes: Βασικές αρχές

Ο Kubernetes είναι ένας ενορχηστρωτής container ανοιχτού κώδικα. Αναπτύχθηκε αρχικά εσωτερικά από την Google. Από το 2014, οπότε και έγινε λογισμικό ανοιχτού κώδικα, ο Kubernetes έχει εξελιχθεί σε ένα από τα μεγαλύτερα και πιο δημοφιλή έργα



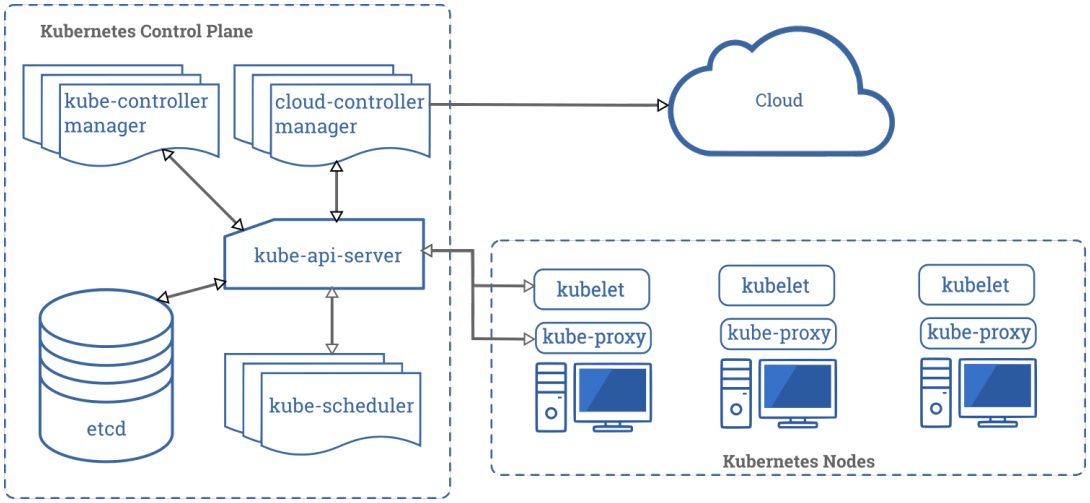
Σχήμα 2.2: *Ο βρόχος reconciliation του Kubernetes*

ανοιχτού κώδικα στον κόσμο. Έχει γίνει το πρότυπο API για τη διαχείριση cloud-native εφαρμογών, με παρουσία σχεδόν σε κάθε υπολογιστικό νέφος. Παρέχει το λογισμικό που είναι απαραίτητο για την επιτυχή ανάπτυξη και διαχείριση αξιόπιστων, κλιμακούμενων, κατανεμημένων συστημάτων.

Ο Kubernetes καθιστά τη διαχείριση του κύκλου ζωής των εφαρμογών που βασίζονται σε containers πολύ πιο εύκολη, μέσω της δηλωτικής διαχείρισης της επιθυμητής κατάστασης της συστοιχίας. Εκθέτει ένα ισχυρό δηλωτικό API, το οποίο οι προγραμματιστές μπορούν να χρησιμοποιήσουν για να περιγράψουν την επιθυμητή κατάσταση μιας εφαρμογής σε όρους Pods, Services, κ.λπ. και οι ελεγκτές του Kubernetes θα εκτελέσουν άμεσα ενέργειες για να φέρουν την παρατηρούμενη κατάσταση του συστήματος στην επιθυμητή κατάσταση.

2.2.1 Η Αρχιτεκτονική του Kubernetes

Μια συστοιχία Kubernetes αποτελείται από ένα σύνολο μηχανών εργασίας, που ονομάζονται κόμβοι, οι οποίοι εκτελούν containerized εφαρμογές. Κάθε συστοιχία διαθέτει τουλάχιστον έναν κόμβο εργασίας. Οι κόμβοι εργασίας φιλοξενούν τα Pods, τα οποία είναι η μικρότερη μονάδα φορτίου εργασίας. Το επίπεδο ελέγχου διαχειρίζεται τους κόμβους εργασίας και τα Pods στη συστοιχία. Σε περιβάλλοντα παραγωγής, το επίπεδο ελέγχου εκτελείται συνήθως σε πολλούς υπολογιστές και μια συστοιχία συνήθως λειτουργεί με πολλούς κόμβους, παρέχοντας ανοχή σε σφάλματα και υψηλή διαθεσιμότητα.



Σχήμα 2.3: *Η αρχιτεκτονική του Kubernetes*

2.2.2 Το API του Kubernetes

Ο πυρήνας του επιπέδου ελέγχου του Kubernetes είναι ο API Server. Ο API Server εκθέτει ένα HTTP API που επιτρέπει στους τελικούς χρήστες, σε διάφορα τμήματα της συστοιχίας και σε εξωτερικούς στοιχεία να επικοινωνούν μεταξύ τους. Το API του Kubernetes επιτρέπει στον χρήστη να υποβάλει ερωτήματα και να χειρίζεται την κατάσταση των αντικειμένων API στον Kubernetes (για παράδειγμα: Pods, Namespaces, ConfigMaps και Events).

Ο Kubernetes χρησιμοποιεί γενικά την κοινή ορολογία RESTful για να περιγράψει τις έννοιες του API:

- Ένας *τύπος πόρου* είναι το όνομα που χρησιμοποιείται στη διεύθυνση URL (Pods, Namespaces, Services).
- Όλοι οι τύποι πόρων έχουν μια συγκεκριμένη αναπαράσταση (το σχήμα του αντικείμενό τους) η οποία ονομάζεται *είδος*.
- Μια λίστα από instances ενός πόρου είναι γνωστή ως *συλλογή*.
- Ένα μεμονωμένο instance ενός τύπου πόρου ονομάζεται πόρος, και επίσης συνήθως αντιπροσωπεύει ένα αντικείμενο.

Σχεδόν όλοι οι τύποι πόρων αντικειμένων υποστηρίζουν τα τυπικά ρήματα HTTP - `GET`, `POST`, `PUT`, `PATCH` και `DELETE`. Ο Kubernetes χρησιμοποιεί επίσης τα δικά του ρήματα, τα οποία συχνά γράφονται με μικρά γράμματα για να τα διακρίνει από τα ρήματα HTTP. Το Kubernetes χρησιμοποιεί τον όρο "*list*" για να περιγράψει την επιστροφή μίας συλλογής

πόρων και να τη διακρίνει από την ανάκτηση ενός μεμονωμένου πόρου που ονομάζεται συνήθως "*get*".

Όλοι οι τύποι πόρων είναι είτε *cluster-scoped* από είτε σε ένα *namespace*.

2.2.3 Τα Αντικείμενα του Kubernetes

Τα *αντικείμενα* του Kubernetes είναι μόνιμες οντότητες στο σύστημα. Ο Kubernetes χρησιμοποιεί αυτές τις οντότητες για να αναπαριστά την κατάσταση της συστοιχίας σας. Συγκεκριμένα, αυτά μπορούν να περιγράψουν:

- Ποιες containerized εφαρμογές εκτελούνται και σε ποιους κόμβους.
- Τους πόρους που είναι διαθέσιμοι σε αυτές τις εφαρμογές.
- Τις πολιτικές γύρω από τον τρόπο συμπεριφοράς αυτών των εφαρμογών, όπως οι πολιτικές επανεκκίνησης, αναβαθμίσεις και ανοχή σε σφάλματα.

Ένα αντικείμενο Kubernetes είναι μια "*καταγραφή της πρόθεσης*". Μόλις ένας χρήστης δημιουργήσει το αντικείμενο, το σύστημα του Kubernetes εργάζεται διαρκώς για να διασφαλίζει την ύπαρξή του και να φέρει το σύστημα στην επιθυμητή κατάσταση που έχει δηλώσει ο χρήστης.

2.2.3.1 Το Αντικείμενο Pod

Ένα *Pod* αντιπροσωπεύει μια συλλογή από containers εφαρμογών και τους τόμους τους, που τρέχουν στο ίδιο περιβάλλον εκτέλεσης. Τα Pods, και όχι τα containers, είναι η μικρότερη μονάδα φορτίου εργασίας που μπορεί να εφαρμόσει ο χρήστης σε μια συστοιχία του Kubernetes. Αυτό σημαίνει ότι όλα τα containers σε ένα Pod καταλήγουν πάντα στο ίδιο μηχάνημα. Κάθε container μέσα σε ένα Pod τρέχει στο δικό του cgroup, αλλά μοιράζονται έναν αριθμό namespaces του Linux. Οι εφαρμογές που εκτελούνται στο ίδιο Pod μοιράζονται την ίδια διεύθυνση IP και τον ίδιο χώρο θυρών (χώρος ονομάτων δικτύου), έχουν το ίδιο hostname (UTS namespace) και μπορούν να επικοινωνούν χρησιμοποιώντας εγγενή κανάλια επικοινωνίας μεταξύ διεργασιών μέσω System V IPC ή POSIX ουρές μηνυμάτων (IPC namespace). Ωστόσο, οι εφαρμογές σε διαφορετικά Pods είναι απομονωμένες μεταξύ τους - έχουν διαφορετικές διευθύνσεις IP, διαφορετικά hostnames, κ.λπ.

Μη χρονοδρομολογήσιμα Pods Ο χρονοδρομολογητής της συστοιχίας είναι υπεύθυνος για την ανάθεση ενός κόμβου για το Pod, ώστε να τρέξει σε αυτόν. Αυτό γίνεται με τον καθορισμό του πεδίου `spec.nodeName` του Pod. Εάν ο χρονοδρομολογητής αποτυγχάνει να βρει ένα μέρος για να τρέξει το Pod, θέτει το `PodScheduled PodCondition` σε `False` και το reason σε `Unschedulable`. Ένα μη χρονοδρομολογήσιμος Pod παραμένει στη φάση `Pending`.

Στο πλαίσιο της διπλωματικής εργασίας, θα αναφερόμαστε σε ένα Pod που δεν μπόρεσε να χρονοδρομολογηθεί ως "*μη χρονοδρομολογήσιμο* Pod".

Αιτήματα πόρων και όρια Οι υπολογιστικοί πόροι είναι μετρήσιμες ποσότητες που μπορούν να ζητηθούν, να κατανεμηθούν, και να καταναλωθούν.

Ο χρήστης μπορεί να καθορίσει τα αιτήματα (requests) και τα όρια (limits) πόρων για κάθε container του Pod. Ο χρονοδρομολογητής χρησιμοποιεί τα αιτήματα για να αποφασίσει σε ποιον κόμβο θα τοποθετήσει το Pod. Το όρια ενός container, χρησιμοποιούνται από το kubelet, το οποίο τα επιβάλλει, έτσι ώστε το τρέχον container να μη χρησιμοποιήσει μεγαλύτερη ποσότητα του πόρου από το όριο που έχει ορίσει ο χρήστης στις προδιαγραφές του Pod. Το kubelet εξασφαλίζει επίσης ότι για το κάθε container θα διαθέτει την ποσότητα των πόρων που αιτήθηκε. Εάν ο κόμβος στον οποίο εκτελείται ένα Pod έχει αρκετή ποσότητα ενός πόρου διαθέσιμη, είναι δυνατό (και επιτρέπεται) ένα container να χρησιμοποιήσει περισσότερο πόρο από ό,τι ορίζει το αίτημά του για τον εν λόγω πόρο. Ωστόσο, ένα container δεν επιτρέπεται να χρησιμοποιήσει περισσότερη ποσότητα από το όριο των πόρων του.

Listing 2.1: *Αιτήματα και όρια του container ενός Pod*

```
apiVersion: v1
kind: Pod
metadata:
  name: a-pod
spec:
  containers:
  - name: app
    image: nginx
resources:
  requests:
    memory: ”100M”
    cpu: ”200m”
  limits:
    memory: ”150M”
    cpu: ”500m”
```

2.2.3.2 Το Αντικείμενο `Node`

Ο Kubernetes εκτελεί το φορτίο εργασίας τοποθετώντας Pods για να τρέξουν σε κόμβους. Ένας κόμβος μπορεί να είναι μία εικονική ή φυσική μηχανή, ανάλογα με τη συστοιχία. Κάθε κόμβος διαχειρίζεται από το επίπεδο ελέγχου, που περιέχει τις υπηρεσίες που είναι απαραίτητες για την εκτέλεση των Pods. Ένας κόμβος που έχει εγγραφεί στη συστοιχία του Kubernetes αναπαρίσταται με ένα Node αντικείμενο.

Node taints Τα taints και τα tolerations είναι ένας μηχανισμός που μπορεί να χρησιμοποιηθεί για να διασφαλιστεί ότι τα Pods δεν τοποθετούνται σε ακατάλληλους κόμβους. Τα taints προστίθενται στους κόμβους, ενώ τα tolerations ορίζονται στις προδιαγραφές του Pod. Όταν ένας χρήστης βάζει taint σε έναν κόμβο, αυτό θα απωθεί όλα τα Pods εκτός από εκείνα που έχουν tolerations για το συγκεκριμένο taint. Ένας κόμβος μπορεί να έχει ένα ή πολλά taints που σχετίζονται με αυτόν.

Node allocatable Ως *allocatable* ενός κόμβου ορίζεται η ποσότητα των υπολογιστικών πόρων που είναι διαθέσιμοι για τα Pods. Οι συνολικοί πόροι (χωρητικότητα) ενός κόμβου μπορούν να κατηγοριοποιηθούν σε:

- `kube-reserved`: δεσμευμένοι πόροι για τους δαίμονες του συστήματος kubernetes όπως το kubelet, το container runtime, κ.λπ.
- `system-reserved`: δεσμευμένοι πόροι για δαίμονες του λειτουργικού συστήματος, όπως το ssh, το udevd, κλπ.
- `eviction-threshold`: καθορίζει τα όρια που ενεργοποιούν τις εξώσεις Pods όταν οι πόροι των κόμβων πέσουν κάτω από τη δεσμευμένη τιμή.
- `allocatable`: οι εναπομείναντες πόροι κόμβων που είναι διαθέσιμοι για τη χρονοδρομολόγηση των Pods.

Node status Κάθε αντικείμενο Node έχει έναν subresource `/status` που υποδεικνύει την κατάσταση του κόμβου. Η κατάσταση περιέχει πολλαπλές συνθήκες για τον κόμβο και διαχειρίζεται από τον Node controller.

Στο πλαίσιο του Cluster Autoscaler και του Scheduler, ένας κόμβος θεωρείται "*Unready*" αν:

- Έχει το πεδίο `Pod.spec.unschedulable`. Αυτό το πεδίο υποδεικνύει ότι ο κόμβος δεν δέχεται νέα Pods.
- Το αντικείμενο του κόμβου δεν έχει καμία συνθήκη τύπου `Ready`.
- Υπάρχει μια συνθήκη τύπου `Ready` και η κατάστασή της είναι `False`.
- Μια συνθήκη τύπου `DiskPressure` ή `PIDPressure` ή `NetworkUnavailable` με την αντίστοιχη κατάστασή του να έχει οριστεί σε `True`. exists.

Οι υπόλοιποι κόμβοι θεωρούνται "*Ready*".

Ένας κόμβος που δεν μπορεί να δεχτεί pods, θα αναφέρεται ως "*μη χρονοδρομολογήσιμος*" κόμβος.

2.2.3.3 Το Αντικείμενο `PodDisruptionBudget`

Ένα `PodDisruptionBudget` (PDB) περιορίζει τον αριθμό των Pods μιας replicated εφαρμογής που μπορούν να τεθούν εκτός λειτουργίας ταυτόχρονα από εκούσιες διακοπές. Για το παράδειγμα, ένα web front end μπορεί να θέλει να διασφαλίσει ότι ο αριθμός των αντιγράφων που εξυπηρετούν το φορτίο δεν πέφτει ποτέ κάτω από ένα συγκεκριμένο ποσοστό του συνόλου.

Ένα PDB καθορίζει τον αριθμό των αντιγράφων που μπορεί να ανεχθεί να έχει μια εφαρμογή, σε σχέση με τον αριθμό που προορίζεται να έχει. Για παράδειγμα, ένα `Deployment` που έχει θέσει `.spec.replicas: 5` υποτίθεται ότι πρέπει να έχει 5 Pods ανά πάσα στιγμή. Εάν το PDB του επιτρέπει να υπάρχουν 4 κάθε φορά, τότε το Eviction API θα επιτρέψει εθελοντική διακοπή ενός (αλλά όχι δύο) Pod κάθε φορά.

2.2.3.4 Το Αντικείμενο `Deployment`

Ένας αντικείμενο `Deployment` εξασφαλίζει ότι ένας συγκεκριμένος αριθμός Pod *replicas* εκτελείται ανά πάσα στιγμή. Με άλλα λόγια, ένα Deployment διασφαλίζει ότι ένα Pod ή ένα ομοιογενές σύνολο Pods είναι πάντα σε λειτουργία και διαθέσιμο. Εάν υπάρχουν περισσότερα Pods από όσα ζητούνται, θα σκοτώσει μερικά. Αν υπάρχουν λιγότερα, ο Kubernetes θα δημιουργήσει πρόσθετα Pods.

2.2.3.5 Το Αντικείμενο `DaemonSet`

Ένας αντικείμενο `DaemonSet` (DS) διασφαλίζει ότι όλοι οι κόμβοι εκτελούν ένα αντίγραφο ενός Pod. Καθώς οι κόμβοι προστίθενται στη συστοιχία, προστίθενται Pods σε αυτούς. Καθώς οι κόμβοι αφαιρούνται από τη συστοιχία, αυτά τα Pods γίνονται garbage collect. Η διαγραφή ενός DaemonSet θα αφαιρέσει τα Pods που δημιούργησε.

2.2.3.6 Το Αντικείμενο `PersistentVolumeClaim`

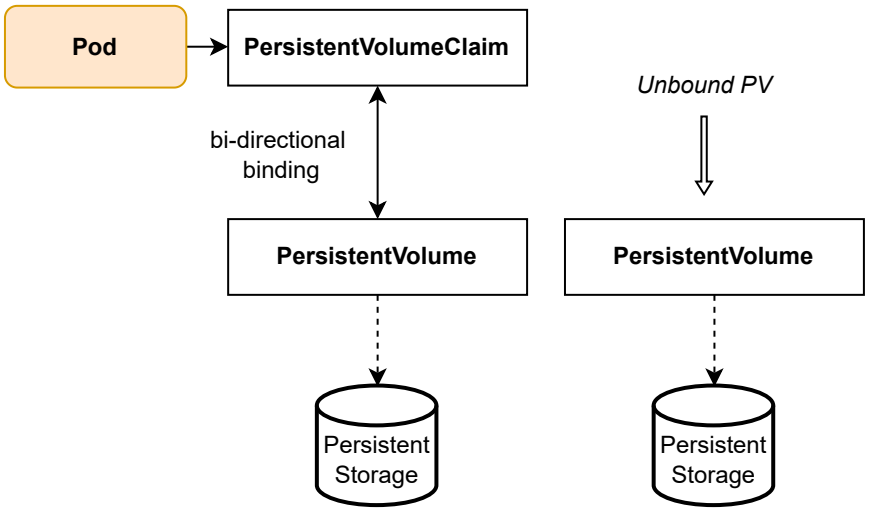
Ένα αντικείμενο `PersistentVolumeClaim` (PVC, ή ισοδύναμα αναφέρεται ως *claim*) αναπαριστά ένα αίτημα ενός χρήστη για αποθηκευτικό χώρο. Τα PVCs μπορούν να ζητήσουν συγκεκριμένο μέγεθος χώρου αλλά και τρόπους πρόσβασης, π.χ., `ReadWriteOnce`, `ReadOnlyMany` ή `ReadWriteMany`, κ.λπ.

2.2.3.7 Το Αντικείμενο `PersistentVolume`

Ένα `PersistentVolume` (PV, ή ισοδύναμα αναφέρεται ως *τόμος*) αναπαριστά τόμο του αποθηκευτικού χώρου στη συστοιχία που έχει διατεθεί στατικά από έναν διαχειριστή ή δυναμικά με τη χρήση κλάσεων αποθήκευσης. Τα PVs έχουν έναν κύκλο ζωής ανεξάρτητο από κάθε μεμονωμένο Pod που χρησιμοποιεί το PV. Αυτό το αντικείμενο API αποτυπώνει τις λεπτομέρειες του υλοποίησης του αποθηκευτικού χώρου, είτε πρόκειται για NFS, iSCSI, είτε για ένα συγκεκριμένο σύστημα αποθήκευσης.

Δέσμευση Τα PVCs είναι αιτήσεις για πόρους αποθήκευσης - κάθε PVC δεσμεύεται σε ένα PV που ταιριάζει με το ζητούμενο ποσό αποθήκευσης και τους τρόπους πρόσβασης του PVC. Κάθε PV δεσμεύεται σε ένα μόνο PVC και αντίστροφα. Η δέσμευση μεταξύ τους είναι αμφίδρομη. Ένα PV θα παραμείνει αδέσμευτο έως ότου αντιστοιχιστεί σε ένα PVC.

Node Affinity Κάθε αντικείμενο PersistentVolume έχα ένα node affinity, που υποδεικνύει του κόμβους από τους οποίους είναι προσβάσιμος ο αντίστοιχος τόμος. Το `nodeAffinity` πεδίο, είναι έναν επιλογέας ετικετών, ο οποίος επιλέγει κόμβους με βάση τις ετικέτες τους.



Σχήμα 2.4: *Ένα Pod ζητά έναν τόμο χρησιμοποιώντας ένα PVC και το PVC δεσμεύεται σε ένα PV. Το αντικείμενο PV αποθηκεύει τις λεπτομέρειες για το υποκείμενο κομμάτι μόνιμης αποθήκευσης.*

Το node affinity ενός PV χρησιμοποιείται με τον ακόλουθο τρόπο για να υποδείξει ότι ένας τόμος είναι τοπικός σε έναν κόμβο:

1. Ο οδηγός αποθήκευσης ορίζει σε κάθε αντικείμενο `Node` μια μοναδική ετικέτα.
2. Ο οδηγός αποθήκευσης θέτει το αντίστοιχο node affinity του PV να ταιριάζει μόνο με τη μοναδική ετικέτα του κόμβου.

2.2.3.8 Το Αντικείμενο `StorageClass`

Ένα `StorageClass` είναι ένας πόρος του Kubernetes που επιτρέπει τη δυναμική παροχή τόμων. Ο διαχειριστής ρυθμίζει τις παραμέτρους του StorageClass. Ένα StorageClass παρέχει έναν τρόπο στους διαχειριστές να περιγράψουν τις "κλάσεις αποθήκευσης" που προσφέρουν.

2.2.4 Το Eviction API

Κατά τη διαγραφή ενός πόρου στον Kubernetes, ο API Server θα βάλει ένα `deletionTimestamp` στο αντικείμενο του πόρου, και αν δεν υπάρχουν finalizers στο αντικείμενο, το αντικείμενο θα αφαιρεθεί από τον API Server.

Στην περίπτωση των Pods, εκτός από την κλασική λειτουργία `DELETE`, ο Kubernetes προσφέρει ένα επιπλέον API για την έναρξη της διαγραφής του Pod: το `Eviction` API.

Η κύρια διαφορά με τη λειτουργία delete, είναι ότι οι εκδιώξεις με πρωτοβουλία του API σέβονται τα ρυθμισμένα PodDisruptionBudgets και `terminationGracePeriodSeconds`.

Έτσι, εάν ένας χρήστης προσπαθήσει να *εκδιώξει* [1] ένα Pod και το αντίστοιχο PodDisruptionBudget δεν επιτρέπει τη διατάραξη του Pod, το Pod δεν θα διαγραφεί. Αντίθετα, η εκτέλεση μιας κλασικής ενέργειας `DELETE`, θα αφαιρέσει το το Pod, ανεξάρτητα από το τι ορίζει το PodDisruptionBudget.

2.2.5 Οι Διαδικασίες Cordon & Drain

Το CLI εργαλείο `kubectl` του Kubernetes, επιτρέπει στον χρήστη να εκτελεί εντολές σε συστοιχίες Kubernetes. Το εργαλείο επιτρέπει την πλήρη διαχείριση των συστοιχιών. Δύο από τις πιο σημαντικές διαδικασίες που χρησιμοποιούνται για τη συντήρηση μίας συστοιχίας είναι οι λειτουργίες *cordon* και *drain*.

Διαδικασία cordon Η διαδικασία *cordon*, η οποία προσφέρεται από το CLI εργαλείο `kubectl`, χαρακτηρίζει έναν κόμβο ως *μη χρονοδρομολογήσιμο*. Η επισήμανση ενός κόμβου ως μη χρονοδρομολογήσιμου εμποδίζει τον χρονοδρομολογητή να τοποθετήσει νέα Pods σε αυτόν τον κόμβο, αλλά δεν επηρεάζει τα υπάρχοντα Pods σε αυτόν. Αυτό είναι χρήσιμο ως προπαρασκευαστικό βήμα πριν από την επανεκκίνηση του κόμβου ή άλλη εργασία συντήρησης.

Ο διαχειριστής της συστοιχίας μπορεί να εκτελέσει τη διαδικασία cordon εκτελώντας την εντολή `kubectl cordon`. Το εργαλείο προσθέτει το *unschedulable taint* [2] στον κόμβο και θέτει επίσης το πεδίο `nodes.spec.unschedulable` σε `True`.

Διαδικασία drain Η διαδικασία *drain* χρησιμοποιείται για την απομάκρυνση του φορτίου εργασίας από έναν κόμβο, σε περίπτωση συντήρησης του κόμβου ή σε περίπτωση που ένας κόμβος πρέπει να αφαιρεθεί εντελώς από ένα cluster. Η λειτουργία αποστράγγισης θα κάνει cordon τον κόμβο για να τον χαρακτηρίσει ως μη χρονοδρομολογήσιμο και θα εκδιώξει με ασφάλεια όλα τα Pods από τον κόμβο. Οι ασφαλείς εκδιώξεις επιτρέπουν στα containers του Pod να τερματίσουν gracefully και σέβονται τα PodDisruptionBudgets που έχει ορίσει ο χρήστης.

[1] Αναφερόμαστε στην έννοια *eviction* με τον ελληνικό όρο "*εκδίωξη*".
[2] Unschedulable taint: `node.kubernetes.io/unschedulable:NoSchedule`

Ο διαχειριστής της συστοιχίας μπορεί να εκτελέσει τη λειτουργία drain εκτελώντας την εντολή `kubectl drain`. Εάν η εντολή επιστρέψει επιτυχώς, σημαίνει ότι όλα τα Pods έχουν εκδιωχθεί με ασφάλεια, τηρώντας το PodDisruptionBudget που έχει οριστεί. Στη συνέχεια, είναι ασφαλές να τερματιστεί ο κόμβος με την απενεργοποίηση της φυσικής του μηχανής ή, αν εκτελείται σε πλατφόρμα υπολογιστικού νέφους, διαγράφοντας την εικονική του μηχανή.

2.3 Kubernetes Admission Webhooks

Τα admission webhooks είναι HTTP callbacks που λαμβάνουν αιτήματα εισδοχής και κάνουν κάτι με αυτά. Μπορούν να οριστούν δύο τύποι admission webhooks: *validating admission* webhook και *mutating admission* webhook.

Τα mutating webhooks καλούνται πρώτα και μπορούν να τροποποιήσουν αντικείμενα που αποστέλλονται στον API Server για την επιβολή προσαρμοσμένων προεπιλογών. Αφού ολοκληρωθούν όλες οι τροποποιήσεις αντικειμένων, και αφού το εισερχόμενο αντικείμενο επικυρωθεί από τον API Server, καλούνται τα validating webhooks και μπορούν να απορρίψουν αιτήσεις για την επιβολή συγκεκριμένων πολιτικών αναφορικά με τα αντικείμενα που δημιουργούνται στη συστοιχία.

Ο διαχειριστής της συστοιχίας μπορεί να ρυθμίσει δυναμικά τι πόροι υπόκεινται σε ποια webhooks μέσω των αντικειμένων `ValidatingWebhookConfiguration` ή `MutatingWebhookConfiguration`.

`MutatingWebhookConfiguration` Object Κάθε `MutatingWebhookConfiguration` περιέχει μια λίστα με webhooks, που καθορίζονται στο πεδίο `webhooks`. Καθένα από τα webhooks που ορίζονται, περιλαμβάνει τα ακόλουθα πεδία:

- `rules`: Μία λίστα κανόνων που χρησιμοποιούνται για να καθοριστεί εάν ένα αίτημα προς στον API Server θα πρέπει να σταλεί στο webhook.
- `failurePolicy`: Καθορίζει τον τρόπο με τον οποίο θα χειριστεί ο API Server ένα timeout error ή οποιοδήποτε άλλο σφάλμα του webhook. Οι επιτρεπόμενες τιμές είναι `Ignore` ή `Fail`.

- `namespaceSelector`: Καθορίζει εάν θα εκτελεστεί το webhook σε ένα αντικείμενο βάσει του namespace στο οποίο ανήκει.

2.4 Kubernetes Scheduler

2.4.1 Χρονοδρομολόγηση: Βασικές Αρχές

Πολλαπλοί χρονοδρομολογητές Ένας χρονοδρομολογητής σε μία συστοιχία Kubernetes είναι ένα στοιχείο που παρακολουθεί για πρόσφατα δημιουργημένα Pods που δεν έχουν ανατεθεί σε κάποιον κόμβο. Για κάθε Pod που ο scheduler ανακαλύπτει, γίνεται υπεύθυνος για την εύρεση του καλύτερου κόμβου για το συγκεκριμένο Pod για να εκτελεστεί.

Ο `kube-scheduler` είναι ο προεπιλεγμένος χρονοδρομολογητής για τον Kubernetes και εκτελείται ως μέρος του επιπέδου ελέγχου. Ωστόσο, είναι πιθανό να εκτελούνται πολλαπλοί χρονοδρομολογητές σε μια συστοιχία. Σε αυτή την περίπτωση κάθε Pod πρέπει να καθορίσει στο spec του το όνομα του χρονοδρομολογητή που θα το χειριστεί, θέτοντας στο πεδίο `spec.schedulerName` το όνομα του προτιμώμενου scheduler. Εάν το Pod δεν έχει ορίσει ρητά το όνομα του χρονοδρομολογητή, το Pod θα χρονοδρομολογηθεί χρησιμοποιώντας τον προεπιλεγμένο χρονοδρομολογητή.

Εφικτοί κόμβοι Κάθε Pod έχει διαφορετικές απαιτήσεις, π.χ. CPU, μνήμη, που επηρεάζουν σε ποιους κόμβους μπορεί να εκτελεστεί το Pod. Οι παράγοντες που πρέπει να ληφθούν υπόψη για τις αποφάσεις χρονοδρομολόγησης περιλαμβάνουν ατομικούς και συλλογικούς πόρους, περιορισμοί υλικού/λογισμικού/πολιτικής, προτιμήσεις και μη προτιμήσεις κόμβων, τοπικότητα δεδομένων, παρεμβολές μεταξύ φορτίων εργασίας, κ.λπ. Οι κόμβοι που πληρούν τις απαιτήσεις χρονοδρομολόγησης για ένα Pod ονομάζονται εφικτοί κόμβοι. Εάν κανένας από τους κόμβους στη συστοιχία δεν είναι εφικτός, το Pod θα παραμείνει *μη χρονοδρομολογήσιμο*, δηλαδή δεν θα του ανατεθεί κάποιος κόμβος για να τρέξει.

Ο προεπιλεγμένος χρονοδρομολογητής του Kubernetes επιλέγει έναν κόμβο για το Pod σε μια διαδικασία 3 βημάτων:

1. *Filtering*: ο χρονοδρομολογητής βρίσκει το σύνολο των κόμβων όπου είναι εφικτό να χρονοδρομολογηθεί το Pod. Εκτελεί μια σειρά από πρόσθετα φιλτραρίσματος, που αξιολογούν αν το Pod μπορεί να τοποθετηθεί στον εξεταζόμενο κόμβο. Για το παράδειγμα, το φίλτρο "PodFitsResources" ελέγχει αν ένας υποψήφιος κόμβος έχει αρκετούς διαθέσιμους πόρους (CPU, μνήμη, GPU) για να καλύψει τις αιτήματα πόρων του Pod. Ένας κόμβος είναι εφικτός εάν όλα τα πρόσθετα φίλτρων θεωρούν τον κόμβο ως εφικτό για το Pod. Αυτό το βήμα υπολογίζει μια λίστα κόμβων με κατάλληλους (εφικτούς) κόμβους. Αν η λίστα αυτή είναι κενή, το Pod δεν μπορεί να χρονοδρομολογηθεί σε κάποιον κόμβο (μη χρονοδρομολογήσιμο).
2. *Scoring*: ο χρονοδρομολογητής αποδίδει μία βαθμολογία σε κάθε εφικτό κόμβο και τους κατατάσσει για να επιλέξει τον πιο κατάλληλη τοποθέτηση Pod.
3. Στη συνέχεια, ο χρονοδρομολογητής αναθέτει το Pod στον κόμβο με την υψηλότερη βαθμολογία. Εάν υπάρχουν περισσότεροι από ένας κόμβοι με ίση βαθμολογία, επιλέγει έναν από αυτούς τυχαία. Η ανάθεση το Pod σε έναν κόμβο γίνεται θέτοντας στο πεδίο `spec.nodeName` του Pod το όνομα του επιλεγμένου κόμβου.

2.4.2 Το Πλαίσιο Χρονοδρομολόγησης

Το πλαίσιο χρονοδρομολόγησης είναι μια pluggable αρχιτεκτονική για τον Kubernetes Scheduler. Προσθέτει ένα σύνολο από "πρόσθετα" APIs στον υπάρχοντα χρονοδρομολογητή. Τα πρόσθετα μεταγλωττίζονται και είναι ενσωματωμένα στον χρονοδρομολογητή. Τα APIs επιτρέπουν τα περισσότερα χαρακτηριστικά της χρονοδρομολόγησης να υλοποιούνται ως πρόσθετα, διατηρώντας παράλληλα τον πυρήνα του χρονοδρομολογητή ελαφρύ και συντηρήσιμο.

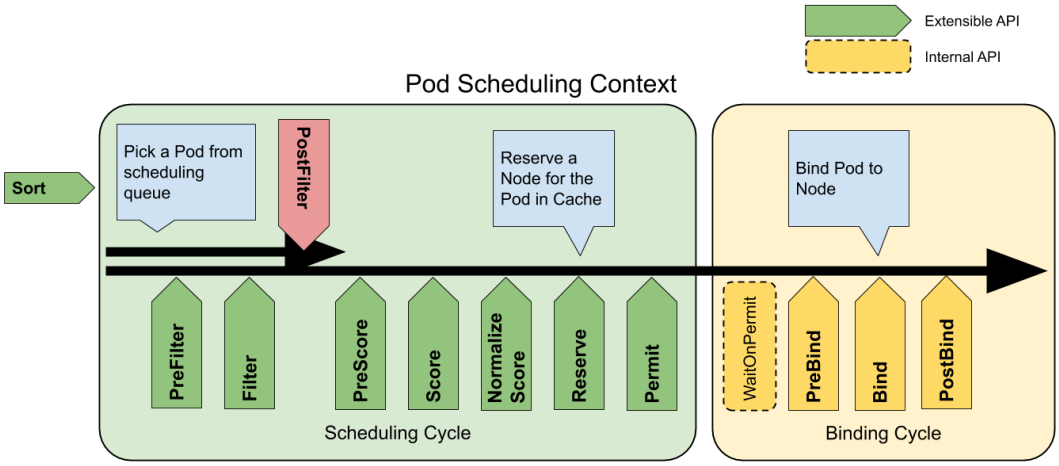
Κύκλος χρονοδρομολόγησης & κύκλος δέσμευσης Ο χρονοδρομολογητής διατηρεί μια ουρά με Pods που περιμένουν να χρονοδρομολογηθούν. Διαλέγει ένα Pod από την ουρά και επιχειρεί να το χρονοδρομολογήσει. Κάθε προσπάθεια χρονοδρομολόγησης ενός Pod χωρίζεται σε δύο φάσεις:

- *Κύκλος χρονοδρομολόγησης*: αποφασίζει έναν κόμβο για το Pod. Οι κύκλοι χρονοδρομολόγησης διαφορετικών Pods εκτελούνται σειριακά.

- *Κύκλος δέσμευσης*: εφαρμόζει αυτή την απόφαση για το Pod στη συστοιχία. Πολλαπλοί κύκλοι δέσμευσης για διαφορετικά Pods εκτελούνται παράλληλα.

Ο κύκλος χρονοδρομολόγησης και ο κύκλος δέσμευσης αναφέρονται συνολικά ως "*scheduling context*". Ένας κύκλος χρονοδρομολόγησης ή ένας κύκλος δέσμευσης μπορεί να διακοπεί εάν το Pod διαπιστωθεί ότι δεν μπορεί να ανατεθεί σε κανέναν κόμβο ή αν υπάρξει κάποιο εσωτερικό σφάλμα. Το Pod θα επιστρέψει στην ουρά αναμονής και θα γίνει προσπάθεια εκ νέου κάποια επόμενη στιγμή. Εάν ένας κύκλος δέσμευσης ματαιωθεί, θα ενεργοποιήσει τη μέθοδο `Unreserve` των πρόσθετων ώστε να απελευθερωθούν τυχόν πόροι που έχουν δεσμευτεί για το Pod και να αναστραφούν τυχόν αποφάσεις που έχουν αποτυπωθεί στη συστοιχία.

Το πλαίσιο χρονοδρομολόγησης εκθέτει ορισμένα σημεία επέκτασης. Τα πρόσθετα εγγράφονται για να κληθούν σε ένα ή περισσότερα από αυτά τα σημεία επέκτασης. Ένα πρόσθετο μπορεί να εγγραφεί σε πολλαπλά σημεία επέκτασης. Τα σημεία επέκτασης παρουσιάζονται στο Σχήμα 2.5.



Σχήμα 2.5: *Τα σημεία επέκτασης του πλαισίου χρονοδρομολόγησης*

Το πλαίσιο του χρονοδρομολογητή προσφέρει τα ακόλουθα σημεία επέκτασης, όπου κάθε πρόσθετο μπορεί να καταχωρηθεί:

- **`Queue sort`**: Αυτά τα πρόσθετα χρησιμοποιούνται για την ταξινόμηση των Pods στην ουρά του scheduler. Ένα πρόσθετο ταξινόμησης ουράς ουσιαστικά θα παρέχει μια `less(pod1, Pod2)` συνάρτηση. Μόνο ένα πρόσθετο ταξινόμησης ουράς μπορεί να είναι ενεργοποιημένο.

- **`PreFilter`**: Αυτά τα πρόσθετα χρησιμοποιούνται για την προεπεξεργασία πληροφοριών σχετικά με το Pod, ή για να ελέγξουν ορισμένες συνθήκες που η συστοιχία ή το Pod πρέπει να πληρούν. Ένα πρόσθετο PreFilter θα πρέπει να υλοποιεί μια PreFilter μέθοδο. Εάν η PreFilter μέθοδος επιστρέψει σφάλμα, ο κύκλος χρονοδρομολόγησης ματαιώνεται.

- **`Filter`**: Αυτά τα πρόσθετα χρησιμοποιούνται για το φιλτράρισμα (αποκλεισμό) των κόμβων που δεν μπορούν να εκτελέσουν το Pod, και ισοδύναμα, για την εύρεση των κόμβων που μπορούν να φιλοξενήσουν ένα Pod (*εφικτοί κόμβοι*). Για κάθε κόμβο, ο χρονοδρομολογητής θα καλέσει τα Filter πρόσθετα με τη σειρά που έχουν ρυθμιστεί. Εάν κάποιο πρόσθετο Filter χαρακτηρίσει τον κόμβο ως μη εφικτό, τα υπόλοιπα πρόσθετα δεν θα κληθούν για τον κόμβο αυτό. Οι κόμβοι μπορούν να αξιολογούνται ταυτόχρονα και, συνεπώς, ένα Filter πρόσθετο μπορεί να κληθεί περισσότερες από μία φορές στον ίδιο κύκλο χρονοδρομολόγησης.

- **`PostFilter`**: Αυτά τα πρόσθετα καλούνται μετά το Filter φάση, αλλά μόνο όταν δεν βρέθηκαν εφικτοί κόμβοι για το Pod. Τα πρόσθετα αυτά καλούνται με τη ρυθμισμένη σειρά τους. Εάν κάποιο από τα πρόσθετα PostFilter θεωρήσει τον κόμβο ως εφικτό για το Pod, τα υπόλοιπα πρόσθετα δεν θα κληθούν. Μια τυπική εφαρμογή του PostFilter είναι το preemption, που προσπαθεί να καταστήσει εφικτή τη χρονοδρομολόγηση ενός Pod με το να εκδιώξει άλλα Pods από τον κόμβο.

- **`PreScore`**: Αυτό είναι ένα σημείο επέκτασης για το την εκτέλεση εργασιών προβαθμολόγησης. Τα πρόσθετα θα κληθούν με μια λίστα κόμβων που πέρασαν τη φάση Filter (εφικτοί κόμβοι). Ένα πρόσθετο μπορεί να χρησιμοποιήσει αυτά τα δεδομένα για να ενημερώσει την εσωτερική κατάσταση ή να δημιουργήσει αρχεία καταγραφής και να υπολογίσει διάφορες μετρικές.

- **`Scoring`**:

  - Η πρώτη φάση ονομάζεται "*score*" και χρησιμοποιείται για την κατάταξη των κόμβων που έχουν περάσει τη φάση φιλτραρίσματος. Ο χρονοδρομολογητής θα καλέσει τη μέθοδο `Score` του κάθε προσθέτου βαθμολόγησης για κάθε κόμβο.

  - Η δεύτερη φάση ονομάζεται "*normalize scoring*" και χρησιμοποιείται για την τροποποίηση των βαθμολογιών πριν ο χρονοδρομολογητής υπολογίσει την τελική κατάταξη των κόμβων.

- **`Reserve`**: Ένα πρόσθετο που υλοποιεί την επέκταση Reserve έχει δύο μεθόδους, συγκεκριμένα τις `Reserve` και `Unreserve`. Τα πρόσθετα που διατηρούν την κατάσταση εκτέλεσης (*stateful πρόσθετα*) θα πρέπει να χρησιμοποιούν αυτές τις φάσεις για να ειδοποιούνται από τον χρονοδρομολογητή όταν οι πόροι σε έναν κόμβο δεσμεύονται και αποδεσμεύονται για ένα συγκεκριμένο Pod.

  Η φάση *Reserve* υπάρχει για να αποτρέψει συνθήκες ανταγωνισμού ενώ ο χρονοδρομολογητής περιμένει να πετύχει η δέσμευση. Εκτελείται πριν ο χρονοδρομολογητής δεσμεύσει πραγματικά ένα Pod στον καθορισμένο κόμβο. Η μέθοδος Reserve μπορεί να επιτύχει ή να αποτύχει.

  Εάν η μέθοδος `Reserve` όλων των πρόσθετων στοιχείων πετύχει, η φάση Reserve θεωρείται επιτυχής και το υπόλοιπο του κύκλου χρονοδρομολόγησης και ο κύκλος δέσμευσης εκτελούνται.

  Αν μια κλήση της μεθόδου `Reserve` ενός προσθέτου αποτύχει, τα επόμενα πρόσθετα δεν εκτελούνται και η φάση Reserve θεωρείται αποτυχημένη. Τότε, ο χρονοδρομολογητής θα καλέσει τη φάση `Unreserve`. Η `Unreserve` φάση υπάρχει για να καθαρίσει την κατάσταση που σχετίζεται με τις δεσμεύσεις ενός Pod. Όταν συμβαίνει αυτό, οι μέθοδοι `Unreserve` όλων των `Reserve` προσθέτων θα εκτελεστούν με την αντίστροφη σειρά από αυτή των κλήσεων των μεθόδων `Reserve`.

- **`Permit`**: Αυτά τα πρόσθετα χρησιμοποιούνται για να αποτρέψουν ή να καθυστερήσουν τη δέσμευση ενός Pod. Τα πρόσθετα Permit εκτελούνται ως το τελευταίο βήμα ενός κύκλου χρονοδρομολόγησης, ωστόσο η αναμονή στη φάση της άδειας συμβαίνει κατά την αρχή ενός κύκλου δέσμευσης, πριν εκτελεστούν τα πρόσθετα PreBind.

2.4.3 Το Πρόσθετο VolumeBinding

Το πρόσθετο `VolumeBinding` είναι ένα πρόσθετο του Kubernetes Scheduler που δεσμεύει τους τόμους Pod κατά τη χρονοδρομολόγηση. Το πρόσθετο VolumeBinding

είναι εγγεγραμμένο στα PreFilter, Filter, PreBind, Reserve, Unreserve σημεία επέκτασης του πλαισίου χρονοδρομολόγησης.

Ιδιαίτερο ενδιαφέρον στο πλαίσιο της παρούσας διπλωματικής εργασίας παρουσιάζουν οι PreFilter, Filter, PreBind φάσεις του πρόσθετου, επομένως εξηγούμε εδώ εν συντομία τις λειτουργίες που λαμβάνουν χώρα:

- `PreFilter`: ελέγχει αν από τα PVCs που αναφέρει το Pod, τα PVC με "immediate" binding mode, είναι δεσμευμένα με κάποιο PV. Εάν δεν ικανοποιείται αυτή η συνθήκη, επιστρέφεται σφάλμα.
- `Filter`: αξιολογεί αν ένα Pod μπορεί να τοποθετηθεί σε έναν κόμβο, βάσει των τόμων που ζητά, τόσο για τα δεσμευμένα όσο και για τα μη δεσμευμένα PVC:
  - Για τα *δεσμευμένα PVC*, ελέγχει ότι το PV του κάθε PVC είναι προσβάσιμο (βάσει toy node affinity που φέρει) από τον εξεταζόμενο κόμβο.
  - Για τα *μη δεσμευμένα PVC*, προσπαθεί να βρει διαθέσιμα PVs που μπορούν να ικανοποιήσουν τις απαιτήσεις του PVC και που είναι προσβάσιμα (βάσει του node affinity τους) από τον εξεταζόμενο κόμβο. Εάν δεν βρει κατάλληλα PVs, αναλαμβάνει να ενημερώσει με τα κατάλληλα annotations τον οδηγό, ώστε να γίνει δυναμική παροχή των τόμων.
  - Εάν είναι ενεργοποιημένη η παρακολούθηση της χωρητικότητας αποθήκευσης, ελέγχει εάν υπάρχει αρκετός χώρος διαθέσιμος από τον κόμβο για τους τόμους που πρέπει να δημιουργηθούν στο σύστημα αποθήκευσης.
  - Το πρόσθετο επιστρέφει `true` αν όλα τα δεσμευμένα PVC έχουν συμβατά PV με τον κόμβο και αν όλα τα μη δεσμευμένα PVC μπορούν να αντιστοιχιστούν με ένα διαθέσιμο και προσβάσιμο από τον κόμβο PV ή αν τα μη δεσμευμένα μπορούν να δημιουργηθούν δυναμικά και να αντιστοιχιστούν εν συνεχεία με το αντίστοιχο PV (dynamic provision).
- `PreBind`: Η PreBind μέθοδος θα ενημερώσει τον API Server με τις δεσμεύσεις (bindings) που υπολόγισε στα προηγούμενα βήματα και θα περιμένει έως ότου ο ελεγκτής PersistentVolume ολοκληρώσει πλήρως τη δέσμευση (bidirectional binding PV-PVC). Εάν η δέσμευση παρουσιάσει σφάλμα, λήξει ο χρόνος ή αναιρεθεί, τότε θα εμφανιστεί ένα σφάλμα και το Pod θα επιστραφεί στην ουρά για να επαναληφθεί η χρονοδρομολόγηση.

2.5 Kubernetes Cluster Autoscaler

Υπάρχουν 3 διαφορετικοί τύποι αυτόματης κλιμάκωσης στον Kubernetes:

- **Cluster Autoscaler (Autoscaler)**: ρυθμίζει τον αριθμό των κόμβων στη συστοιχία όταν τα Pods αποτυγχάνουν να χρονοδρομολογηθούν ή όταν οι κόμβοι υποχρησιμοποιούνται.
- **Horizontal Pod Autoscaler (HPA)**: ρυθμίζει τον αριθμό (replicas) των αντιγράφων μιας εφαρμογής που τρέχουν στη συστοιχία.
- **Vertical Pod Autoscaler (VPA)**: προσαρμόζει τους αιτούμενους πόρους και τα όρια των πόρων των container ενός Pod.

Στο πλαίσιο αυτής της διπλωματικής εργασίας συζητάμε μόνο για τον Cluster Autoscaler. Ο Cluster Autoscaler προσαρμόζει το μέγεθος της συστοιχίας του Kubernetes με βάση τους πόρους που αιτούνται από τα Pods και το ποσοστό αξιοποίησης των πόρων του κάθε κόμβου στη συστοιχία. Ο Cluster Autoscaler προσθέτει ή αφαιρεί αυτόματα κόμβους σε μια συστοιχία με βάση τα αιτήματα πόρων από τα Pods. Σε καμία περίπτωση δεν μετράει ζωντανά τις τιμές χρήσης της CPU και της μνήμης του Pod για να λάβει μια απόφαση κλιμάκωσης, αλλά, αντ' αυτού βασίζεται αποκλειστικά στα αιτήματα του Pod για CPU και μνήμη.

2.5.1 Αυτόματη Κλιμάκωση: Βασικές Αρχές

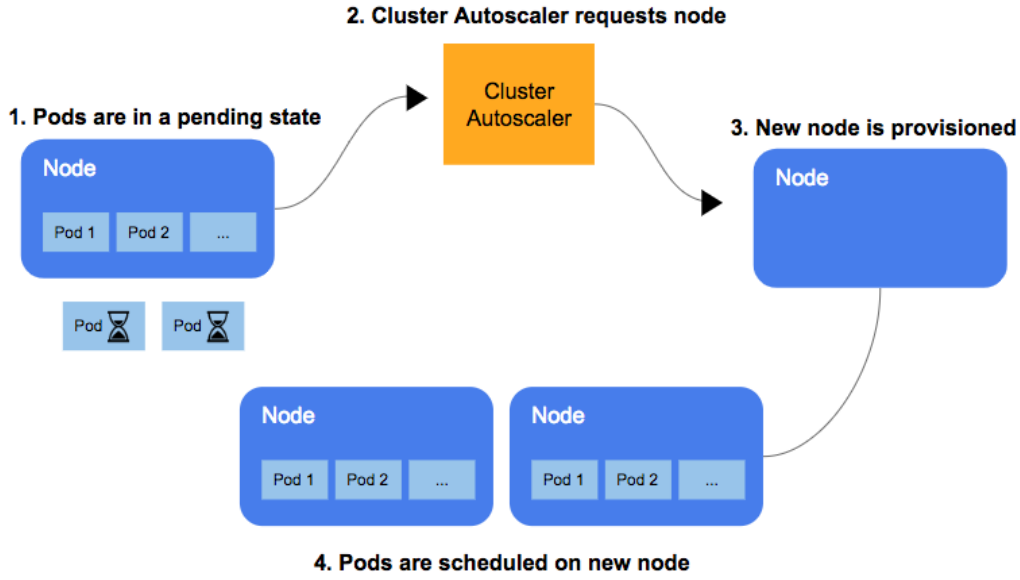
Εάν υπάρχουν Pods στη συστοιχία που δεν μπόρεσαν να χρονοδρομολογηθούν από τον Scheduler, (βρίσκονται σε κατάσταση `Pending`), ο Autoscaler θα προσθέσει έναν νέο κόμβο στη συστοιχία για να βοηθήσει το Pod να τρέξει. Αυτή η ενέργεια ονομάζεται "κλιμάκωση προς τα πάνω" (scale-up) της συστοιχίας. Μόλις προστεθεί ο κόμβος στη συστοιχία, ο Scheduler θα αναθέσει το Pod στον κόμβο αυτό και θα τρέξει.

Εάν υπάρχουν κόμβοι στη συστοιχία που δεν χρειάζονται, ο Autoscaler θα αφαιρέσει αυτούς τους κόμβους. Αυτή η ενέργεια ονομάζεται "κλιμάκωση προς τα κάτω" (scale-down) της συστοιχίας.

Στις επόμενες παραγράφους θα εξηγήσουμε τις βασικές αρχές λειτουργίας του Autoscaler.

2.5.2 Διαδικασία Κλιμάκωσης Προς Τα Πάνω

Ο Autoscaler ελέγχει περιοδικά για τυχόν Pods που δεν μπορούν να χρονοδρομολογηθούν (κατάσταση Pending) και, εάν υπάρχουν, προσπαθεί να βρει έναν νέο κόμβο με επαρκείς πόρους που θα μπορούσαν να τοποθετηθούν για να τρέξουν.



Σχήμα 2.6: *Διαδικασία κλιμάκωσης του Autoscaler*

Ο Autoscaler υποθέτει ότι η υποκείμενη συστοιχία αποτελείται από κάποια είδη ομάδων κόμβων. Μέσα σε μία ομάδα κόμβων, όλοι οι κόμβοι έχουν την ίδια ποσότητα CPU και μνήμης και έχουν το ίδιο σύνολο εκχωρημένων ετικετών. Έτσι, αυξάνοντας το μέγεθος μίας ομάδας κόμβων θα δημιουργήσει μια νέα μηχανή που θα είναι παρόμοια με αυτές που βρίσκονται ήδη στη συστοιχία.

Με βάση την παραπάνω παραδοχή, ο Cluster Autoscaler δημιουργεί πρότυπα κόμβους για κάθε ομάδα κόμβων και ελέγχει αν κάποιο από τα μη χρονοδρομολογήσιμα Pods θα χωρούσε σε έναν νέο κόμβο της ομάδας. Εάν μετά από αυτή την αξιολόγηση υπάρχουν πολλαπλές ομάδες κόμβων που, εάν αυξηθούν, θα βοηθούσαν στην εκτέλεση των μη χρονοδρομολογήσιμων Pod, μπορούν να εφαρμοστούν διαφορετικές στρατηγικές για την επιλογή της ομάδας κόμβων της οποίας το πλήθος κόμβων θα αυξηθεί.

2.5.3 Διαδικασία Κλιμάκωσης Προς Τα Κάτω

Ο Autoscaler ελέγχει περιοδικά, υπό την προϋπόθεση ότι δεν απαιτείται κλιμάκωση προς τα πάνω, ποιοι κόμβοι δεν είναι απαραίτητοι. Ένας κόμβος θεωρείται μη απαραίτητος προς αφαίρεση όταν ισχύουν όλες οι παρακάτω συνθήκες:

- Το άθροισμα των αιτήσεων CPU και μνήμης όλων των Pods που εκτελούνται σε αυτόν τον κόμβο είναι μικρότερο από το 50% των διατιθέμενων πόρων του κόμβου (ο κόμβος υποχρησιμοποιείται). Το κατώφλι αυτό είναι προσαρμόσιμο.
- Όλα τα Pods που εκτελούνται στον κόμβο μπορούν να μετακινηθούν σε άλλους κόμβους. Κατά τον έλεγχο αυτής της συνθήκης, οι νέες θέσεις όλων των μετακινούμενων Pods απομνημονεύονται. Με αυτό τον τρόπο, ο Cluster Autoscaler γνωρίζει πού μπορεί να μετακινηθεί κάθε Pod και ποιοι κόμβοι εξαρτώνται από ποιους άλλους κόμβους όσον αφορά τη μετακίνηση Pod.

Εάν ένας κόμβος είναι μη απαραίτητος για περισσότερο από 10 λεπτά (προσαρμόσιμη διάρκεια), θα τερματιστεί και θα αφαιρεθεί από τη συστοιχία. Ο Cluster Autoscaler τερματίζει έναν *μη κενό* κόμβο κάθε φορά για να μειώσει τον κίνδυνο δημιουργίας μη χρονοδρομολογήσιμων Pods. Οι *άδειοι* κόμβοι (κόμβοι που τρέχουν μόνο DaemonSet Pods) μπορούν να τερματιστούν μαζικά.

Όταν τερματίζεται ένας μη κενός κόμβος, όπως αναφέρθηκε παραπάνω, όλα τα Pods θα πρέπει να μεταφερθούν αλλού. Ο Autoscaler επιτυγχάνει να μετακινήσει τα Pods αλλού ως εξής:

1. Σημειώνει τον κόμβο στον API Server ως μη χρονοδρομολογήσιμο (δεν επιτρέπεται να χρονοδρομολογηθούν Pods σε αυτόν).

2. Για κάθε Pod του κόμβου, στέλνει αίτημα εκδίωξης στον API Server.

2.6 To Container Storage Interface

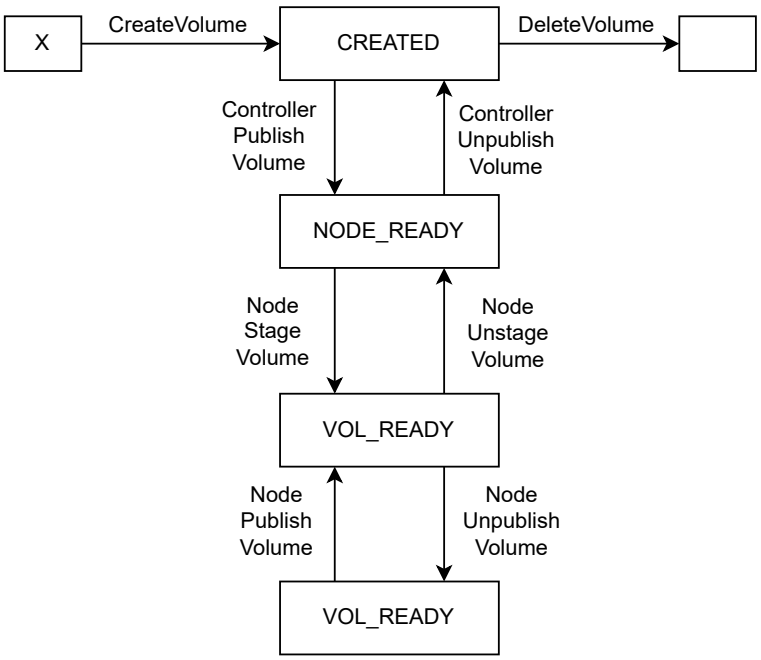
Το Container Storage Interface (CSI) είναι ένα πρότυπο για την έκθεση συστημάτων block αποθήκευσης και συστημάτων αποθήκευσης αρχείων σε φορτία εργασίας που τρέχουν σε containers, τα οποία διαχειρίζεται ένας ενορχηστρωτής containers (Container

Orchestrator - COs) όπως ο Kubernetes. Χρησιμοποιώντας το CSI οι πάροχοι αποθήκευσης τρίτων μπορούν να γράψουν και να αναπτύσσουν plugins που εκθέτουν νέα συστήματα αποθήκευσης στον Kubernetes χωρίς ποτέ να χρειαστεί να αγγίξουν τον βασικό κώδικα του Kubernetes.

2.6.1 Αρχιτεκτονική CSI Οδηγών

Ένας CO αλληλεπιδρά με ένα CSI driver Plugin μέσω κλήσεων απομακρυσμένων διαδικασιών (RPC). Κάθε CSI driver αποτελείται από τα ακόλουθα plugins:

- **Node Plugin**: Ένα gRPC endpoint που εξυπηρετεί CSI RPCs που χρειάζεται να εκτελεστούν στον κόμβο όπου θα δημοσιευτεί ο τόμος που ζητείται.
- **Controller Plugin**: Ένα gRPC endpoint που εξυπηρετεί CSI RPCs που μπορούν να εκτελεστούν οπουδήποτε.



Σχήμα 2.7: *Ο κύκλος ζωής ενός δυναμικά δημιουργούμενου τόμου, από τη δημιουργία μέχρι τη διαγραφή του*

2.6.2 CSI Remote Procedure Calls

To Container Storage Interface ορίζει τις RPCs που χρησιμοποιεί ένας ενορχηστρωτής containers προκειμένου να αλληλεπιδράσει με τον οδηγό αποθήκευσης. Κάθε μία από

τις RPCs είναι μια idempotent λειτουργία. Η σειρά με την οποία μπορούν να εκδοθούν οι κλήσεις παρουσιάζεται στο Σχήμα 2.7. Η λίστα των διαθέσιμων RPCs είναι η ακόλουθη:

- `CreateVolume`: Ο external provisioner υποβάλλει αυτή την RPC στο CSI Controller service, ζητώντας του να παρέχει έναν νέο τόμο για έναν χρήστη. Εάν το πρόσθετο δεν είναι σε θέση να ολοκληρώσει την `CreateVolume` με επιτυχία, πρέπει να επιστρέψει ένα non-OK gRPC code.

  Ιδιαίτερο ενδιαφέρον στο πλαίσιο της παρούσας διπλωματικής εργασίας παρουσιάζει ο `RESOURCE_EXHAUSTED` κωδικός. Με αυτόν τον κωδικό, υποδεικνύει ότι δεν μπορεί να παράσχει τον αιτούμενο τόμο με τους καθορισμένους περιορισμούς τοπολογίας, ενδεχομένως λόγω ανεπαρκούς χωρητικότητας αποθήκευσης.

- `ControllerPublishVolume`: Ο external attacher υποβάλλει αυτή την RPC στο CSI Controller service όταν ο Kubernetes θέλει να τοποθετήσει ένα φορτίο εργασίας που χρησιμοποιεί τον (ήδη provisioned) τόμο σε ένα κόμβο. Το πρόσθετο θα πρέπει να εκτελέσει την απαραίτητη εργασία για να καταστήσει τον τόμο διαθέσιμο στο συγκεκριμένο κόμβο.

- `NodeStageVolume`: Το kubelet υποβάλλει αυτή την RPC στο CSI Node service όταν ο τόμος πρόκειται να χρησιμοποιηθεί από το πρώτο Pod στον κόμβο. Πρέπει να εκτελείται μόνο μετά την επιτυχή εκτέλεση της `NodePublishVolume`. Ουσιαστικά χρησιμοποιείται για να μορφοποιηθεί ο τόμος και να προσαρτηθεί σε ένα staging directory του κόμβου.

- `NodePublishVolume`: Το kubelet υποβάλλει αυτή την RPC στο CSI Node service όταν ένα Pod ξεκινά να εκτελείται σε έναν κόμβο. Ουσιαστικά προσαρτά τον τόμο στον κατάλογο του Pod.

- `NodeUnpublishVolume`: Το kubelet υποβάλλει αυτή την RPC στο CSI Node service για να αναιρέσει την εργασία που έχει γίνει από την αντίστοιχη `NodePublishVolume`. Ουσιαστικά αποπροσαρτά τον τόμο από το κατάλογο του Pod.

- `NodeUnstageVolume`: Το kubelet υποβάλλει αυτή την RPC προς τον CSI Node

service για να αναιρέσει την εργασία του αντίστοιχου `NodeStageVolume`. Ουσιαστικά, αποπροσαρτά τον τόμο από τον κατάλογο staging του κόμβου.

- `ControllerUnpublishVolume`: Ο external attacher υποβάλλει αυτή την RPC στο CSI Controller service για να εκτελέσει τις εργασίες που είναι απαραίτητες για να καταστήσει τον τόμο έτοιμο να καταναλωθεί από έναν διαφορετικό κόμβο. Αυτή η κλήση αναιρεί κάθε εργασία που έχει γίνει από την encoControllerPublishVolume.

- `DeleteVolume`: Ο external provisioner υποβάλλει αυτή την RPC στο CSI Controller service για να καταργήσει την παροχή ενός τόμου. Είναι η αντίστροφη λειτουργία της `CreateVolume`.

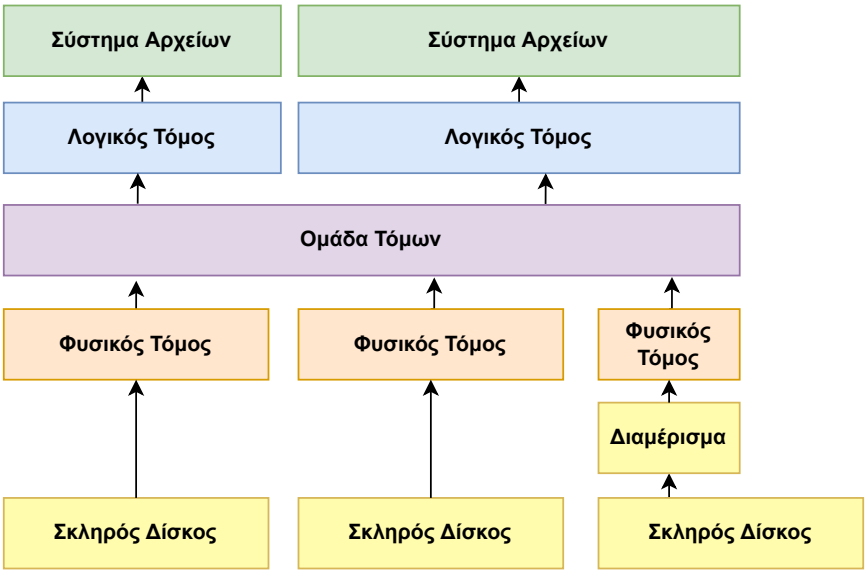
2.7 Διαχείριση Λογικών Τόμων

Η διαχείριση λογικών τόμων επιτρέπει το συνδυασμό πολλαπλών μεμονωμένων σκληρών δίσκων ή/και κατατμήσεων δίσκου σε μια ενιαία ομάδα τόμων. Αυτή η ομάδα τόμων μπορεί στη συνέχεια να υποδιαιρεθεί σε λογικούς τόμους (LV) ή να χρησιμοποιηθεί ως ένας ενιαίος τόμος. Στη συνέχεια, μπορούν να δημιουργηθούν συστήματα αρχείων, όπως EXT3 ή EXT4, σε κάθε λογικό τόμο.

Ο διαχειριστής λογικών τόμων (LVM) εισάγει ένα επιπλέον επίπεδο μεταξύ των φυσικών δίσκων και του συστήματος αρχείων επιτρέποντας στα συστήματα αρχείων να:

- Αλλάζουν μέγεθος και να μετακινούνται εύκολα, χωρίς να απαιτείται διακοπή της λειτουργίας σε όλο το σύστημα.
- Χρησιμοποιούν ασυνεχή τμήματα χώρου στο δίσκο.
- Έχουν ουσιώδη ονόματα για τους τόμους, αντί για τα συνηθισμένα κρυπτογραφημένα ονόματα συσκευών.
- Επεκτείνονται σε πολλαπλούς φυσικούς δίσκους.

Το LVM αποτελείται από μερικά εννοιολογικά επίπεδα, όπως ο φυσικός τόμος, ο λογικός τόμος και τα συστήματα αρχείων. Τα εννοιολογικά επίπεδα αποτελούνται με τη σειρά τους από μικρότερες μονάδες όπως οι φυσικές εκτάσεις (στην περίπτωση των φυσικών τόμων) και οι λογικές εκτάσεις (στην περίπτωση των λογικών τόμων).



Σχήμα 2.8: *Τα επίπεδα της διαχείρισης λογικών τόμων*

- **Φυσικός Τόμος**: Κάθε φυσικός τόμος μπορεί να είναι ένα διαμέρισμα δίσκου, ολόκληρος δίσκος, μία μετα-συσκευή ή ένα loopback αρχείο.

- **Ομάδα Τόμων**: Μια ομάδα τόμων συγκεντρώνει μια συλλογή από λογικούς τόμους και φυσικούς τόμους σε μια διοικητική μονάδα. Η ομάδα τόμων χωρίζεται σε ομάδες σταθερού μεγέθους που αποκαλούνται φυσικές εκτάσεις. Οι ομάδες τόμων αποτελούνται από φυσικούς τόμους, τα οποία με τη σειρά τους αποτελούνται από φυσικές εκτάσεις.

- **Λογικός Τόμος**: Ένας λογικός τόμος είναι το εννοιολογικό ισοδύναμο μιας κατάτμησης δίσκου σε ένα μη-LVM σύστημα. Οι λογικοί τόμοι είναι συσκευές μπλοκ οι οποίες δημιουργούνται από τις φυσικές εκτάσεις που υπάρχουν στην ίδια ομάδα τόμων.

2.8 Το Λογισμικό Rok της Arrikto

Το Rok παρέχει ένα επίπεδο διαχείρισης δεδομένων που καθιστά δυνατό για τους χρήστες να δημιουργούν στιγμιότυπα των containers τους για τοπικά και εξωτερικά αντίγραφα ασφαλείας, να λαμβάνουν αναλλοίωτα, ομαδικά συνεπή στιγμιότυπα των εφαρμογών τους και να διατηρούν αυτά τα στιγμιότυπα σε ένα αποθετήριο αντιγράφων ασφαλείας, π.χ. στο Amazon S3. Επιτρέπει στους χρήστες να δημιουργούν στιγμιότυπα,

εκδόσεις, πακέτα, να διανέμουν και να κλωνοποιούν το πλήρες περιβάλλον τους μαζί με τα δεδομένα του. Είναι εγγενώς ενσωματωμένο με το Kubernetes ως μία από τις υποστηριζόμενες πλατφόρμες του.

2.8.1 Ο Rok Operator

Οι συστοιχίες Rok συνοδεύονται από το *Rok operator*, ένα στοιχείο που υλοποιεί τον Kubernetes *operator pattern* και διαχειρίζεται τη συστοιχία. Ο Rok Operator παρακολουθεί το `rokCluster` Custom Resource και εκτελεί οποιεσδήποτε ενέργειες απαιτούνται για να φέρει την κατάσταση της συστοιχίας στο επιθυμητή κατάσταση.

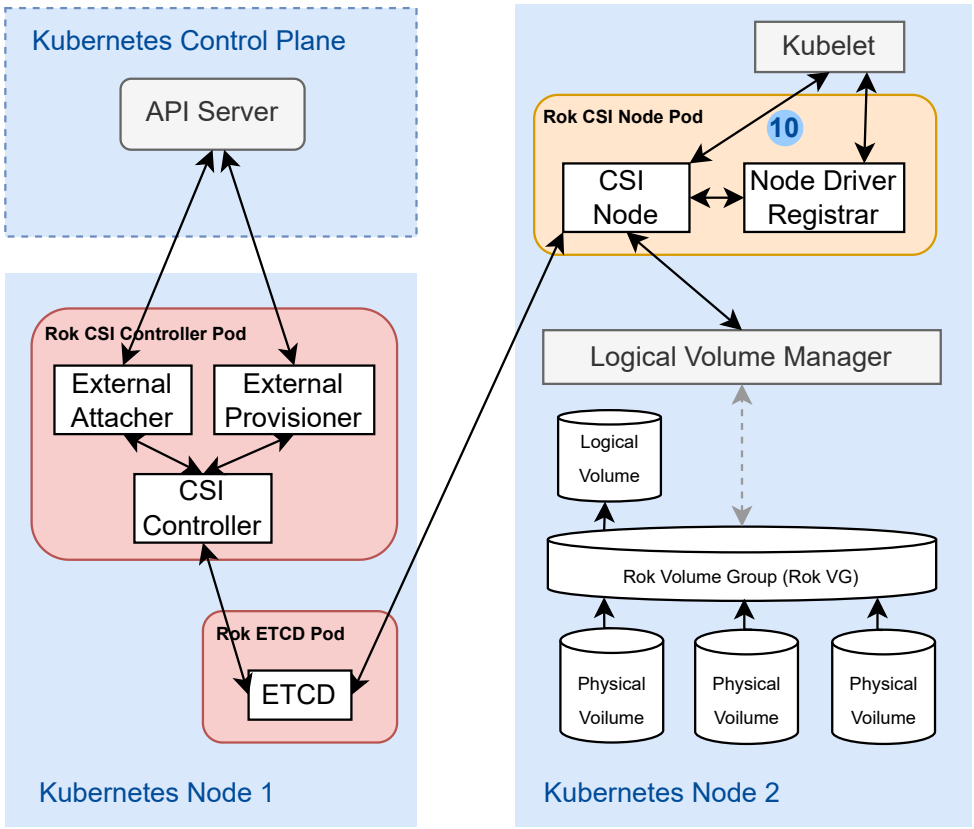
Ο χειριστής Rok είναι υπεύθυνος –μεταξύ άλλων– για την εγκατάσταση του προγράμματος οδήγησης Rok CSI στη συστοιχία, καθώς και για τη διαχείριση του μηχανισμού προστασίας δεδομένων Rok CSI (Rok CSI Guards), τον οποίο θα εξηγήσουμε στο επόμενο κεφάλαιο.

2.8.2 Το Σύστημα Αποθήκευσης του Rok

Το σύστημα αποθήκευσης του Rok συγκεντρώνει τους διαθέσιμους τοπικούς δίσκους NVMe που είναι προσαρτημένοι σε έναν κόμβο και χρησιμοποιώντας το LVM, τους συγκεντρώνει σε μια ενιαία ομάδα τόμων, που αναφέρεται ως "*Rok VG*". Το Rok VG είναι ο χώρος αποθήκευσης απ' τον οποίο παρέχονται οι τόμοι του Rok.

Ο οδηγός αποθήκευσης της Rok ενσωματώνεται με τον Kubernetes υλοποιώντας το Container Storage Interface. Θα αναφερόμαστε στον οδηγό αποθήκευσης της Rok ως το "*Rok CSI*". Το Rok CSI ακολουθεί την αρχιτεκτονική των CSI plugins και εγκαθίσταται στη συστοιχία με τα ακόλουθα αντικείμενα του Kubernetes:

- ένα DaemonSet, που εκτελεί σε κάθε κόμβο το πρόσθετο CSI Node (στο εξής αναφέρεται ως "*Rok CSI Node*"). Το Rok CSI Node διαχειρίζεται την παροχή (δημιουργία) και τη διαχείριση των τοπικών τόμων σε κάθε κόμβο.
- ένα StatefulSet, που εκτελεί σε έναν οποιονδήποτε κόμβο το πρόσθετο CSI Controller, (στο εξής αναφέρεται ως "*Rok CSI Controller*").



Σχήμα 2.9: *Η αρχιτεκτονική του συστήματος αποθήκευσης του Rok*

3 Σχεδίαση

Σε αυτό το κεφάλαιο, περιγράφουμε το σχεδιασμό και τους αλγορίθμους που καθορίζουν τη λειτουργία του Cluster Autoscaler και του Scheduler, επισημαίνουμε τις υπάρχουσες ελλείψεις όσον αφορά την τοπική αποθήκευση δεδομένων και προτείνουμε σχεδιαστικές βελτιώσεις για να καταστεί δυνατή η απρόσκοπτη χρονοδρομολόγηση και αυτόματη κλιμάκωση όταν χρησιμοποιούνται τοπικοί μόνιμοι τόμοι.

3.1 Σχεδιαστική Λογική και Στόχοι

Στόχος μας είναι να επεκτείνουμε τον Kubernetes Scheduler και τον Kubernetes Cluster Autoscaler έτσι ώστε να λειτουργούν απρόσκοπτα με φορτία εργασίας που χρησιμοποιούν τόμους με τοπικό αποθηκευτικό χώρο.

Πιο συγκεκριμένα, σκοπεύουμε:

- Να επεκτείνουμε τον Kubernetes Scheduler ώστε να λαμβάνει υπόψη του την αποθηκευτική ικανότητα κάθε κόμβου και να δρομολογεί τα Pods σε κόμβους που έχουν επαρκή ελεύθερο αποθηκευτικό χώρο για να φιλοξενήσουν τους τόμους που ζητάει το κάθε Pod.
- Να επεκτείνουμε τον Cluster Autoscaler ώστε να εκτελεί κλιμάκωση προς τα κάτω των κόμβων που εκτελούν Pods που χρησιμοποιούν τοπικούς τόμους αποθήκευσης, εξασφαλίζοντας ότι δημιουργούνται αντίγραφα ασφαλείας των δεδομένων πριν από τη διαγραφή του κόμβου από τη συστοιχία, προκειμένου να ανακτηθούν αργότερα σε έναν διαφορετικό κόμβο.

- Να επεκτείνουμε τον Cluster Autoscaler για να ελέγχει αν υπάρχει αρκετός διαθέσιμος αποθηκευτικός χώρος σε άλλους κόμβους για τη φιλοξενία των τόμων ενός Pod που εκτελείται σε έναν κόμβο, όταν ο κόμβος αξιολογείται για μείωση της κλίμακας.
- Να επεκτείνουμε τον Cluster Autoscaler ώστε κατά την κλιμάκωση προς τα πάνω να λαμβάνει υπόψη την αποθηκευτική χωρητικότητα των κόμβων που προσθέτει σε μια συστοιχία και να επιλέγει τον κατάλληλο τύπο κόμβου που διαθέτει επαρκή τοπική χωρητικότητα αποθήκευσης.

3.2 Ο Μηχανισμός Τοπικών Τόμων του Rok

Τα τοπικά δεδομένα που ζουν σε έναν κόμβο χρειάζονται ένα μηχανισμό για τη δημιουργία αντιγράφων ασφαλείας εάν ο κόμβος πρόκειται να αφαιρεθεί από τη συστοιχία, διαφορετικά τα δεδομένα θα χαθούν και δεν θα είναι δυνατή η ανάκτησή τους.

Το Rok υλοποιεί έναν μηχανισμό που επιτρέπει τη μετακίνηση τόμων μεταξύ των κόμβων κόμβων μιας συστοιχίας. Αξιοποιεί ένα εξωτερικό σύστημα αποθήκευσης, όπως το S3 της Amazon, για τη δημιουργία αντιγράφων ασφαλείας των τοπικών τόμων και δύναται να τους ανακτήσει σε οποιονδήποτε άλλο κόμβο, εφόσον ζητηθεί. Το Rok αποκαλεί τη διαδικασία μετακίνησης ενός τοπικού τόμου στο S3 ως *"unpinning"* του τόμου και τη διαδικασία επαναφοράς των δεδομένων ενός τόμου από το S3 σε έναν κόμβο ως *"pinning"* του τόμου. Περιγράφουμε αυτόν τον μηχανισμό σε μεγαλύτερο βάθος στην ενότητα που ακολουθεί.

3.2.1 Pinning και Unpinning των Τοπικών Τόμων του Rok

Όταν δημιουργείται ένας τοπικός τόμος σε έναν κόμβο, το αντίστοιχο `PersistentVolume` αντικείμενο στον API Server που αναπαριστά τον τόμο διαθέτει ένα πεδίο node affinity, το οποίο καθορίζει τους κόμβους από τους οποίους είναι προσβάσιμος ο τόμος. Στην περίπτωση του τοπικού αποθηκευτικού χώρου, το node affinity του τόμου ορίζεται έτσι ώστε να επιλέγει μόνο τον κόμβο στον οποίο βρίσκονται τοπικά τα δεδομένα του τόμου.

Το Rok εισάγει την εξής ορολογία:

- *Pinned PV*: Ένα PV που αντιπροσωπεύει έναν τόμο του οποίου τα δεδομένα ζουν τοπικά στον κόμβο. Αυτό το PV διαθέτει node affinity για να υποδεικνύει ότι είναι προσβάσιμο μόνο από τον συγκεκριμένο κόμβο.
- *Unpinned PV*: Ένα PV που αντιπροσωπεύει έναν τόμο του οποίου τα δεδομένα βρίσκονταν προηγουμένως τοπικά, αλλά πλέον έχουν μεταφερθεί στο Amazon S3. Το node affinity του PV είναι κενό για να υποδεικνύει ότι είναι προσβάσιμο από κάθε κόμβο της συστοιχίας. Ο χρονοδρομολογητής θα θεωρήσει αυτό το PV ως προσβάσιμο από κάθε κόμβο και, συνεπώς, δεν θα περιορίσει τη χρονοδρομολόγηση του αντίστοιχου Pod στον κόμβο όπου τα δεδομένα ζούσαν προηγουμένως.

Ένα pinned PV μπορεί να γίνει unpinned με τη διαδικασία του *unpinning*. Ένα unpinned PV μπορεί να γίνει pinned με τη διαδικασία του *pinning*. Ένας τόμος μπορεί να αλλάξει από pinned σε unpinned και αντίστροφα πολλές φορές, επιτρέποντας ουσιαστικά στον τόμο να μετακινηθεί σε διαφορετικούς κόμβους της συστοιχίας όσες φορές χρειάζεται.

Το Rok, και συγκεκριμένα ο ελεγκτής Rok CSI, υλοποιεί τον ακόλουθο μηχανισμό:

1. Παρακολουθεί τη συστοιχία για να βρει κόμβους που είναι unschedulable.
2. Βρίσκει τους τόμους σε κάθε μη χρονοδρομολογήσιμο κόμβο που δεν χρησιμοποιούνται από κανένα Pod.
3. Εκκινεί τη διαδικασία unpinning των τόμων αυτών: δημιουργεί με αποδοτικό τρόπο snapshots των δεδομένων του τόμου στο Amazon S3.
4. Αφαιρεί το node affinity από το PV. Ας σημειωθεί ότι το πεδίο `nodeAffinity` ενός PV είναι αμετάβλητο και για να ξεπεραστεί αυτός ο περιορισμός, το Rok διαγράφει το PV και το αναδημιουργεί στιγμιαία.

Το Rok υλοποιεί τον ακόλουθο μηχανισμό για το pinning ενός PV:

1. Ο χρονοδρομολογητής χρονοδρομολογεί ένα Pod που αναφέρεται στο unpinned PV (μέσω του PVC του).
2. Ο ελεγκτής `attachDetach` του Kubernetes δημιουργεί ένα αντικείμενο `VolumeAttachment` για να σηματοδοτήσει στον external attacher να προσαρτήσει τον τόμο στον κόμβο.
3. Ο external attacher βλέπει το VolumeAttachment και στέλνει μία κλήση ControllerPublishVolume στον ελεγκτή Rok CSI.

4. Ο ελεγκτής του Rok CSI δημιουργεί έναν λογικό τόμο στον Rok VG και επαναφέρει τα δεδομένα από το Amazon S3 στον λογικό τόμο.
5. Ο ελεγκτής του Rok CSI θέτει το κατάλληλο node affinity στο PV ώστε να υποδεικνύει ότι είναι προσβάσιμο μόνο από τον κόμβο στον οποίο έγινε η ανάκτηση του τόμου.

3.2.2 Ο Μηχανισμός Προστασίας Τοπικών Τόμων του Rok

Τα εργαλεία συντήρησης και αναβάθμισης του Kubernetes βασίζονται στη διαδικασία drain. Ουσιαστικά, πριν από οποιαδήποτε ενέργεια για την αφαίρεση ή αναβάθμιση ενός κόμβου στη συστοιχία, τα εργαλεία κάνουν drain τον κόμβο (`kubectl drain`), ώστε να επισημανθεί ως unschedulable και να εκδιώξουν με ασφάλεια όλα τα Pods του κόμβου. Ο ίδιος ο Cluster Autoscaler, χρησιμοποιεί επίσης τη λειτουργία drain πριν από την αφαίρεση ενός κόμβου.

Το Rok διαθέτει έναν μηχανισμό προκειμένου να διευκολύνει τις αναβαθμίσεις μιας συστοιχίας και να διασφαλίσει ότι οι κόμβοι δεν αφαιρούνται πριν από τη δημιουργία στιγμιότυπων όλων των τοπικών τόμων.

Ο μηχανισμός αξιοποιεί Pods με κατάλληλα PodDisruptionBudgets για να μπλοκάρει την εκδίωξή τους. Όσο αποτυγχάνει το eviction ενός Pod, η λειτουργία drain αποτυγχάνει. Ο μηχανισμός λειτουργεί ως εξής:

1. Ο Rok Operator δημιουργεί ένα αντικείμενο `Deployment` *για κάθε κόμβο* στη συστοιχία. Το Deployment του κάθε κόμβου δημιουργεί *ακριβώς ένα* replica Pod, το οποίο διαθέτει node affinity που ταιριάζει μόνο στον συγκεκριμένο κόμβο. Το Rok ονομάζει αυτά τα Pods "*CSI Guard Pod*", καθώς φυλάσσουν τον αποθηκευτικό χώρο του κόμβου.
2. Ο Rok Operator δημιουργεί ένα αντικείμενο `PodDisruptionBudget` για κάθε Rok CSI Guard Deployment. Το PodDisruptionBudget απαιτεί σε κάθε στιγμή να υπάρχει τουλάχιστον ένα Rok CSI Guard Pod του Deployment. Με αυτό τον τρόπο, όσο υπάρχει το PodDisruptionBudget, το eviction του αντίστοιχου Rok CSI Guar Pod θα αποτυγχάνει.
3. Η διαδικασία drain χαρακτηρίζει τον κόμβο ως unschedulable και ξεκινά την εκδίωξή των Pods του κόμβου. Η εκδίωξη του CSI Guard αποτυγχάνει λόγω του

PodDisruptionBudget.

4. Ο Rok Operator, ελέγχει αν όλοι οι τοπικοί τόμοι Rok στον unschedulable κόμβο έχουν γίνει unpinned. Αν ισχύει αυτή η συνθήκη, αφαιρεί το PodDisruptionBudget που αντιστοιχεί στο CSI Guard του κόμβου.
5. Εφόσον το PDB αφαιρέθηκε, επιτυγχάνει η εκδίωξή του Rok CSI Guard Pod του κόμβου και η διαδικασία drain ολοκληρώνεται επιτυχώς.

3.3 Kubernetes Scheduler

Σε αυτήν την ενότητα, θα παρουσιάσουμε συνοπτικά τη τρέχουσα σχεδίαση του Kubernetes Scheduler, θα επισημάνουμε τις ελλείψεις που υπάρχουν και θα προτείνουμε βελτιώσεις που επιλύουν τους τρέχοντες περιορισμούς.

3.3.1 Το Πρόσθετο VolumeBinding

Για λόγους συντομίας, στο ελληνικό τμήμα της διπλωματικής παρουσιάζουμε μόνο τη σχεδίαση της `Filter` και της `PreBind` φάσης του προσθέτου, καθώς σχετίζεται άμεσα με τις προτεινόμενες επεκτάσεις. Για την αναλυτική παρουσίαση των υπολοίπων φάσεων του προσθέτου, μπορείτε να ανατρέξετε στο αντίστοιχο αγγλικό κεφάλαιο, στην ενότητα 3.3.1.

PreFilter Φάση

`Filter`: αξιολογεί αν ένα Pod μπορεί να τοποθετηθεί σε έναν κόμβο, βάσει των τόμων που ζητά, τόσο για τα δεσμευμένα όσο και για τα μη δεσμευμένα PVC:

- Για τα *δεσμευμένα PVC*, ελέγχει ότι το PV του κάθε PVC είναι προσβάσιμο (βάσει toy node affinity που φέρει) από τον εξεταζόμενο κόμβο.
- Για τα *μη δεσμευμένα PVC*, προσπαθεί να βρει διαθέσιμα PVs που μπορούν να ικανοποιήσουν τις απαιτήσεις του PVC και που είναι προσβάσιμα (βάσει του node affinity τους) από τον εξεταζόμενο κόμβο. Τα PVCs για τα οποία δεν κατάφερε να βρει κατάλληλα PVs, θα τα αποκαλούμε εφεξής "*PVCs to provision*".

- Για κάθε *PVC to provision*, ελέγχει αν η `StorageClass` του PVC υποστηρίζει τη δυναμική παροχή και αν υπάρχει αρκετή χωρητικότητα αποθήκευσης προσβάσιμη από τον κόμβο. Εάν όχι, το Pod δεν μπορεί να ανατεθεί στον κόμβο. Αυτό είναι το βήμα όπου η χωρητικότητα αποθήκευσης λαμβάνεται υπόψη.

Η τρέχουσα υλοποίηση του χρονοδρομολογητή, ελέγχει αν υπάρχει αρκετή χωρητικότητα για κάθε PVC to provision, καλώντας τη μέθοδο `hasEnough()` με ένα μόνο PVC ως είσοδο. Ζητά από τον API Server όλα τα αντικείμενα `CSIStorageCapacity`. και ελέγχει αν κάποιο από αυτά ταιριάζει με το `StorageClass` του PVC, είναι προσβάσιμο από τον εξεταζόμενο κόμβο και η αναφερόμενη χωρητικότητα του αντικειμένου είναι μεγαλύτερη από τη ζητούμενη χωρητικότητα του PVC. Εάν ένα τέτοιο CSIStorageCapacity υπάρχει, υπάρχει αρκετός χώρος στον κόμβο για τη δυναμική παροχή τόμου για το εξεταζόμενο PVC.

Είναι σημαντικό να επισημάνουμε ότι δεν ελεγχει αν υπάρχει αποθηκευτικός χώρος συνολικά για όλα τα PVCs, αλλά μόνο αν το κάθε PVC χωριστά χωράει σε έναν κόμβο.

PreBind Φάση

Η φάση `PreBind` εκτελείται αφού ο χρονοδρομολογητής έχει επιλέξει έναν κόμβο για το Pod.

Για κάθε ένα από τα *μη δεσμευμένα PVC* που το πρόσθετο βρήκε ένα κατάλληλο PV κατά τη διάρκεια της φάσης `Filter`, θα ενημερώσει τον API Server με τη δέσμευση, δηλαδή, θα ενημερώσει το αντίστοιχο PV ώστε να δείχνει στο PVC, και στη συνέχεια, ο ελεγκτής Kubernetes PersistentVolume θα ολοκληρώσει την αμφίδρομη δέσμευση.

Για κάθε ένα από τα *PVCs to provision*, θα ενημερώσει τα αντίστοιχα PVCs στον API Server με το "selected node annotation" [1] για να σηματοδοτήσει στον external provisioner ότι ένας τόμος για το PVC πρέπει να δημιουργηθεί δυναμικά σε ένα τμήμα τοπολογίας που είναι προσβάσιμο από τον κόμβο που υποδεικνύει η σημείωση.

Στη συνέχεια, το πρόσθετο θα κάνει poll τον API Server έως ότου όλα τα PVCs δεσμευτούν PVs. Εάν το selected node annotation κάποιου PVC to provision αφαιρεθεί, θα ακυρώσει την τρέχουσα προσπάθεια χρονοδρομολόγησης και θα καλέσει τα πρόσθετα

[1] To selected node annotation: `volume.kubernetes.io/selected-node`

`Unreserve`.Η αφαίρεση του annotation είναι ένας μηχανισμός με τον οποίο ο external provisioner ουσιαστικά ειδοποιεί τον χρονοδρομολογητή ότι απέτυχε η παροχή του τόμου και θα πρέπει να δοκιμάσει ξανά, ενδεχομένως σε άλλον κόμβο.

3.3.2 Ελλείψεις & Προτεινόμενες Επεκτάσεις

Σύμφωνα με την προηγούμενη ανάλυση των αλγορίθμων, ο τρέχων σχεδιασμός του Kubernetes Scheduler έχει τους ακόλουθους περιορισμούς:

1. Η μέθοδος `Filter` του πρόσθετου VolumeBinding χρησιμοποιεί τα αντικείμενα CSIStorageCapacity του Kubernetes API για να αντλήσει πληροφορίες για τον διαθέσιμο αποθηκευτικό χώρο. Αυτό το αντικείμενο API έγινε beta στην έκδοση Kubernetes 1.21 και ήταν σε κατάσταση alpha σε προηγούμενες εκδόσεις. Οι κύριοι πάροχοι υπηρεσιών νέφους δεν ενεργοποιούν τα χαρακτηριστικά σε κατάσταση alpha στις υπηρεσίες τους. Ως αποτέλεσμα, τα CSIStorageCapacity αντικείμενα δεν είναι ενεργοποιημένα σε συστοιχίες που εκτελούν εκδόσεις προγενέστερες της 1.21 στους περισσότερους παρόχους cloud. Αυτό είναι ένα σημαντικό πρόβλημα, δεδομένου ότι πολλές επιχειρήσεις (συμπεριλαμβανομένων των πελατών μας) δεν τρέχουν τις τελευταίες εκδόσεις του Kubernetes για λόγους σταθερότητας. Στη δική μας περίπτωση, οι πελάτες μας εκτελούν συστοιχίες Kubernetes 1.19 και 1.20 και χρειάζονταν τη δυνατότητα χρονοδρομολόγησης Pods με εξέταση της τοπικής αποθήκευσης.

2. Η τρέχουσα σχεδιαστική λογική της φάσης `Filter` του πρόσθετου `VolumeBinding` δεν λαμβάνει υπόψη της τον αποθηκευτικό χώρο που απαιτείται για την παροχή πολλαπλών PVC ενός Pod. Αντ' αυτού, ελέγχει αν κάθε μεμονωμένο PVC μπορεί να δημιουργηθεί στον αποθηκευτικό χώρο που είναι προσβάσιμος από τον κόμβο, χωρίς να διασφαλίζει ότι υπάρχει αρκετός χώρος για όλα αυτά ταυτόχρονα. Αυτό είναι ένα κρίσιμο πρόβλημα: σε περίπτωση που ένα Pod αναφέρεται σε πολλαπλά μη δεσμευμένα PVC και δεν υπάρχει αρκετός χώρος για όλα αυτά, ένα από αυτά γίνει provision και η παροχή των υπολοίπων θα αποτύχει, τότε όλες οι μελλοντικές αποφάσεις χρονοδρομολόγησης θα περιορίζονται από το ήδη δημιουργημένο τόμο και το Pod θα κολλήσει.

Δεδομένου ότι ο σχεδιασμός του upstream έρχεται με τους προαναφερθέντες περιορισμούς, προτείνουμε να επεκτείνουμε τον Kubernetes Scheduler και να εγκαταστήσουμε τον επεκταμένο χρονοδρομολογητή στη συστοιχία. Ο προτεινόμενος σχεδιασμός μπορεί να χωριστεί στα ακόλουθα μέρη:

1. Επέκταση του Rok CSI Node του οδηγού αποθήκευσης, ώστε να αναφέρει τη διαθέσιμη χωρητικότητα κάθε κόμβου ως annotation στο αντίστοιχο αντικείμενο `Node` του Kubernetes.
2. Επέκταση του Rok CSI Controller του οδηγού αποθήκευσης ώστε να απαντά με κατάλληλο σφάλμα στην κλήση `CreateVolume` όταν η εναπομένουσα χωρητικότητα για την παροχή του τόμου είναι ανεπαρκής.
3. Επέκταση του πρόσθετου VolumeBinding του Kubernetes Scheduler ώστε να ελέγχει αν πολλαπλοί τόμοι ενός Pod χωρούν σε έναν κόμβο, συγκρίνοντας τη συνολική τους απαίτηση σε χωρητικότητα με την αναφερθείσα διαθέσιμη χωρητικότητα.
4. Εγκατάσταση του επεκταμένου χρονοδρομολογητή στη συστοιχία.
5. Ανάπτυξη και εγκατάσταση ενός webhook που θα μεταλλάσσει τα Pods ώστε να χρησιμοποιούν τον επεκταμένο χρονοδρομολογητή.

3.3.2.1 Επέκταση του Rok CSI Node

Δεδομένου ότι τα αντικείμενα `CSIStorageCapacity` δεν μπορούν να γίνουν back-port σε προηγούμενες εκδόσεις του Kubernetes και, επίσης, η προσθήκη ενός παρόμοιου Custom Resource θα απαιτούσε αρκετή προσπάθεια άνευ αιτίας, αποφασίζουμε να αναφέρουμε τη χωρητικότητα κάθε κόμβου ως annotation στο αντίστοιχο αντικείμενο Node. Το annotation, το οποίο αποκαλούμε “annotation χωρητικότητας” θα είναι της μορφής `rok.arrikto.com/capacity:<free-storage-bytes>`.

Το πρόσθετο Rok CSI Node του οδηγού αποθήκευσης που εκτελείται σε κάθε κόμβο της συστοιχίας υπολογίζει περιοδικά τον διαθέσιμο αποθηκευτικό χώρο και ενημερώνει το annotation χωρητικότητας. Δίνει εντολές στο Logical Volume Manager (LVM) του κόμβου για να μάθει τον ελεύθερο χώρο του Rok Volume Group και ενημερώνει το αντίστοιχο `Node` αντικείμενο με την τιμή της διαθέσιμης χωρητικότητας.

3.3.2.2 Επέκταση του Rok CSI Controler

Επεκτείνουμε το πρόσθετο Rok CSI Controller του οδηγού αποθήκευσης ώστε να επιστρέφει το status code `GRPCResourceExhausted` ως απάντηση στην κλήση `CreateVolume` του external provisioner όταν η παροχή ενός τόμου αποτυγχάνει λόγω ανεπαρκούς χωρητικότητας αποθήκευσης.

3.3.2.3 Επέκταση του VolumeBinding Plugin

Προτείνουμε την επέκταση της `Filter` μεθόδου του πρόσθετου `VolumeBidning` ως εξής:

1. Κατά τον έλεγχο των PVCs του Pod που χρειάζονται να δημιουργηθούν δυναμικά (provision) (μέθοδος `checkVolumeProvisions()`), να επιλέγει όλα τα Rok PVCs [2] (εφεξής αναφέρονται ως “
   textitRok claims to provision”) και να ελέγχει αν υπάρχει αρκετή χωρητικότητα για το συνολικό αποθηκευτικό χώρο που ζητούν.
2. Να ελέγχει αν υπάρχει αρκετή χωρητικότητα για τα Rok claims to provision ως εξής:
   (α′) Να υπολογίζει τη συνολική χωρητικότητα που ζητείται αθροίζοντας τα αιτήματά τους.
   (β′) Να ελέγχει αν ο εξεταζόμενος κόμβος διαθέτει annotation χωρητικότητας του Rok [3] .
   (γ′) Αν το annotation *δεν υπάρχει*, ή αν υπάρχει αλλά δεν είναι έγκυρος ακέραιος αριθμός, τα Rok claims to provision δεν μπορούν να δημιουργηθούν στον κόμβο. Η απουσία της σημείωσης υποδεικνύει ότι το πρόγραμμα οδήγησης Rok CSI δεν εκτελείται στον κόμβο.
   (δ′) Εάν υπάρχει το annotation χωρητικότητας, να ελέγχει αν η αναφερόμενη διαθέσιμη χωρητικότητα είναι μεγαλύτερη ή ίση με τη συνολική χωρητικότητα που ζητούν τα Rok claims to provision. Εάν δεν ισχύει η συνθήκη, δεν υπάρχει αρκετή χωρητικότητα, και τα Rok claims to provision δεν μπορούν

[2] PVCs provisioned by the `rok.arrikto.com` provisioner.
[3] To annotation χωρητικότητας του Rok: `rok.arrikto.com/capacity`

να δημιουργηθούν στον κόμβο, οπότε και το Pod δεν μπορεί να προγραμματιστεί στον κόμβο.

3. Διατηρούμε της συμβατότητα προς τα πίσω με τη μη τροποποίηση του χειρισμού των PVCs που δεν ζητούν αποθηκευτικό χώρο από την κλάση αποθήκευσης Rok. Ο σχεδιασμός μας, διαχωρίζει τα PVCs σε τοπικά Rok PVCs και μη Rok PVCs, και επεκτείνει μονάχα τον τρόπο χειρισμού μονάχα για τα Rok PVCs. Τα PVC που παρέχονται από άλλους παρόχους αποθήκευσης δεν θα επηρεαστούν από τις αλλαγές μας.

3.3.2.4 Εγκατάσταση του Rok Scheduler

Ο `kube-cheduler` εκτελείται από προεπιλογή σε κάθε πάροχο νέφους ως μέρος του επιπέδου ελέγχου του Kubernetes και είναι ο προεπιλεγμένος χρονοδρομολογητής που χρησιμοποιείται για τη χρονοδρομολόγηση των Pods. Οι πάροχοι νέφους αποκρύπτουν το επίπεδο ελέγχου από τον τελικό χρήστη των υπηρεσιών τους, οπότε δεν υπάρχει δυνατότητα αντικατάστασης και παραμετροποίησης του εκτελούμενου χρονοδρομολογητή.

Ως συνέπεια αυτού του περιορισμού, εγκαθιστούμε στη συστοιχία –παράλληλα με τον προεπιλεγμένο χρονοδρομολογητή– τον δικό μας χρονοδρομολογητή, που εκτελεί το επεκταμένο VolumeBinding πρόσθετο. Εφεξής θα αναφερόμαστε στον δικό μας επεκταμένο χρονοδρομολογητή ως "*Rok Scheduler*".

3.3.2.5 Εγκατάσταση του Rok Scheduler Webhook

Δεδομένου ότι εγκαθιστούμε τον Rok Scheduler χωρίς να αντικαταστήσουμε το προεπιλεγμένο Kubernetes Scheduler της συστοιχίας, κάθε Pod πρέπει να καθορίζει ποιος scheduler θα το χρονοδρομολογήσει ορίζοντας το πεδίο `spec.schedulerName`. Εάν το πεδίο δεν έχει οριστεί, ο προεπιλεγμένος χρονοδρομολογητής χρησιμοποιείται.

Σίγουρα δεν θέλουμε κάθε χρήστης να ορίζει χειροκίνητα το όνομα του scheduler στο το Pod - αυτό θα επέτρεπε στους χρήστες να παρακάμψουν την πολιτική χρονοδρομολόγησης που έχουμε ορίσει, είναι επιρρεπές σε σφάλματα και είναι μια κουραστική διαδικασία. Χρειαζόμαστε έναν αυτόματο τρόπο για να το πετύχουμε αυτό. Η λύση για την αυτοματοποίηση της εργασίας, είναι ένα mutating webhook.

Εγκαθιστούμε ένα μεταλλασσόμενο webhook στη συστοιχία, το οποίο στο εξής θα αναφέρεται ως "*Rok Scheduler webhook*", το οποίο δέχεται τα πρόσφατα δημιουργηθέντα Pods σε συγκεκριμένα namespaces της συστοιχίας και τα μεταλλάσσει προσθέτοντας το όνομα του Rok Scheduler στο πεδίο `spec.schedulerName`.

3.4 Kubernetes Cluster Autoscaler

Σε αυτή την ενότητα, θα εκθέσουμε τον σχεδιασμό του Cluster Autoscaler, θα περιγράψουμε τις κύριες αρχές λειτουργίας του, θα εντοπίσουμε τους περιορισμούς του και θα προτείνουμε επεκτάσεις που θα επιτρέψουν την απρόσκοπτη λειτουργία του με τοπικούς μόνιμους τόμους.

3.4.1 Βασικοί Όροι

Πριν περιγράψουμε τους αλγορίθμους λειτουργίας του Cluster Autoscaler, είναι αρκετά σημαντικό να κατανοήσουμε ορισμένους θεμελιώδεις μηχανισμούς και την ορολογία που χρησιμοποιεί.

3.4.1.1 Ομάδα Κόμβων

Ο Cluster Autoscaler χρησιμοποιεί την αφηρημένη έννοια "*ομάδα κόμβων*". Μια ομάδα κόμβων δεν είναι ένα πραγματικό αντικείμενο του Kubernetes, αλλά μια αφηρημένη έννοια για ένα υποσύνολο κόμβων σε μια συστοιχία. Οι κόμβοι που ανήκουν στην ίδια ομάδα κόμβων αναμένεται να έχουν τους ίδιους πόρους (CPU, μνήμη) και να μοιράζονται αρκετές κοινές ιδιότητες, όπως ετικέτες και taints. Ωστόσο, μπορούν ακόμη να διαφοροποιούνται σε ορισμένες λεπτομέρειες, π.χ. μπορεί να ανήκουν σε διαφορετικές ζώνες διαθεσιμότητας.

Κάθε ομάδα κόμβων έχει τις ακόλουθες σημαντικές ιδιότητες:

- `minSize`: το ελάχιστο μέγεθος της ομάδας κόμβων.
- `maxSize`: το μέγιστο μέγεθος της ομάδας κόμβων.
- `targetSize`: το επιθυμητό μέγεθος της ομάδας κόμβων.

3.4.1.2 Η Διεπαφή `CloudProvier`

Ο Cluster Autoscaler λειτουργεί με διάφορους παρόχους cloud, π.χ. με το GCE, AWS, κ.λπ. Για να το επιτύχει αυτό, ορίζει δύο σημαντικές διεπαφές που κάθε πάροχος νέφους που επιθυμεί την ενσωμάτωση των υπηρεσιών του με τον Cluster Autoscaler πρέπει να υλοποιήσει:

- Η διεπαφή `CloudProvider`: περιέχει πληροφορίες διαμόρφωσης και λειτουργίες για την αλληλεπίδραση με τον πάροχο του νέφους.
- Η διεπαφή `NodeGroup`: περιέχει πληροφορίες διαμόρφωσης και λειτουργίες για τον έλεγχο μίας ομάδας κόμβων.

Η διεπαφή `NodeGroup` βασίζεται στην έννοια *ομάδα κόμβων*, την οποία εξηγήσαμε προηγουμένως. Κάθε πάροχος νέφους μπορεί να επιλέξει τη δική του ερμηνεία για το τι είναι ομάδα κόμβων στην υπηρεσία του, αρκεί να συμμορφώνεται με τον ορισμό της έννοιας.

Για παράδειγμα, στην περίπτωση του AWS EKS, η υλοποίηση του `NodeGroup` αντιστοιχίζει την κάθε ομάδα κόμβων σε μια ομάδα αυτόματης κλιμάκωσης (ASG) του AWS. Ο Cluster Autoscaler δεν γνωρίζει την υποκείμενη υλοποίηση κάθε παρόχου νέφους.

3.4.1.3 Η Διεπαφή `ClusterSnapshot`

Ο Cluster Autoscaler εκτελεί προσομοιώσεις στη συστοιχία για τη λήψη αποφάσεων. Για να το επιτύχει αυτό, λαμβάνει ένα στιγμιότυπο της τρέχουσας συστοιχίας χρησιμοποιώντας τη διεπαφή `ClusterSnapshot`, προσθέτει ή αφαιρεί κόμβους στο στιγμιότυπο και προσομοιώνει τις αποφάσεις χρονοδρομολόγησης στο τροποποιημένο στιγμιότυπο. Ένα στιγμιότυπο συστοιχίας περιέχει μια παγωμένη εικόνα των κόμβων της συστοιχίας ως καθώς και των Pods που εκτελούνται σε κάθε κόμβο της. Σημειώνουμε ότι τα PVCs και τα PVs της συστοιχίας δεν περιέχονται στη συστοιχία, αντ' αυτού το VolumeBinding plugin τα ζητά από τον API Server και ελέγχει αν ένα Pod μπορεί να τοποθετηθεί σε έναν κόμβο βάσει τον τόμων του.

3.4.1.4 Η Διεπαφή `PredicateChecker`

Ο Cluster Autoscaler ορίζει τη διεπαφή `PredicateChecker`, η οποία προδιαγράφει μεθόδους που ελέγχουν αν όλα τα απαιτούμενα predicates / κριτήρια επιτυγχάνουν για

δεδομένο Pod και κόμβο. Ένα predicate είναι ισοδύναμο με ένα πρόσθετο `Filter` και χρησιμοποιείται για το φιλτράρισμα των κόμβων που δεν μπορούν να εκτελέσουν ένα Pod. Ουσιαστικά, με τον PredicateChecker ο Cluster Autoscaler ελέγχει αν ένα Pod μπορεί να τρέξει σε έναν κόμβο. Η υλοποίηση `SchedulerBasedPredicateChecker` της διεπαφής, κάνει χρήση του κώδικα του Scheduler για να ελέγξει κατά τις προσομοιώσεις του Cluster Autoscaler αν ένα Pod μπορεί να τρέξει σε έναν κόμβο.

3.4.1.5 Οι Διεπαφές `Esitmator` & `Strategy`

Ο Autoscaler ορίζει δύο διεπαφές που χρησιμοποιούνται στη διαδικασία κλιμάκωσης:

- `Estimator`: διεπαφή για τον υπολογισμό του αριθμού των κόμβων ενός δεδομένου τύπου που απαιτούνται για τον προγραμματισμό
- `Strategy`: διεπαφή για τη λήψη απόφασης σχετικά με την καλύτερη επιλογή κλιμάκωσης.

Ο Estimator που χρησιμοποιείται επί του παρόντος από τον Autoscaler είναι ο `BinPackingNodeEstimator` και υλοποιεί τον First Fit Decreasing bin packing αλγόριθμο.

3.4.1.6 Πρότυπος Κόμβος

Ο Cluster Autoscaler για να εκτελέσει το προσομοιώσεις του, κατασκευάζει έναν πρότυπο κόμβο για κάθε ομάδα κόμβων. Ένας πρότυπος κόμβος, όπως υποδηλώνει το όνομα, είναι μια αναπαράσταση του πώς θα έμοιαζε ένας νέος κόμβος της ομάδας κόμβων μόλις μετά την προσθήκη του στη συστοιχία.

Ο Cluster Autoscaler προσπαθεί να δημιουργήσει έναν πρότυπο κόμβο μιας ομάδας κόμβων ως εξής:

1. Αρχικά, αναζητά έναν Ready και χρονοδρομολογήσιμο κόμβο της ομάδας κόμβων στη συστοιχία.
2. Εάν το προηγούμενο βήμα απέτυχε, αναζητά ένα πρότυπο στη μνήμη cache του.
3. Εάν το προηγούμενο βήμα απέτυχε, καλεί το μεταγλωττισμένο πρόσθετο του παρόχου νέφους για τη δημιουργία ενός προτύπου για τη συγκεκριμένη ομάδα κόμβων.

4. Εάν το προηγούμενο βήμα απέτυχε, αναζητά τυχόν Unready ή μη χρονοδρομολογήσιμους κόμβους της ομάδας κόμβων στη συστοιχία και τον χρησιμοποιεί για τη δημιουργία του προτύπου.

Οι κόμβοι του κατασκευασμένου προτύπου *απολυμαίνονται*: η *απολύμανση* ενός κόμβου είναι ένας μηχανισμός που αφαιρεί άσχετες ή ανεπιθύμητες λεπτομέρειες από το κατασκευασμένο πρότυπο κόμβου, όπως, το όνομα του κόμβου, συγκεκριμένες ετικέτες, κ.λπ.

3.4.1.7 Χρησιμοποίηση Κόμβου

Όσον αφορά την κλιμάκωση προς τα κάτω, ο Autoscaler ενεργεί με βάση τη *χρησιμοποίηση* ενός κόμβου, η οποία υπολογίζεται χρησιμοποιώντας τα αιτήματα πόρων των Pods που εκτελούνται στον κόμβο και όχι σε πραγματικές (ζωντανές) μετρήσεις χρήσης πόρων.

Κάθε κόμβος της συστοιχίας μπορεί να διαθέτει πολλαπλούς πόρους, όπως CPU, μνήμη κ.λπ. Ο Cluster Autoscaler υπολογίζει τη χρησιμοποίηση κάθε κόμβου στη συστοιχία. Για ένα συγκεκριμένο πόρο, η χρησιμοποίηση των κόμβων είναι το άθροισμα των αιτήσεων των Pods που εκτελούνται στον κόμβο για τον πόρο δια την διαθέσιμη ποσότητα του πόρου στον κόμβο. Η χρησιμοποίηση είναι ένας αριθμός float, με εύρος από 0 έως 1, όπου το 1 υποδηλώνει πλήρη χρησιμοποίηση και το 0 καθόλου χρησιμοποίηση.

3.4.2 Ο Κύριος Βρόχος

Ο Cluster Autoscaler τρέχει διαρκώς τον *κύριο βρόχο*, που εκτελεί 2 βασικές λειτουργίες στη συστοιχία:

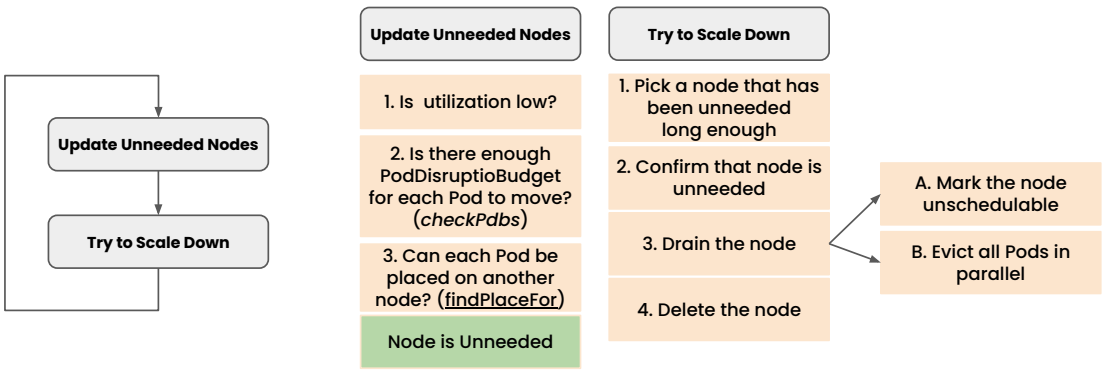
- *Κλιμάκωση προς τα πάνω*: προσθήκη νέων κόμβων στη συστοιχία για να βοηθηθούν τα Pods που δεν μπορούν να χρονοδρομολογηθούν.
- *Κλιμάκωση προς τα κάτω*: αφαίρεση μη απαραίτητων κόμβων από τη συστοιχία.

3.4.3 Κλιμάκωση Προς τα Κάτω

Σε κάθε επανάληψη του κύριου βρόχου, εάν δεν επιχειρήθηκε κλιμάκωση προς τα πάνω, ο Cluster Autoscaler εκτελεί τη διαδικασία αξιολόγησης της κλιμάκωσης προς τα κάτω.

Η αξιολόγηση της κλιμάκωσης προς τα κάτω αποτελείται από δύο μέρη:

1. *Ενημέρωση περιττών κόμβων*: υπολογίζει ποιοι κόμβοι της συστοιχίας είναι περιττοί και για πόσο χρονικό διάστημα.
2. *Προσπάθεια κλιμάκωσης προς τα κάτω*: προσπαθεί να μειώσει την κλίμακα της συστοιχίας προς τα κάτω αφαιρώντας περιττούς κόμβους.



Σχήμα 3.1: *Cluster Autoscaler: Διαδικασία κλιμάκωσης προς τα κάτω*

3.4.3.1 Διαδικασία Ενημέρωσης Περιττών Κόμβων

Η διαδικασία *ενημέρωσης περιττών κόμβων* υπολογίζει ποιοι κόμβοι της συστοιχίας είναι περιττοί και ενημερώνει την εσωτερική κατάσταση του Autoscaler με τη χρονική διάρκεια για την οποία έχουν παραμείνει περιττοί.

Ένας κόμβος θεωρείται *περιττός* εάν πληρούνται όλα τα ακόλουθα κριτήρια:

- Η μετρική της χρησιμοποίησης των πόρων του κόμβου είναι κάτω από ένα συγκεκριμένο κατώφλι. Σε αυτή την περίπτωση ο κόμβος αναφέρεται ως *"υποχρησιμοποιούμενος"*.
- Τα Pods που εκτελούνται στον κόμβο μπορούν να εκδιωχθούν, δηλαδή η εκδίωξή τους δεν περιορίζεται από κάποιο PodDisruptionBudget.

- Τα Pods που εκτελούνται στον κόμβο μπορούν να μετακινηθούν σε διαφορετικό κόμβο της συστοιχίας.

Εάν ένας κόμβος δεν ικανοποιεί τα παραπάνω κριτήρια, αποκαλείται *αναφαίρετος*, αφού δεν μπορεί να αφαιρεθεί από τη συστοιχία.

Για να καθορίσει αν ένας υποχρησιμοποιούμενος κόμβος είναι περιττός, ο Autoscaler εκτελεί τα εξής βήματα:

1. Υπολογίζει τα Pods που πρέπει να μετακινηθούν σε περίπτωση που ο κόμβος αφαιρεθεί.
2. Ελέγχει εάν υπάρχουν PodDisruptionBudgets που εμποδίζουν την έξωση οποιουδήποτε Pod. Εάν ναι, ο κόμβος δεν μπορεί να αφαιρεθεί.
3. Καλεί τη μέθοδο `FindPlaceFor()` για να βρει θέση για τα Pods σε ένα διαφορετικό κόμβο. Η `FindPlaceFor()` χρησιμοποιεί την διεπαφή `SchedulerBasedPredicateChecker` για να καθορίσει αν το Pod μπορεί να είναι τοποθετηθεί σε οποιονδήποτε άλλο κόμβο. Ελέγχει αν το Pod μπορεί να χωρέσει σε έναν κόμβο λόγω οποιωνδήποτε άλλων περιορισμών χρονοδρομολόγησης (CPU, μνήμη), καθώς και αν ο οι τόμοι του Pod μπορούν να προσπελαστούν από τον κόμβο.

3.4.3.2 Διαδικασία Προσπάθειας Κλιμάκωσης Προς Τα Κάτω

Εάν ένας *Ready* κόμβος παραμείνει περιττός για διάστημα μεγαλύτερο από το `scaleDownUnneededTime` ή ένας *Unready* κόμβος παραμείνει περιττός για διάστημα μεγαλύτερο από το `scaleDownUnreadyTime` (δείτε τον Πίνακα 3.1 στο αγγλικό μέρος της εργασίας για τους ορισμούς αυτών των συμβολικών ονομάτων), ο Cluster Autoscaler θα θεωρήσει τον κόμβο ως υποψήφιο για διαγραφή.

Ο Cluster Autoscaler κάνει διάκριση μεταξύ μη κενών και κενών κόμβων:

- *Κενοί κόμβοι*: κόμβοι που εκτελούν *μονάχα* DaemonSet Pods. Ο Autoscaler αφαιρεί τους κενούς κόμβους μαζικά.
- *Non-empty nodes*: κόμβοι που δεν εκτελούν μόνο DaemonSet Pods. Ο Autoscaler αφαιρεί τους μη κενούς κόμβους έναν προς έναν προκειμένου να μη δημιουργηθούν μη χρονοδρομολογήσιμα Pods.

Διαγραφή κόμβου Η διαγραφή κόμβου εκτελείται ως εξής:

1. Βάζει ένα taint `ToBeDeletedByClusterAutoscaler:NoSchedule` στον κόμβο, ουσιαστικά χαρακτηρίζοντας τον κόμβο ως unschedulable.
2. Ξεκινά τη διαδικασία drain του κόμβου, εκδιώκοντας παράλληλα όλα τα Pods του κόμβου. Εάν κάποιο Pod δεν μπορεί να εκδιωχθεί λόγω κάποιου PDB, επαναλαμβάνει την προσπάθεια έως ότου παρέλθει το χρονικό διάστημα `MaxPodEvictionTimeout`.
3. Όταν όλα τα Pods εκδιωχθούν επιτυχώς, ζητά από τον πάροχο νέφους να διαγράψει τον κόμβο.

3.4.4 Κλιμάκωση Προς τα Πάνω

Εάν η συστοιχία έχει Pods που δεν μπορούν να χρονοδρομολογηθούν, ο Cluster Autoscaler θα προσπαθήσει να τα βοηθήσει κλιμακώνοντας προς τα πάνω τη συστοιχία, δηλαδή προσθέτοντας νέους κόμβους. Η κλιμάκωση προς τα πάνω μίας συστοιχίας, επιτυγχάνεται αυξάνοντας το επιθυμητό μέγεθος (`targetSize`) κάποιων ομάδων κόμβων. Εάν έχουν διαμορφωθεί πολλαπλές ομάδες κόμβων στη συστοιχία, τότε πρέπει να αποφασίσει τα εξής:

- Ποια ομάδα κόμβων μπορεί να τρέξει το μη χρονοδρομολογήσιμο Pod.
- Πόσοι κόμβοι της κάθε ομάδας κόμβων απαιτούνται.
- Στην περίπτωση που η κλιμάκωση προς τα πάνω διαφορετικών ομάδων κόμβων είναι εφικτή, ποια ομάδα κόμβων θα κλιμακωθεί.

Όταν το επιθυμητό μέγεθος μιας ομάδας κόμβων αυξάνεται, ο πάροχος του νέφους θα δημιουργήσει νέους κόμβους, ο οποίοι θα ενταχθούν στη συστοιχία Kubernetes και ο χρονοδρομολογητής θα αναθέσει σταδιακά τα μέχρι πρότινος μη χρονοδρομολογήσιμα Pods στους νέους κόμβους.

Επιλογές κλιμάκωσης Για να αποφασίσει αν η αύξηση της κλίμακας μιας ομάδας κόμβων θα βοηθήσει τα μη χρονοδρομολογήσιμα Pods, ο Cluster Autoscaler εκτελεί τα εξής βήματα:

1. Λαμβάνει ένα στιγμιότυπο της συστοιχίας

2. Προσθέτει έναν πρότυπο κόμβο της ομάδας κόμβων στο στιγμιότυπο
3. Προσομοιώνει, χρησιμοποιώντας τον `SchedulerBasedPredicateChecker`, αν το μη χρονοδρομολογήσιμο Pod μπορεί να χρονοδρομολογηθεί στο τροποποιημένο στιγμιότυπο της συστοιχίας.
4. Εάν η προσομοίωση δείξει ότι το Pod μπορεί να χρονοδρομολογηθεί, ο Cluster Autoscaler θα χρησιμοποιήσει τον `BinPackingNodeEstimator` για να υπολογίσει πόσοι κόμβοι της συγκεκριμένης ομάδας κόμβων απαιτούνται.

Η δυνατότητα για κλιμάκωση μιας συγκεκριμένης ομάδας κόμβων μαζί με τον αριθμό των κόμβων της ομάδας που απαιτούνται, αναφέρεται ως "*επιλογή κλιμάκωσης*".

Στρατηγική κλιμάκωσης Εάν υπάρχουν πολλαπλές επιλογές κλιμάκωσης, ο Cluster Autoscaler θα αποφασίζει ποια είναι η καλύτερη χρησιμοποιώντας τη διεπαφή `Strategy`. Υπάρχουν διάφορες στρατηγικές και ο τελικός χρήστης μπορεί να ρυθμίσει τον Autoscaler ώστε να να χρησιμοποιεί την επιθυμητή, π.χ. την επιλογή με το μικρότερο κόστος, την τυχαία στρατηγική κ.λπ.

3.4.5 Ελλείψεις & Προτεινόμενες Επεκτάσεις

3.4.5.1 Κλιμάκωσης Προς τα Κάτω: Οι Μόνιμοι Τόμοι του Rok Μετακινούνται

Κατά την αξιολόγηση της αφαίρεσης ενός κόμβου, ο Autoscaler προσπαθεί να βρει θέση για τα Pods που εκτελούνται στον κόμβο σε άλλους κόμβους της συστοιχίας. Για τον σκοπό αυτό, καλεί τη μέθοδο `FindPlaceFor()`, η οποία με τη σειρά της αξιοποιεί τις μεθόδους της διεπαφής `PredicateChecker` για να διαπιστώσει αν ένα Pod ταιριάζει σε έναν κόμβο. Η υλοποίηση της διεπαφής `SchedulerBasedPredicateChecker`, εκτελεί τη μέθοδο `Filter()` του πρόσθετου VolumeBinding για να ελέγξει αν οι τόμοι του Pod μπορούν να προσπελαστούν από άλλον κόμβο.

Τα PV της κλάσης αποθήκευσης του Rok έχουν node affinity που ταιριάζει μόνο με τον κόμβο στον οποίο ο τόμος βρίσκεται. Δεδομένου ότι το node affinity των τόμων δεν ταιριάζει με κανέναν άλλο κόμβο στη συστοιχία, ο Autoscaler θα θεωρήσει ότι το Pod και ο τόμος του δεν μπορούν να μετακινηθούν σε διαφορετικό κόμβο, χαρακτηρίζοντας έτσι το τρέχοντα κόμβο ως αναφαίρετο. Ο Autoscaler, και ο `SchedulerBasedPredicateChecker` ειδικότερα, δεν γνωρίζουν ότι οι τόμοι της κλάσης Rok διαθέτουν έναν

μηχανισμό που θα λάβει ένα στιγμιότυπο και θα μπορεί να τους ανακτήσει σε διαφορετικό κόμβο (διαδικασία unpinning & pinning, βλ. ενότητα 3.2.1).

Προτείνουμε την επέκταση του Autoscaler ώστε να προσομοιώνει τους τόμους Rok ως unpinned (σαν να μην έχουν node affinity) κατά την αξιολόγηση μιας κλιμάκωσης προς τα κάτω (τότε, και μόνο τότε - σε άλλες περιπτώσεις οι τόμοι διατηρούν το node affinity τους). Με αυτή την επέκταση, ο Autoscaler θα αντιλαμβάνεται ότι οι τόμοι Rok μπορούν να μετακινηθούν οπουδήποτε στη συστοιχία και ο κόμβος μπορεί να αφαιρεθεί με ασφάλεια.

3.4.5.2 Κλιμάκωση Προς Τα Κάτω: Συνεργασία με τον Μηχανισμό Rok CSI Guard

Στο πλαίσιο του μηχανισμού προστασίας των τόμων του Rok (βλ. ενότητα 3.2.2), δημιουργούμε ένα αντικείμενο `Deployment` για κάθε κόμβο της συστοιχίας, το οποίο δημιουργεί ένα Pod ανά κόμβο (Rok CSI Guard) με αυστηρό node affinity που ταιριάζει μόνο με τον κόμβο που προστατεύει. Ο Autoscaler θα προσπαθήσει να βρει κάποιον κόμβο για να μετακινήσει το Pod. Ωστόσο, επειδή το Pod έχει node affinity ταιριάζει μόνο στον τρέχοντα κόμβο δεν μπορεί να μετακινηθεί αλλού, και ως εκ τούτου ο Autoscaler θα χαρακτηρίσει τον τρέχοντα κόμβο ως αναφαίρετο. Φυσικά, το Pod θα αφαιρεθεί εντελώς από τον Rok Operator μόλις ο κόμβος αφαιρεθεί, αλλά ο Autoscaler δεν το γνωρίζει αυτό.

Επιπλέον, ο Autoscaler θα ελέγξει το PDB του Guard Pod. Το PDB του Rok CSI Guard Pod έχει ρυθμιστεί ώστε να προκαλεί την αποτυχία τυχόν εκδιώξεων. Ο Autoscaler θα το παρατηρήσει αυτό, θα θεωρήσει ότι το Guard Pod δεν θα μπορέσει να εκδιωχθεί με ασφάλεια, και θα χαρακτηρίσει τον κόμβο αναφαίρετο. Ο Autoscaler δεν γνωρίζει ότι το PDB διαχειρίζεται από τον Rok Operator και ότι θα αφαιρεθεί μόλις ξεκινήσει η κλιμάκωση προς τα κάτω του κόμβου και οι τοπικοί τόμοι του Rok στον κόμβο γίνουν unpinned.

Προτείνουμε την επέκταση του Autoscaler έτσι ώστε να μην προσπαθεί να βρει θέση για την Guard Pod. Επιπλέον, επεκτείνουμε την Autoscaler ώστε να αγνοεί το PDB του Guard Pod όταν κατά την αξιολόγηση της δυνατότητας αφαίρεσης ενός κόμβου. Παρόλα αυτά, ο Autoscaler θα γνωρίζει ότι ο Pod υπάρχει και θα αρχίσει να εκδιώκει (evict) το Guard Pod όταν κάνει drain τον κόμβο.

Για να γίνουν τα πράγματα πιο κατανοητά, ακολουθεί η διαδικασία που θα πραγματοποιείται με το νέο σχεδιασμό:

1. Ο Autoscaler αξιολογεί έναν κόμβο για αφαίρεση:
   (α′) Ελέγχει αν το PDB επιτρέπει την εκδίωξη των Pods που εκτελούνται στον κόμβο, αλλά αγνοεί το PDB του Guard Pod.
   (β′) Προσπαθεί να βρει θέση για κάθε Pod σε διαφορετικό κόμβο, αλλά αγνοεί το Guard Pod.
2. Ο Autoscaler αποφασίζει να αφαιρέσει τον κόμβο.
3. Ο Autoscaler προσθέτει το taint διαγραφής στον κόμβο, χαρακτηρίζοντάς τον ουσιαστικά ως ως μη χρονοδρομολογήσιμο.
4. Ο Autoscaler αποστέλλει αιτήματα εκδίωξης για κάθε Pod στον API Server.
5. Η εκδίωξη όλων των Pods –εκτός από το Guard Pod– είναι επιτυχής.
6. Ο Autoscaler συνεχίζει να προσπαθεί εκ νέου να εκδιώξει το Guard Pod, αλλά ο API Server απαντά ότι η εκδίωξη δεν επιτρέπεται λόγω του ρυθμισμένου PDB.
7. Ο ελεγκτής CSI του Rok παρατηρεί ότι ο κόμβος είναι μη χρονοδρομολογήσιμος και ότι κανένα Pod δεν προσαρτά τους τόμους, οπότε αρχίζει να τους τους κάνει unpin.
8. Ο ελεγκτής CSI του Rok ολοκληρώνει το unpin των τόμων και αφαιρεί το node affinity των αντίστοιχων PV.
9. Ο Rok Operator αφαιρεί το PodDisruptionBudget του Guard Pod.
10. Το αίτημα του Autoscaler για την έξωση του Guard Pod τελικά πετυχαίνει (αφού το PDB αφαιρέθηκε).
11. Ο Autoscaler ζητά από τον πάροχο νέφους να διαγράψει τον κόμβο.
12. Ο Rok Operator αφαιρεί το Rok CSI Guard Deployment του κόμβου που μόλις αφαιρέθηκε.

3.4.5.3 Κλιμάκωση Προς τα Κάτω: Διαθέσιμος Αποθηκευτικός Χώρος

Ο Autoscaler θα πρέπει ελέγχει εάν οι τόμοι της κλάσης αποθήκευσης Rok ενός Pod μπορούν να χωρέσουν σε έναν κόμβο ως προς τον απαιτούμενο αποθηκευτικό χώρο κατά την αξιολόγηση της κλιμάκωσης προς τα κάτω. Όπως έχουμε εξηγήσει, ο Autoscaler χρησιμοποιεί τη διεπαφή `SchedulerBasedPredicateChecker` προκειμένου να ελέγξει

αν ένα Pod χωράει σε έναν κόμβο, η οποία –μεταξύ άλλων– καλεί τη μέθοδο `Filter()` του πρόσθετου VolumeBinding.

Προτείνουμε την επέκταση της μεθόδου `Filter` του VolumeBinding προσθέτου του SchedulerBasedPredicateChecker , έτσι ώστε όταν καλείται να ελέγξει αν ένα Pod μπορεί να μετακινηθεί σε διαφορετικό κόμβο να λαμβάνει υπόψη αν υπάρχει αρκετός διαθέσιμος αποθηκευτικός χώρος στον κόμβο για να μεταφερθούν εκεί οι τόμοι του Pod.

Επιπλέον, δεδομένου ότι η δημιουργία στιγμιότυπων και η μετακίνηση ενός τόμου είναι μια διαδικασία που κοστίζει σε χρόνο, θα θέλαμε ο Autoscaler να μην αφαιρεί κόμβους που έχουν υψηλή χρησιμοποίηση της αποθήκευσης, κατά παρόμοιο τρόπο με το πως χειρίζεται τη χρησιμοποίηση της CPU και τη μνήμης. Για να το επιτύχουμε αυτό, προτείνουμε την επέκταση του Autoscaler με μία νέα μετρική, που ονομάζεται "*χρησιμοποίηση αποθηκευτικού χώρου*", και ορίζεται ως ο λόγος του χρησιμοποιούμενου αποθηκευτικού χώρου προς τη μέγιστη αποθηκευτική ικανότητα του κόμβου. Ο Autoscaler θα συγκρίνει αυτή τη μετρική με ένα κατώφλι, το οποίο μπορεί να διαμορφωθεί από τον διαχειριστή της συστοιχίας μέσω μίας σημαίας στον Autoscaler. Εάν η χρήση του αποθηκευτικού χώρου είναι πάνω από το κατώφλι, ο κόμβος θα θεωρείται αναφαίρετος.

3.4.5.4 Κλιμάκωση Προς τα Κάτω: Οι Unready Κόμβοι Να Μην Αφαιρούνται

Ένας κόμβος με κατάσταση `Ready` μπορεί να γίνει `Unready` (ή `NotReady`) εάν προκύψει κάποιο πρόβλημα. Οι συνήθεις λόγοι περιλαμβάνουν την έλλειψη πόρων στον κόμβο, ένα πρόβλημα με το kubelet, ένα σφάλμα που σχετίζεται με τον kube-proxy ή ένα πρόβλημα δικτύωσης.

Ο Autoscaler αφαιρεί κάθε περιττό κόμβο σε κατάσταση Unready μόλις παρέλθει το `scaleDownUnreadyTime` διάστημα. Αυτό είναι πρόβλημα, δεδομένου ότι οι τοπικοί τόμοι που βρίσκονται στον κόμβο θα χαθούν μόνιμα. Ο Autoscaler δεν πρέπει να αφαιρεί Unready κόμβους. Αντιθέτως, οι κόμβοι πρέπει να παραμένουν στη συστοιχία, έτσι ώστε ένας διαχειριστής να εκτελέσει κατάλληλες ενέργειες για την ανάκαμψη από της κατάσταση Unready.

Προτείνουμε την επέκταση του Autoscaler με μια σημαία για τη ρητή απενεργοποίηση της μείωση της κλίμακας για τους κόμβους που βρίσκονται σε κατάσταση `Unready`, και

να θεωρεί αυτούς τους κόμβους ως αναφαίρετους όταν απενεργοποιείται η κλιμάκωση προς τα κάτω για τους Unready κόμβους.

3.4.5.5 Κλιμάκωση Προς τα Πάνω: Χωρητικότητα Αποθήκευσης

Ο Autoscaler δεν γνωρίζει πόσος τοπικός αποθηκευτικός χώρος θα είναι διαθέσιμος όταν ένας νέος κόμβος προστεθεί στη συστοιχία. Ο πρότυπος κόμβος που δημιουργείται από έναν ζωντανό κόμβο περιέχει πληροφορίες μονάχα για τον εναπομείναντα ελεύθερο αποθηκευτικό χώρο (ο οδηγός αποθήκευσης αναφέρει την τιμή αυτή ως annotation στο Node αντικείμενο) . Χρειαζόμαστε έναν μηχανισμό για να γνωρίζουμε πόσο ελεύθερο αποθηκευτικό χώρο θα έχει ένας νέος κόμβος μιας ομάδας κόμβων μόλις προστεθεί, τον οποίο ο Autoscaler θα το λαμβάνει υπόψη στις προσομοιώσεις του.

Αναφορά μεγίστης χωρητικότητας Υποθέτοντας ότι όλοι οι κόμβοι σε μια ομάδα κόμβων έχουν την ίδια διαμόρφωση δίσκου και την ίδια μέγιστη χωρητικότητα αποθήκευσης, μπορούμε να χρησιμοποιήσουμε το πρόγραμμα οδήγησης αποθήκευσης για να αναφέρουμε πάνω στα αντικείμενα `Node` ποια θα είναι η χωρητικότητα αποθήκευσης ενός νέου κόμβου. Προτείνουμε την επέκταση των στοιχείων Rok CSI Node του οδηγού αποθήκευσης που τρέχουν σε κάθε κόμβο ώστε να αναφέρουν τη μέγιστη χωρητικότητα ενός κόμβου ως ετικέτα στο αντικείμενο `Node`. Αυτή η ετικέτα θα αναφέρεται ως η "ετικέτα μέγιστης χωρητικότητας" [4].

Επιλέγουμε να χρησιμοποιήσουμε μια ετικέτα αντί για ένα σχολιασμό, επειδή διάφοροι πάροχοι νέφους δίνουν στους διαχειριστές των συστοιχιών την επιλογή να περνούν ετικέτες στα πρότυπα των ομάδων κόμβων, όταν αυτά κατασκευάζονται από το πρόσθετο του παρόχου νέφους.

Σε αυτό το σημείο, ας διακρίνουμε τις δύο αναφερόμενες ποσότητες:

- *annotation χωρητικότητας*: ο εναπομείνας ελεύθερος αποθηκευτικός χώρος ενός live κόμβου. Ο οδηγός αποθήκευσης αναφέρει την τιμή αυτή, και ο χρονοδρομολογητής τη λαμβάνει υπόψη κατά τη χρονοδρομολόγηση.
- *ετικέτα μέγιστης χωρητικότητας*: η μέγιστη χωρητικότητα αποθήκευσης του ζωντανού κόμβου. δηλαδή, η συνολική χωρητικότητα αποθήκευσης. Πρόκειται για

[4] Η ετικέτα *μεγίστης χωρητικότητας*: rok.arrikto.com/max-instance-capacity

μια χρονικά αμετάβλητη ποσότητα που εξαρτάται από τον τύπο του κόμβου.

Προκειμένου ο Autoscaler να προσομοιώνει την χρονοδρομολόγηση λαμβάνοντας υπόψη τη χωρητικότητα αποθήκευσης, θα εισάγουμε στον SchedulerBasedPredicateChecker το επεκταμένο VolumeBinding (βλ. ενότητα 3.3.2.3).

Το επεκταμένο VolumeBinding πρόσθετο αναζητά τη χωρητικότητα ενός κόμβου προτύπου στο annotation χωρητικότητας και όχι στην ετικέτα. Δεδομένου ότι το annotation ενός πρότυπου κόμβου προκύπτει από αυτό ενός live κόμβου, αντιπροσωπεύει τον τρέχοντα ελεύθερο αποθηκευτικό χώρο στον ζωντανό κόμβο, αντί της μέγιστης χωρητικότητας αποθήκευσης. Πρέπει να καθαρίσουμε την τιμή και να την ορίσουμε στην πραγματική μέγιστη χωρητικότητα. Για να γίνει αυτό, προτείνουμε την επέκταση του μηχανισμού απολύμανσης του Autoscaler ώστε να αντιγράφει την τιμή της ετικέτας μέγιστης χωρητικότητας στο annotation χωρητικότητας. Με αυτόν τον τρόπο, το annotation χωρητικότητας του κόμβου προτύπου θα υποδεικνύει τη μέγιστη χωρητικότητα αποθήκευσης ενός νέου κόμβου της συγκεκριμένης ομάδας κόμβων.

Εάν η ετικέτα μέγιστης χωρητικότητας δεν υπάρχει, ο μηχανισμός απολύμανσης θέτει το annotation χωρητικότητας σε μια απείρως μεγάλη τιμή. Αυτή η σχεδιαστική επιλογή προσφέρει τα ακόλουθα πλεονεκτήματα:

1. Ο Autoscaler θα αντιμετωπίζει τον κόμβο σαν να έχει άπειρη χωρητικότητα αποθήκευσης και θα προσθέσει έναν κόμβο της ομάδας κόμβων. Εάν η απόφαση ήταν λανθασμένη (*false scale-up*), δηλαδή ο κόμβος δεν έχει αρκετή χωρητικότητα αποθήκευσης για το Pod που προκάλεσε την προσθήκη του κόμβου, ο χρονοδρομολογητής της συστοιχίας δεν θα αναθέσει το Pod στον νέο κόμβο, ο κόμβος θα παραμείνει περιττός και ο Autoscaler θα τον αφαιρέσει μετά από κάποιο χρονικό διάστημα. Σταδιακά λοιπόν, θα διορθωθεί το σφάλμα.

2. Η λανθασμένη προσθήκη του κόμβου, θα επιτρέψει στον Autoscaler να μάθει την πραγματική μέγιστη χωρητικότητα του κόμβου. Ο οδηγός αποθήκευσης Rok CSI θα λάβει την ευκαιρία να τρέξει στον κόμβο και ο Autoscaler θα δημιουργήσει έναν πρότυπο κόμβο από το ζωντανό κόμβο, ο οποίος θα περιέχει ακριβείς πληροφορίες για τη μέγιστη διαθέσιμη χωρητικότητα που αναφέρει ο οδηγός Rok CSI.

Αναμονή για το Rok CSI Εάν ένα Pod προκαλέσει κλιμάκωση προς τα πάνω λόγω του αποθηκευτικού χώρου που ζητάει, ενδέχεται να χρειαστεί ένα εύλογο χρονικό διάστημα από την προσθήκη του νέου στον κόμβο. Όσο το πρόγραμμα οδήγησης Rok CSI δεν εκτελείται στον κόμβο, το annotation χωρητικότητας δεν θα υπάρχει στο αντικείμενο `Node`. Ο χρονοδρομολογητής της συστοιχίας δεν θα αναθέτει το Pod στον κόμβο, καθώς η απουσία του annotation χωρητικότητας υποδηλώνει έλλειψη χωρητικότητας. Το Autoscaler θα λάβει ένα στιγμιότυπο των κόμβων, θα εκτελέσει την προσομοίωση της χρονοδρομολόγησης και θα αποφασίσει ότι το Pod δεν χωράει στους υπάρχοντες κόμβους (καθώς ο ζωντανός –πλέον– κόμβος που μόλις προσέθεσε στις προσομοιώσεις του φαίνεται να έχει μηδενική χωρητικότητα), οπότε και θα εκτελέσει εκ νέου κλιμάκωση προς τα πάνω για να βοηθήσει το μη χρονοδρομολογήσιμο Pod.

Για να επιλυθεί αυτό το πρόβλημα, ο Autoscaler πρέπει να περιμένει να τρέξει το πρόγραμμα οδήγησης Rok CSI στον νέο κόμβο- όσο το πρόγραμμα οδήγησης δεν εκτελείται, (δηλαδή, δεν υπάρχει κόμβος που να έχει την capacity label set), η Autoscaler αντικαθιστά τον κόμβο με ένα αντίγραφο *Unready*. Οι Unready κόμβοι αντιμετωπίζονται ως επερχόμενοι κόμβοι: αντικαθίστανται από πρότυπο της ομάδας κόμβων στην οποία ανήκουν στις προσομοιώσεις του Autoscaler. Το πρότυπο της ομάδας κόμβου θα διαθέτει το annotation χωρητικότητας και η προσομοίωση του Autoscaler θα χρονοδρομολογήσει το Pod που χρειαζόταν βοήθεια σε αυτόν. Αυτός ο μηχανισμός αποτρέπει οποιαδήποτε περαιτέρω αύξηση της κλίμακας έως ότου ο οδηγός αρχίσει να εκτελείται στον κόμβο.

Εάν ο οδηγός Rok CSI Driver δεν εκτελεστεί στον κόμβο μέσα σε 15 λεπτά, ο Cluster Autoscaler θα σταματήσει να αντικαθιστά τον κόμβο με το μη έτοιμο αντίγραφο. Η διάρκεια των 15 λεπτών είναι ένα εύλογο χρονικό διάστημα για το πρόγραμμα οδήγησης CSI να αρχίσει να εκτελείται - εάν δεν το κάνει, αποτελεί ένδειξη ότι υπάρχει πρόβλημα με τον κόμβο.

4 Υλοποίηση

Σε αυτό το κεφάλαιο περιγράφουμε συνοπτικά την υλοποίηση των προτεινόμενων αλλαγών σχεδιασμού και τις τεχνολογίες που χρησιμοποιήθηκαν.

4.1 Λογισμικό

Ο προτεινόμενος σχεδιασμός περιλαμβάνει πολλά διαφορετικά μέρη που έπρεπε να επεκταθούν:

- Kubernetes Scheduler, γραμμένο σε Go.
- Kubernetes Cluster Autoscaler, γραμμένο σε Go.
- Rok CSI driver, γραμμένο σε Python.

Εισάγει επίσης ένα νέο στοιχείο , το webhook του scheduler, το οποίο αναπτύξαμε εξ ολοκλήρου στη γλώσσα Go.

Χτίζουμε τα δομικά μέρη με αναπαράξιμο τρόπο, χρησιμοποιώντας Docker containers κατάλληλα για τη γλώσσα που είναι γραμμένο το κάθε μέρος. Για να περιγράψουμε και να αυτοματοποιήσουμε τη διαδικασία χτισίματος χρησιμοποιήσαμε Dockerfiles και Makefiles.

Προκειμένου να εγκαταστήσουμε τα δομικά μέρη (Cluster Autoscaler, Rok Scheduler, Rok Scheduler webhook) στη συστοιχία, γράψαμε YAML manifests που χρησιμοποιούν το δηλωτικό API του Kubernetes για να περιγράψουν τους απαραίτητα αντικείμενα που

πρέπει να δημιουργηθούν. Για να διευκολύνουμε τη διαχείριση των manifests, χρησιμοποιήσαμε το εργαλείο Kustomize . Το Kustomize είναι ένα εργαλείο διαχείρισης manifests που αξιοποιεί τη διαστρωμάτωση για τη διατήρηση των βασικών ρυθμίσεων των εφαρμογών και των δομικών στοιχείων, ενώ μπορεί να επικαλύψει κομμάτια των YAML manifests (δημιουργώντας patches) που παρακάμπτουν επιλεκτικά τις προεπιλεγμένες ρυθμίσεις χωρίς να αλλάζουν στην πραγματικότητα τα αρχικά αρχεία.

Καθώς στο τρέχον κεφάλαιο περιγράφουμε συνοπτικά την υλοποίηση μας, μπορείτε να δείτε την πλήρη υλοποίηση στο αντίστοιχο αγγλικό κεφάλαιο 4.

5 Επίλογος

Σε αυτό το κεφάλαιο, θα επαναδιατυπώσουμε τις συνεισφορές μας και θα συνοψίσουμε τι προσφέρει ο μηχανισμός μας. Τέλος, θα κλείσουμε αυτή τη διπλωματική εργασία αναφέροντας τι μελλοντικό έργο μπορεί να γίνει ώστε να εμπλουτιστεί ο μηχανισμός μας και να φτάσει στο σύνολο των δυνατοτήτων του.

5.1 Συμπερασματικά Σχόλια

Συνολικά, ο πρωταρχικός στόχος αυτής της διπλωματικής ήταν η υλοποίηση ενός σχεδιασμού που θα επιτρέψει την απρόσκοπτη αυτόματη κλιμάκωση και χρονοδρομολόγηση σε συστοιχίες με τοπική μόνιμη αποθήκευση. Ο στόχος αυτός επιτεύχθηκε. Η υλοποίησή μας παραδόθηκε σε αρκετές εταιρείες και εγκαταστάθηκε στης συστοιχίες παραγωγής τους. Η υλοποίησή μας είναι ιδιαίτερης σημασίας για τις εταιρείες αυτές, καθώς τους επιτρέπει να επωφεληθούν από όλα τα πλεονεκτήματα της τοπικής αποθήκευσης, ενώ το μέγεθος της συστοιχίας προσαρμόζεται δυναμικά στις απαιτήσεις του φορτίου εργασίας για υπολογιστικούς πόρους και τοπική χωρητικότητα.

Ο σχεδιασμός που υλοποιήσαμε είναι άμεσα συνυφασμένος με το λογισμικό Rok της Arrikto, καθώς το Rok καθιστά δυνατή τη μετακίνηση των τοπικών τόμων από κόμβο σε κόμβο χάρη στον αποδοτικό μηχανισμό snapshots που διαθέτει. Παρόλα αυτά, η λύση μας μπορεί να γενικευτεί και να προσαρμοστεί και σε άλλα συστήματα τοπικής αποθήκευσης. Ο μακροπρόθεσμος στόχος μας, που εκτείνεται πέρα από το πλαίσιο αυτής της διπλωματικής, είναι να γενικεύσουμε το σχεδιασμό και να τον ενσωματώσουμε

στο upstream project του Kubernetes. Αυτή είναι μια διαδικασία στην οποία αρχίσαμε να εμπλεκόμαστε παρακολουθώντας τις συναντήσεις των Kubernetes Storage [1] και Autoscaling [2] Special Interest Groups και σκοπεύουμε να εμπλακούμε ενεργά προκειμένου να συνεισφέρουμε το σύνολο του σχεδιασμού upstream. Τη στιγμή που αυτό το κείμενο γράφεται, έχουν γίνει merge κάποια αρχικά Pull Requests [3] [4] που υποβάλαμε.

5.2 Μελλοντικό Έργο

Παρόλο που υλοποιήσαμε αρκετές βελτιώσεις στον Kubernetes Scheduler και το Cluster Autoscaler, υπάρχουν ακόμη περιθώρια ανάπτυξης. Δεδομένου ότι πρόκειται για μία επαναληπτική διαδικασία, στις επόμενες επαναλήψεις θα θέλαμε να προσφέρουμε αυτές τις βελτιώσεις:

- Να επεκτείνουμε τον Scheduler ώστε να δεσμεύει τον αποθηκευτικό χώρο κατά τη χρονοδρομολόγηση ενός Pod (στο στάδιο `Reserve` του κύκλου χρονοδρομολόγησης), προκειμένου να αποτρέπονται πιθανές συνθήκες ανταγωνισμού.
- Να επεκτείνουμε τον Cluster Autoscaler να λαμβάνει υπόψη τον αποθηκευτικό χώρο που απαιτείται για τα PVCs πολλαπλών Pods κατά την κλιμάκωση προς τα κάτω. Ο τρέχων σχεδιασμός ελέγχει μόνο αν τα PVs ενός μόνο Pod μπορούν να χωρέσουν σε έναν κόμβο, αλλά όχι τα PVs πολλαπλών Pods. Παρόλο που μια λανθασμένη απόφαση για την κλιμάκωση προς τα κάτω λόγω αυτής της έλλειψης θα διορθωθεί με μια επόμενη κλιμάκωση προς τα πάνω, η λήψη της σωστής απόφασης εξ' αρχής θα ήταν πολύ πιο αποτελεσματική.
- Να επεκτείνουμε την τρέχουσα υλοποίηση της διεπαφή `Estimator` του Autoscaler, δηλαδή τον `BinPackingEstimator` ώστε να λαμβάνει υπόψη την αποθηκευτική χωρητικότητα και να υπολογίζει, έτσι, πόσοι κόμβοι απαιτούνται όσον αφορά τα αιτήματα αποθηκευτικού χώρου πολλαπλών Pods ενός StatefulSet. Ο τρέχων σχεδιασμός προσθέτει έναν έναν τους κόμβους μέχρι όλα τα Pods να λάβουν τον ζητούμενο αποθηκευτικό χώρο. Θα ήταν πολύ πιο αποδοτικό αν ξέραμε εξ' αρχής πόσοι κόμβοι χρειάζονται ώστε να προστεθούν μονομιάς στη συστοιχία.

[1] https://github.com/kubernetes/community/blob/master/sig-storage/README.md
[2] https://github.com/kubernetes/community/blob/master/sig-autoscaling/README.md
[3] https://github.com/kubernetes/autoscaler/pull/4877
[4] https://github.com/kubernetes/autoscaler/pull/4842

Τέλος, όπως έχουμε ήδη αναφέρει, ο μακροπρόθεσμος στόχος μας είναι να συνεισφέρουμε το σχεδιασμό μας στο upstream project του Kubernetes.

1 Introduction

In this first chapter, we outline the scope of our work. We provide a brief overview of the task at hand, and we illustrate the gap that there is to fill. Then, we review the existing approaches, highlighting their offerings and drawbacks. Moving on, we give a high-level overview of the mechanism we built. Finally, we present the structure of this thesis

1.1 Motivation

Kubernetes is the de-facto container orchestrator choice for every company going cloud-native. The popularity of Kubernetes arises from the fact that it makes application life-cycle management for container-based applications a lot easier for DevOps via a declarative desired state-based management approach. It exposes a powerful declarative API that developers can use to describe the desired state of an application in terms of Pods, Services, etc. Kubernetes controllers take immediate actions to bring the observed state of the system to the desired state. Kubernetes can run on-premises and on most major cloud providers such as Amazon Web Services (AWS), Google Cloud, and Microsoft Azure, which is also key to its success.

By packaging code and dependency into containers, the development teams can use standard code units that start and terminate quickly to allow applications to scale to any size. They can use containers to package entire applications and move them to the cloud without needing to undergo code changes. Kubernetes act as an orchestrator platform to let large numbers of containers work harmoniously together and reduce operational burdens.

The Kubernetes Scheduler and Cluster Autoscaler (Autoscaler) work together to ensure the cluster's workload is running. The Scheduler ensures that the workload units (Pods) are assigned to cluster nodes with sufficient resources, i.e., memory, CPU, and storage. At the same time, the Cluster Autoscaler maintains an appropriate number of nodes in the cluster that allows the workload to run seamlessly without wasting any excess resources. The latter implies that the Cluster Autoscaler is a component that enables enterprises to optimize costs by dynamically scaling the number of nodes in the cluster in response to the current cluster workload and, thus, meet dynamic demand. Without the Cluster Autoscaler, the enterprises are bound to use a fixed-size cluster, which leads to either being charged for unneeded resources in the cluster, or the necessary resources would not be enough for the workload to run.

Combining Kubernetes with local persistent volumes allows users to access the local storage of a node in the cluster through the standard Kubernetes `PersistentVolumeClaim` interface in a simple and portable way. The primary benefit of local persistent volumes over remote persistent storage (e.g., network-attached volumes) is performance: local disks usually offer higher IOPS, higher throughput, and lower latency than remote storage systems. For instance, by attaching non-volatile memory express (NVMe) disks to a node, the end-user can benefit from a huge performance boost when executing applications that demand heavy storage utilization. For instance, in the case of enterprises, local storage can enable them to optimize the speed that they run the disk-intensive workload, such as machine learning, big data analysis tasks, etc., which is a crucial factor for revenue.

Enabling the seamless operation of the Cluster Autoscaler on Kubernetes clusters that leverage local storage is of high importance for enterprises. Local storage will enable them to run disk-intensive workload efficiently and fast, the Cluster Autoscaler will maintain the appropriate size of the cluster and the Autoscaler will ensure that workload runs on the right kind of node. This triple combination works towards infrastructure cost reduction, which is highly important for enterprises.

However, working with local persistent storage has a few implications that Kubernetes has not yet resolved:

- The Scheduler does not consider the available local storage when scheduling the workload units.

- The Cluster Autoscaler does not scale down (remove nodes) the cluster if local volumes are provisioned on the nodes since the removal would lead to data loss.
- The Cluster Autoscaler does not scale up the cluster (add nodes) when the nodes do not have enough local storage for the workload requests.

Motivated by the significance and the impact of using local persistent storage on Kubernetes clusters, in this thesis, we propose extensions for the Kubernetes Scheduler and the Cluster Autoscaler to enable seamless operation with local storage. More specifically, our work is two-fold:

- We propose a extensions so that the Cluster Autoscaler can scale down and up clusters with local persistent storage.
- We propose extensions for the Scheduler to schedule the workload on nodes with the required amount of local storage available.

1.2 Problem Statement

As stated in section 1.1, this thesis is motivated by local persistent storage's benefits. However, the integration of the Kubernetes Scheduler and the Cluster Autoscaler with local storage has a few implications that are not yet resolved:

- The Scheduler does not schedule the workload considering the available storage. Without considering the available storage, it may schedule a Pod onto an unsuitable node where the storage driver cannot provide the requested volumes because the underlying storage system does not have sufficient capacity.
- The Cluster Autoscaler does not scale down clusters with local persistent storage since the local data reside on each node, and removing the node would lead to losing access to these data.
- The Cluster Autoscaler does not scale up clusters, i.e., it does not add nodes in the cluster when there is not enough local storage for the workload to run.

In this thesis, we make design proposals and implement them to overcome these problems and enable the seamless operation of the Kubernetes Scheduler and the Cluster Autoscaler with local persistent storage.

1.3 Existing Solutions

Currently, the Cluster Autoscaler does not support autoscaling for cluster nodes that use local persistent storage. Each volume that is leveraging local storage will be considered by the Autoscaler to be only accessible by that node, and, thus, it will not remove the node where the data live; in other words, scaling down with local volumes is infeasible. Moreover, there is no option to scale up when there is not enough local storage for the workload that requests it. There are no existing solutions to fix these problems.

Regarding the Kubernetes Scheduler, the Kubernetes community has currently implemented preliminary support for scheduling with storage consideration. They call the feature "Storage Capacity Tracking". Scheduling a Pod with Capacity storage capacity tracking allows the storage (CSI) drivers to publish information about the remaining capacity on each topology segment of the cluster. The Scheduler then uses that information to pick a suitable node for a Pod that requests volumes to be provisioned. However, the current approach comes with a few limitations:

- It does not attempt to model how scheduling decisions affect storage capacity. The Kubernetes SIG Storage team took this design decision to ease the development of the feature, since the effect can vary considerably depending on how the storage system handles storage. As a consequence of this decision, a Pod requesting multiple volumes to be provisioned might get scheduled on a node where there is only enough space for each of the volumes individually, without considering the total amount of storage needed to accommodate all the volumes. As a consequence, the scheduling of a Pod can fail permanently: one volume might get created successfully in a topology segment that does not have enough capacity left for the other volume. Then, the already provisioned volume will restrict future attempts to schedule the Pod. Manual intervention is necessary to recover from this state, for example, by increasing capacity or deleting the already created volume.

- The feature is only available on clusters running Kubernetes version 1.21 or later. Running Kubernetes 1.21 or later is a requirement that not all production clusters currently meet. In our case, a few enterprises are still using older versions of Kubernetes; thus, the Capacity Tracking feature not available.

As it comes for the Cluster Autoscaler, there are no reported solutions to overcome its current limitations.

1.4 Proposed Solution

As already explained in previous sections, this thesis aims to enable seamless autoscaling and scheduling on clusters that leverage local persistent storage. Our proposed design involves the extension of various components, namely, the Scheduler, the Cluster Autoscaler, and the controller of the local storage driver (CSI driver).

In a nutshell, we propose the following design:

- Extend the Rok CSI storage driver to report the available storage capacity on each cluster node on the Kubernetes `Node` objects. The driver will calculate the available storage by issuing commands to the underlying Local Volume Manager.
- Extend the Kubernetes Scheduler to consider the reported storage capacity of each node when scheduling a Pod that requests Rok's local volumes to be provisioned. We call the extended scheduler "Rok Scheduler".
- Integrate the Cluster Autoscaler with the Rok CSI mechanism to snapshot and protect local volumes when a node is removed, enabling the cluster's seamless scale-down.
- Extend the Kubernetes Cluster Autoscaler to consider the storage utilization when deciding whether to scale down a node and check if there is enough storage on other nodes to accommodate the Pods of the node.
- Extend the Kubernetes Cluster Autoscaler to scale up the cluster when there is insufficient storage.
- Extend the Kubernetes Cluster Autoscaler not to remove nodes of a cluster that are in the Unready state since the local volumes still live on the Unready nodes, and the removal of the node will lead to data loss.

In the context of this thesis, we use the "Rok" data management system by Arrikto, which integrates with Kubernetes using the Container Storage Interface (see section 2.8) and

exposes the local storage of a node as volumes for the workload to consume. Although the design implementation focuses on Rok, it can be easily generalized to any other local storage systems.

1.5 Outline

The rest of this thesis is organized as follows:

- In **chapter 2** we provide the theoretical background necessary for the reader to understand our work.
- In **chapter 3** we analyze the design of the Scheduler and the Cluster Autoscaler, expose parts of their mechanism and propose design changes for enabling their seamless operation when local persistent storage is involved.
- In **chapter 4** we analyze the implementation of our design.
- In **chapter 5** we provide a summary of our contributions and possible future work directions.

2 Background

In this chapter, we provide the theoretical background necessary for understanding the core practical ideas of the rest of the thesis.

2.1 Architectural Evolution

The popularity of microservices and containerization has exploded in recent years. The need for scalable, easily deployable applications has led to abandoning monolithic architectures, where all processes are tightly coupled and run as a single service, in favor of microservices. Microservices are an architectural and organizational approach for software development where software is composed of small independent services that communicate over well-defined APIs.

The current industry trend for microservices is to use containerization to deliver smaller, single-function modules, which work together to create more agile, scalable applications. Containerization is a form of virtualization where applications run in isolated user spaces, called containers, while using the same shared operating system. A container is essentially a fully packaged and portable computing environment. Everything an application needs to run –its binaries, libraries, configuration files, and dependencies– are encapsulated and isolated in their container.

The extensive use of containers has driven the need to automate containers' deployment and management. Container orchestration automates the operational effort required to run containerized workloads and services. Container orchestration includes various

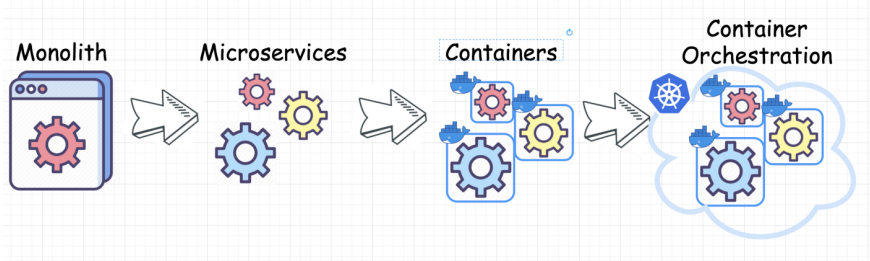


Figure 2.1: *Architectural evolution: From monolithic applications to containerized microservices that are managed by a container orchestrator*

actions software teams need to manage a container's lifecycle, including provisioning, deployment, scaling (up and down), networking, load balancing, and more.

Kubernetes is the most popular solution among various container orchestrators and has become the industry standard. Initially launched by Google, Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services that facilitates declarative configuration and automation.

Kubernetes operates on a cluster. A Kubernetes cluster is a set of node machines for running containerized applications. The cluster is the heart of Kubernetes' key advantage: the ability to schedule and run containers across a group of machines, be they physical or virtual, on-premises or in the cloud.

2.2 Kubernetes Fundamentals

Kubernetes is an open-source orchestrator for deploying containerized applications. It was initially developed by Google, inspired by a decade of experience deploying scalable, reliable systems in containers via application-oriented APIs. Since its introduction in 2014, Kubernetes has grown to be one of the world's largest and most popular open-source projects. It has become the standard API for building cloud-native applications in nearly every public cloud. Kubernetes is a proven distributed system infrastructure suitable for cloud-native developers of all scales. It provides the software necessary to build and deploy reliable, scalable distributed systems.

Kubernetes makes application lifecycle management for container-based applications much more effortless for DevOps via a declarative desired state-based management approach. It exposes a powerful declarative API that developers can use to describe the

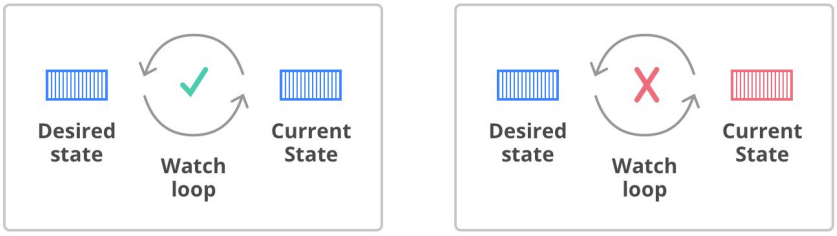


Figure 2.2: *The Kubernetes reconciliation loop*

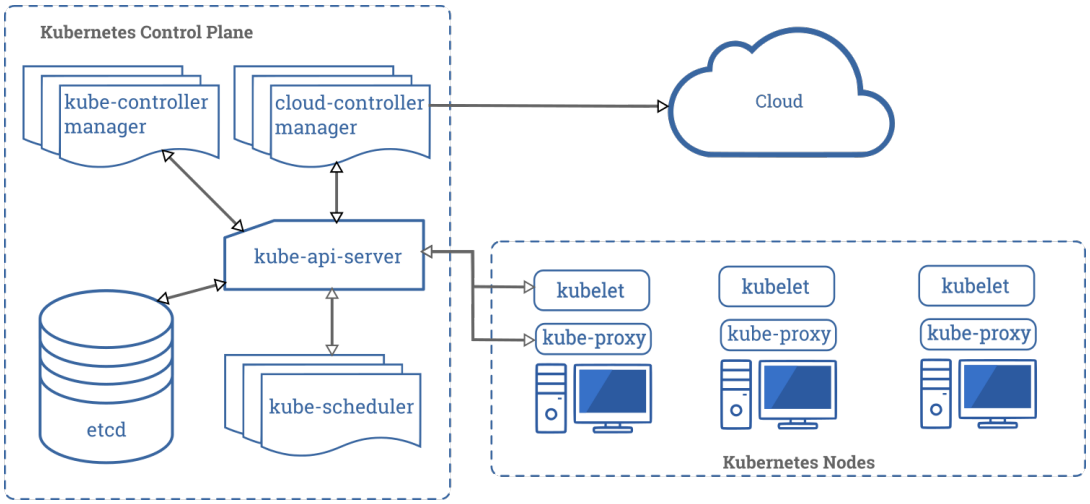


Figure 2.3: *Kubernetes Architecture*

desired state of an application in terms of Pods, Services, etc., and Kubernetes controllers will take immediate actions to bring the observed state of the system to the desired state.

2.2.1 Kubernetes Architecture

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

2.2.2 Kubernetes Control Plane Components

The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, starting up a new Pod when a deployment's replicas field is unsatisfied).

The main control plain components are:

- `kube-apiserver`: The API server is a component of the Kubernetes control plane that exposes the Kubernetes API. The API server is the front end of the Kubernetes control plane. The primary implementation of a Kubernetes API server is `kube-apiserver`. `Kube-apiserver` is designed to scale horizontally, i.e., it scales by deploying more instances. A cluster may run several instances of `kube-apiserver` and balance traffic between those instances.

- `etcd`: A consistent and highly-available key-value store that is used as Kubernetes' backing store for all cluster data. Distributed and fault-tolerant, etcd is an open-source, key-value store database that stores configuration data and information about the state of the cluster. Etcd may be configured externally, although it is often part of the Kubernetes control plane.

  Etcd stores the cluster state based on the Raft consensus algorithm. This helps cope with a common problem in the context of replicated state machines and involves multiple servers agreeing on values. Raft defines three different roles: leader, candidate, and follower, and achieves consensus by electing a leader.

  In this way, etcd acts as the single source of truth (SSOT) for all Kubernetes cluster components, responding to queries from the control plane and retrieving various parameters of the state of the containers, nodes, and Pods and other cluster components, in general.

- `kube-scheduler`: Control plane component that watches for newly created Pods with no assigned node and selects a node for them to run on. Factors considered for scheduling decisions include individual and collective resource requirements, hardware/software/policy constraints, affinity, and anti-affinity specifications, data locality, inter-workload interference, and deadlines.

- `kube-controller-manager`: Control plane component that runs controller processes. Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process. Some types of these controllers are:
  - **Node controller**: Responsible for noticing and responding when nodes go down.
  - **Job controller**: Watches for `Job` objects that represent one-off tasks, then creates Pods to run those tasks to completion.
  - **Endpoints controller**: Populates the Endpoints object (that is, joins Services & Pods).
  - **Service Account** & **Token controllers**: Create default accounts and API access tokens for new namespaces.
- `cloud-controller-manager`: Kubernetes control plane component that embeds cloud-specific control logic. The cloud controller manager lets a user link their cluster into their cloud provider's API, and separates out the components that interact with that cloud platform from components that only interact with their cluster. The cloud-controller-manager only runs controllers that are specific to your cloud provider. If a Kubernetes cluster runs on a user's premises, the cluster does not have a cloud controller manager.

2.2.3 Kubernetes Node Components

Node components run on every node, maintaining running Pods and providing the Kubernetes runtime environment.

Every node runs the following components:

- `kubelet`: An agent that makes sure that containers are running in a Pod.
- `kube-proxy`: kube-proxy is a network proxy implementing part of the Kubernetes Service concept. Kube-proxy maintains network rules on nodes. These network rules allow network communication to the user's Pods from network sessions inside or outside of their cluster. Kube-proxy uses the operating system

packet filtering layer if there is one and it's available. Otherwise, kube-proxy forwards the traffic itself.

- `Container Runtime`: The container runtime is the software that is responsible for running containers. Kubernetes supports container runtimes such as containerd, CRI-O, and any other implementation of the Kubernetes CRI (Container Runtime Interface).

2.2.4 The Kubernetes API

The core of Kubernetes' control plane is the API server. The API server exposes an HTTP API that lets end-users, different cluster parts, and external components communicate. The Kubernetes API lets the user query and manipulate the state of API objects in Kubernetes (for example, Pods, Namespaces, ConfigMaps, and Events). Kubernetes generally leverages common RESTful terminology to describe the API concepts:

- A *resource type* is the name used in the URL (Pods, Namespaces, Services).
- All resource types have a concrete representation (their object schema) which is called a *kind*.
- A list of instances of a resource is known as a *collection*.
- A single instance of a resource type is called a resource and usually represents an object.

Almost all object resource types support the standard HTTP verbs - `GET`, `POST`, `PUT`, `PATCH`, and `DELETE`. Kubernetes also uses its own, often written lowercase, to distinguish them from HTTP verbs. Kubernetes uses the term "list" to describe returning a collection of resources to distinguish from retrieving a single resource, usually called a "get". All resource types are scoped either to the cluster or to a *namespace*. A namespace-scoped resource type will be deleted when its namespace is deleted, and access to that resource type is controlled by authorization checks on the namespace scope.

2.2.5 Kubernetes Objects

Kubernetes *objects* are persistent entities in the Kubernetes system. Kubernetes uses these entities to represent the state of the cluster. Specifically, they can describe:

- What containerized applications are running and on which nodes.
- The resources available to those applications.
- The policies around how those applications behave, such as restart policies, upgrades, and fault-tolerance.

A Kubernetes object is a "record of intent"; once a user creates the object, the Kubernetes system will constantly work to ensure that the object exists. By creating an object, a user is effectively telling the Kubernetes system what they want their cluster's workload to look like.

2.2.5.1 The Pod Object

Pods are the smallest deployable artifact in a Kubernetes cluster. A *Pod* represents a collection of application containers and volumes running in the same execution environment. This means all of the containers in a Pod always land on the same machine. Each container within a Pod runs in its own cgroup, but they share several Linux namespaces. Applications running in the same Pod share the same IP address and port space (network namespace), have the same hostname (UTS namespace), and can communicate using native interprocess communication channels over System V IPC or POSIX message queues (IPC namespace). However, applications in different Pods are isolated from each other; they have different IP addresses, different hostnames, etc. Containers in different Pods running on the same node might also be on different servers.

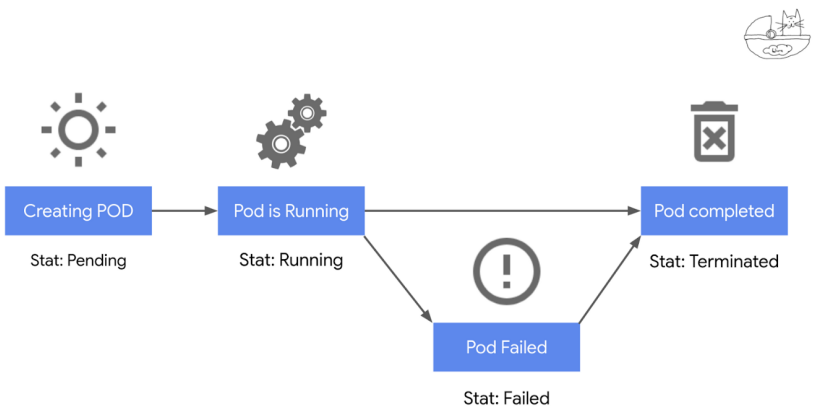


Figure 2.4: *The lifecycle of a Pod*

Phase The *phase* of a Pod is a simple, high-level summary of where the Pod is in its lifecycle. Each Pod follows a defined lifecycle, starting in the `Pending` phase, moving through `Running` if at least one of its primary containers starts OK, and then through either the `Succeeded` or `Failed` phases depending on whether any container in the Pod terminated in failure.

More specifically, the phase of a Pod can be:

- `Pending`: the Pod has been accepted by the system, but one or more of the containers has not been started. This includes time before being bound to a node, as well as time spent pulling images onto the host.
- `Running`: the Pod has been bound to a node and all of the containers have been started. At least one container is still running or is in the process of being restarted.
- Succeeded: all the containers of the Pod have voluntarily terminated with a container exit code of 0, and the system is not going to restart any of these containers.
- `Failed`: all the containers of the Pod have terminated, and at least one container has terminated in a failure (exited with a non-zero exit code or was stopped by the system).
- `Unknown`: for some reason the state of the Pod could not be obtained, typically due to an error in communicating with the host of the Pod.

Status A Pod has a `PodStatus`, which has an array of `PodConditions` through which the Pod has or has not passed:

- `PodScheduled`: the Pod has been scheduled to a node.
- `ContainersReady`: all containers in the Pod are ready.
- `Initialized`: all init containers have completed successfully.
- `Ready`: the Pod is able to serve requests and should be added to the load balancing pools of all matching Services.

Unschedulable Pods The cluster scheduler is responsible for assigning a node for the Pod to run on. It assigns a node by setting the `spec.nodeName` field of the Pod. If the scheduler fails to find a place to run the Pod, it sets `PodScheduled` `PodCondition` to `False` and reason to `Unschedulable`. An unschedulable Pod will remain in `Pending` phase.

In the context of this thesis, we will refer to a Pod that could not be scheduled as "*unschedulable Pod*" or, equivalently, "*Pending Pod*".

Resource requests and limits Compute resources are measurable quantities that can be requested, allocated, and consumed. For instance, but not limited to, CPU and memory are some types of computing resources.

The user can specify resource requests and limits for each container of the Pod. The scheduler uses the requests to decide which node assign to the Pod. The kubelet uses the limits for a container and enforces them so that the running container cannot use more of that resource than the limit a user has set. The kubelet also reserves at least the requested amount of that system resource specifically for that container to use. If the node where a Pod is running has enough of that resources available, it is possible (and allowed) for a container to use more than requested. However, a container cannot use more than the limit of that resource.

Listing 2.1: *Requests and limits of a Pod's container*

```
apiVersion: v1
kind: Pod
metadata:
  name: a-pod
spec:
  containers:
  - name: app
    image: nginx
resources:
  requests:
    memory: "100M"
    cpu: "200m"
  limits:
    memory: "150M"
    cpu: "500m"
```

2.2.5.2 The `Node` Object

Kubernetes runs the workload by placing Pods to run on nodes. Depending on the cluster, a node may be a virtual or physical machine. The control plane manages each node and contains the services necessary to run Pods. A node registered in the Kubernetes cluster is represented using a `Node` object on the API Server.

Node taints Taints and tolerations are a mechanism that users can use to ensure that Pods are not placed on inappropriate nodes. Taints are added to nodes, while tolerations are defined in the Pod specification. A node can have one or many taints associated with

it. When a user taints a node, it repells all the Pods except those that have a toleration for that taint.

A taint can produce three possible effects:

- `NoSchedule`: The scheduler will only allow scheduling Pods that have tolerations for the tainted nodes.
- `PreferNoSchedule`: The scheduler will try to avoid scheduling Pods that don't have tolerations for the tainted nodes.
- `NoExecute`: Kubernetes will evict the running Pods from the nodes if the Pods don't have tolerations for the tainted nodes.

Node status Each `Node` object has a `/status` subresource that indicates the status of the node. The status contains multiple conditions for the node and is managed by the node controller. Some conditions that are often encountered include:

- `MemoryPressure`: If `True`, it indicates that the node is running out of memory.
- `DiskPressure`: A `True` value in this field indicates that the node lacks enough space.
- `PIDPressure`: If too many processes are running on the node, this field will be `True`.
- `NetworkUnavailable`: If the network for the node is not correctly configured, this will be `True`.
- `Ready`: If the node is healthy and ready to accept Pods, this will be `True`. In this field, a `False` is equivalent to the `NotReady` status in the get nodes output. It can also have the Unknown value, which means the node controller has not heard from the node in the last `node-monitor-grace-period`.

In the context of the Cluster Autoscaler and the Scheduler, a node will be considered as "*Unready*" if:

- It has `Pod.spec.unschedulable` field. This field indicates the node shall not accept Pods.
- The `Node` object does not have any condition of type `Ready`.
- A condition of type `Ready` exists and the its status is `False`.

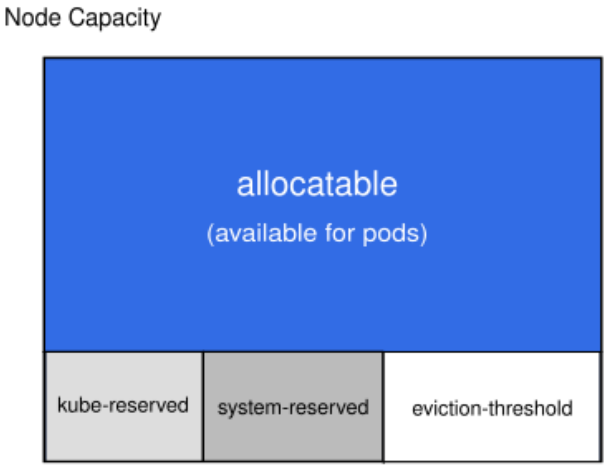


Figure 2.5: *Node resources: capacity and allocatable*

- A condition of type `DiskPressure` or `PIDPressure` or `NetworkUnavailable` with its corresponding status set to `True`. exists.

The rest of the nodes shall be considered as `Ready`.

A node that cannot accept Pods will be referred to as "*unschedulable*" node.

Node allocatable *Allocatable* on a Kubernetes node is defined as the amount of computing resources that are available for Pods. The total resources (capacity) of a node are categorized into:

- `kube-reserved`: resource reservation for kubernetes system daemons like the kubelet, container runtime, node problem detector, etc. It is not meant to reserve resources for system daemons that are run as Pods.
- `system-reserved`: resource reservation for OS system daemons like sshd, udev, etc. system-reserved should reserve memory for the kernel too since kernel memory is not accounted to Pods in Kubernetes at this time.
- `eviction-threshold`: specifies limits that trigger evictions when node resources drop below the reserved value.
- `allocatable`: the remaining node resources available for scheduling of Pods.

2.2.5.3 The `PodDisruptionBudget` Object

Pods do not disappear until someone (a person or a controller) destroys them or there is an unavoidable hardware or system software error.

The unavoidable cases are called *involuntary disruptions* and include:

- A hardware failure of the physical machine backing the node.
- Cluster administrator deletes VM (instance) by mistake.
- Cloud provider or hypervisor failure makes VM disappear.
- A kernel panic.
- The node disappears from the cluster due to cluster network partition.
- Eviction of a Pod due to the node being out-of-resources.

All other cases are called *voluntary disruptions*. These include both actions initiated by the application owner and those initiated by a Cluster Administrator. Typical voluntary disruptions include:

- Deleting the deployment or other controller that manages the Pod.
- Updating a deployment's Pod template causing a restart.
- Directly deleting a Pod.
- Draining a node for repair or upgrade.
- Draining a node from a cluster to scale the cluster down.
- Removing a Pod from a node to permit something else to fit on that node.

A `PodDisruptionBudget` (PDB) limits the number of Pods of a replicated application that are down simultaneously from voluntary disruptions. For example, a web front end might want to ensure that the number of replicas serving load never falls below a certain percentage of the total.

A PDB specifies the number of replicas that an application can tolerate having, relative to how many it is intended to have. For example, a Deployment that has a `.spec.replicas: 5` is supposed to have 5 Pods at any given time. If its PDB allows for there to be 4 at a time, then the Eviction API will allow voluntary disruption of one (but not two) Pods at a time.

Involuntary disruptions cannot be prevented by PDBs; however they do count against the budget. Pods which are deleted or unavailable due to a rolling upgrade to an application do count against the disruption budget.

2.2.5.4 The `Deployment` Object

A `Deployment` resource ensures that a specified number of Pod *replicas* are running at any time. In other words, a Deployment ensures that a Pod or homogeneous set of Pods are always up and available. If there are too many Pods, it will kill some. If there are too few, the Deployment will start more.

2.2.5.5 The `DaemonSet` Object

A `DaemonSet` (DS) resource ensures that all Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created.

2.2.5.6 The `PersistentVolumeClaim` object

A `PersistentVolumeClaim` (PVC, or equivalently referred to as "claim")) is a request for storage by a user. Claims can request specific size and access modes, e.g., they can be mounted `ReadWriteOnce`, `ReadOnlyMany` or `ReadWriteMany`, etc.

Phase The *Phase* of a PVC can be one of the following:

1. `Pending`: the PVC is not yet bound.
2. `Bound`: the PVC is bound to a PV.
3. `Lost`: the PVC lost its underlying PV. The claim was bound to a PV and this volume does not exist any longer and all data on it was lost.

2.2.5.7 The `PersistentVolume` object

A `PersistentVolume` (PV, or equivalently referred to as "volume") is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned

using Storage Classes. It is a resource in the cluster. PVs have a lifecycle independent of any individual Pod that uses the PV. This `API` object captures the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system.

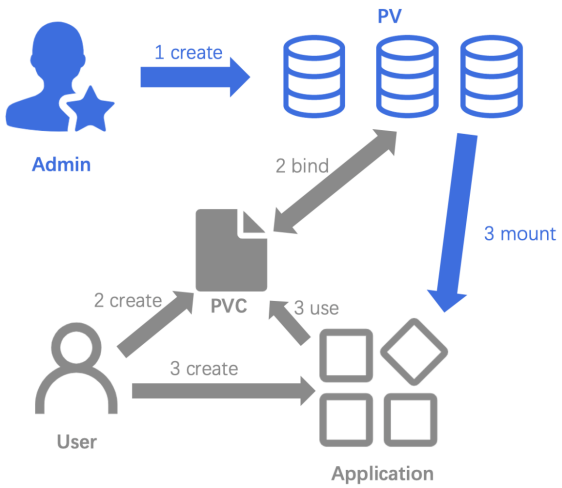


Figure 2.6: *The lifecycle of a PVC and PV in the case of static provisioning*

Phase A PV can be in one of the following phases:

- `Available`: the PV is not yet bound; it is available to be matched to a PVC.
- `Bound`: the PV is bound to a PVC.
- `Released`: the PVs must be recycled before becoming available again. This phase is used by the persistent volume claim binder to signal to another process to reclaim the resource.
- `Failed`: the PV has failed to be correctly recycled or deleted after being released from a claim.

Binding PVCs are requests for storage resources; each PVC gets bound to a PV that matches the PVC's requested storage amount and access modes. Each PV gets bound to one PVC only, and vice versa. The binding between them is bidirectional. A PV will remain unbound till it is matched to a PVC. The binding is illustrated in Figure 2.7.

The interaction between PVs and PVCs follows this lifecycle:

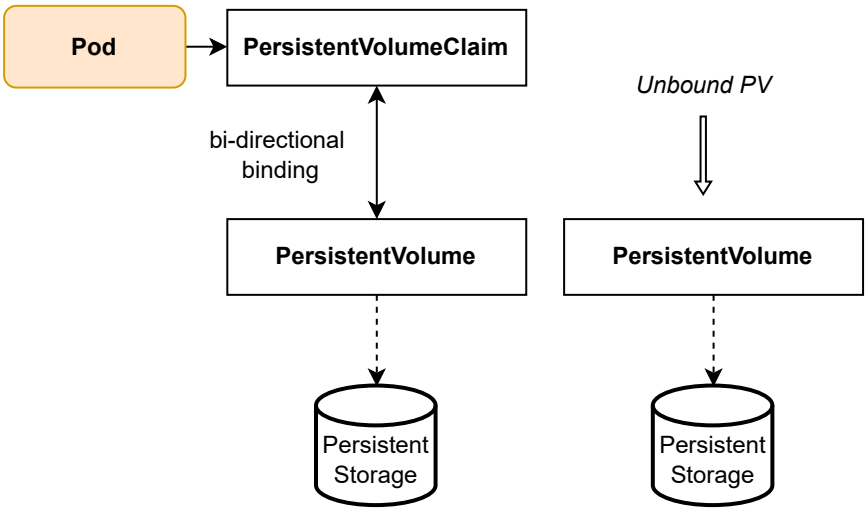


Figure 2.7: *A Pod requests a volume using a PVC and the PVC gets bound to a PV. The PV object stores the details for the underlying persistent storage piece.*

1. **Provisioning**: There are two ways PVs may be provisioned: statically or dynamically.

    - **Statically**: A cluster administrator creates some PVs. They carry the details of the actual storage, which is available for use by cluster users. They exist in the Kubernetes API and are available for consumption.
    - **Dynamically**: When none of the static PVs the
    - **Dynamically**: When none of the static PVs the administrator created matches a user's PersistentVolumeClaim, the storage system may try to dynamically provision a volume for the PVC. This provisioning relies on storage classes: the PVC must request a storage class, and the administrator must have created and configured that class for dynamic provisioning. Claims that do not specify a storage class effectively disable dynamic provisioning for themselves.

2. **Binding**: A user creates a PersistentVolumeClaim with a specific amount of storage requested and with certain access modes. A control loop (the PersistentVolumeController) in the Kubernetes control plane watches for new PVCs, finds a matching PV (if possible), and binds them together. If a PV was dynamically provisioned for a new PVC, the loop binds that PV to the PVC. Otherwise, the user will get at least what they asked for, but the volume may be more than what was requested. Once bound, PersistentVolumeClaim binds are exclusive, regardless

of how they were bound. A PVC to PV binding is a one-to-one mapping, using a `ClaimRef` which is a bidirectional binding between the PersistentVolume and the PersistentVolumeClaim. Claims will remain unbound indefinitely if a matching volume does not exist. Claims will be bound as matching volumes become available. For example, a cluster provisioned with many 50Gi PVs would not match a PVC requesting 100Gi. The PVC can be bound when a 100Gi PV is added to the cluster.

3. **Using**: Pods use claims as volumes. The cluster inspects the claim to find the bound volume and mounts that volume for a Pod. For volumes that support multiple access modes, the user specifies which mode is desired when using their claim as a volume in a Pod. Once a user has a claim and that claim is bound, the bound PV belongs to the user for as long as they need it. Users access their claimed PVs by including a `persistentVolumeClaim` section in a Pod's `volumes` block.

In the context of this thesis, a PVC will be also referred to as a "`claim`"

Node affinity A PV can specify node affinity to define constraints that limit what nodes this volume can be accessed from. The `nodeAffinity` field of the PV is a label selector that matches nodes with the appropriate labels. Labels are key/value pairs that are attached to objects.

The node affinity of a PV is used in the following way to indicate a volume is local to a node:

1. The storage driver sets on each `Node` object a unique label.
2. The storage driver sets the corresponding node affinity of the PV to match only the unique label of the node.

Listing 2.2 presents a PV with node affinity that matches node having the label `node: node-1`.

Listing 2.2: *A PV with node affinity*

```
1 apiVersion: v1
2 kind: PersistentVolume
3 metadata:
4   ...
```

```
  name: a-pv
spec:
  ...
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: node
        operator: In
        values:
        - node-1
```

2.2.5.8 The `StorageClass` Object

A `StorageClass` is a Kubernetes resource that enables dynamic storage provisioning. A StorageClass provides a way for administrators to describe the "classes" of storage they offer. The administrator configures the StorageClass, which can then no longer be modified.

A storage class can specify a `volumeBindingMode`, which is either `Immediate` or `WaitForFirstConsumer`:

- `Immediate`: Indicates that volume binding and dynamic provisioning occur once the user creates the PersistentVolumeClaim. For storage backends that are topology-constrained and not globally accessible from all nodes in the cluster, PersistentVolumes will be bound or provisioned without knowledge of the Pod's scheduling requirements, possibly resulting in unschedulable Pods.

- `WaitForFirstConsumer`: The binding and provisioning of a PersistentVolume will be delayed until a Pod using the PersistentVolumeClaim is created. PersistentVolumes will be selected or provisioned conforming to the topology that is specified by the Pod's scheduling constraints.

2.2.6 The Eviction API

When deleting a resource on Kubernetes, the API server will put a `deletionTimestamp` on the resource object. Unless there are any finalizers on the object, the object will be removed from the API Server.

In the case of Pods, apart from the classic `DELETE` operation, Kubernetes offers an extra API to initiate the deletion of the Pod: the `Eviction` API. The main difference with the

delete operation is that API-initiated evictions respect the configured PodDisruptionBudgets and `terminationGracePeriodSeconds`. So, if a user tries to *evict* a Pod and the corresponding PodDisruptionBudget does not allow the disruption of the Pod, the Pod will not be deleted. Instead, issuing a classical `DELETE` operation will remove the Pod, no matter what the PodDisruptionBudget specifies.

2.2.7 The Cordon & Drain Operations

The Kubernetes command-line tool, `kubectl`, allows a user to run commands against Kubernetes clusters. The tool allows the complete management of the cluster. Two essential operations used for the maintenance of the cluster are the *cordon* and the *drain* operations.

Cordon operation *Cordon* is an operation offered by the `kubectl` CLI tool that marks the node as *unschedulable*. Marking a node as unschedulable prevents the scheduler from placing new Pods onto that node but does not affect existing Pods running on it. This is a preparatory step before a node reboot or other maintenance.

The admin of the cluster can execute the cordon operation by running `kubectl cordon`. When cordoning a node, the tool adds the *unschedulable taint* [1] on the node and also sets the `nodes.spec.unschedulable` field to `True`.

We will refer to the action of marking the node as unschedulable as "*cordoning the node*" and the node as "*cordoned*".

Drain operation The *drain* operation is used to remove workload from a node. It is run in case the node needs maintenance, or it needs to be removed from a cluster. The drain operation cordons the node to mark it as unschedulable, and evicts all the Pods from the node. Evictions allow the Pod's containers to terminate gracefully and will respect the PodDisruptionBudgets the user has specified.

The admin of the cluster can execute the drain operation by running `kubectl drain`. If `kubectl drain` returns successfully, it indicates that all the Pods have been safely

[1] Unschedulable taint: `node.kubernetes.io/unschedulable:NoSchedule`

evicted (respecting the desired graceful termination period and the PodDisruptionBudget that is defined). It is then safe to bring down the node by powering down its physical machine or deleting its virtual machine if it runs on a cloud platform.

2.3 Kubernetes Controllers

In Kubernetes, controllers are control loops that watch the state of the cluster, then make or request changes where needed. Each controller tries to move the current cluster state closer to the desired state. A controller tracks at least one Kubernetes resource type. These objects have a `spec` field representing the desired state. The controllers for that resource are responsible for making the current state come closer to that desired state.

In this section, we will describe some of the controller that play a significant role in the storage system of Kubernetes.

2.3.1 The PersistentVolume Controller

The `PersistentVolumeController` is a controller that synchronizes PersistentVolumeClaims and PersistentVolumes. It binds PVs and PVCs and manages their lifecycles. If the PVC references a StorageClass with static provisioning, the control loop attempts to find a matching PV and then binds it to the PVC. In the case of dynamic provisioning, as soon as the PV gets provisioned for the PVC, the control loop binds them together.

2.3.2 The AttachDetach Controller

The `AttachDetach` controller manages volume attach and detach operations. It looks for any Pods that get scheduled on a node and triggers the attach operation, i.e., it creates a `VolumeAttachment` object to signal the external attacher that it shall issue a `ControllerPublish` call to the CSI driver. Similarly, when no Pods use a volume on a node, the controller executes a detach operation: deletes the `VolumeAttachment` object to signal the external attacher it shall issue a `ControllerUnpublish` request to the CSI driver.

2.3.3 Kubernetes Admission Controllers

An *admission controller* is a piece of code that intercepts requests to the Kubernetes API server prior to the persistence of the object but after the request is authenticated and authorized. Admission controllers may be *validating*, *mutating*, or both. Mutating controllers may modify related objects to the requests they admit; validating controllers may not.

The admission control process proceeds in two phases. In the first phase, it runs the mutating admission controllers. In the second phase, it runs the validating admission controllers. If any controller in either phase rejects the request, the entire request is rejected immediately and an error is returned to the end-user.

Various admission controllers come compiled into the `kube-apiserver` binary, and out of them, there are two controllers of particular interest, the `MutatingAdmissionWebhook` and `ValidatingAdmissionWebhook`:

- `MutatingAdmissionWebhook`: This admission controller calls any mutating webhooks which match the request. Matching webhooks are called serially; each one may modify the object if desired.
- `ValidatingAdmissionWebhook`: This admission controller calls any validating webhooks which match the request. Matching webhooks are called in parallel; if any of them rejects the request, the request fails.

The admission controller phases are shown in Figure 2.8.

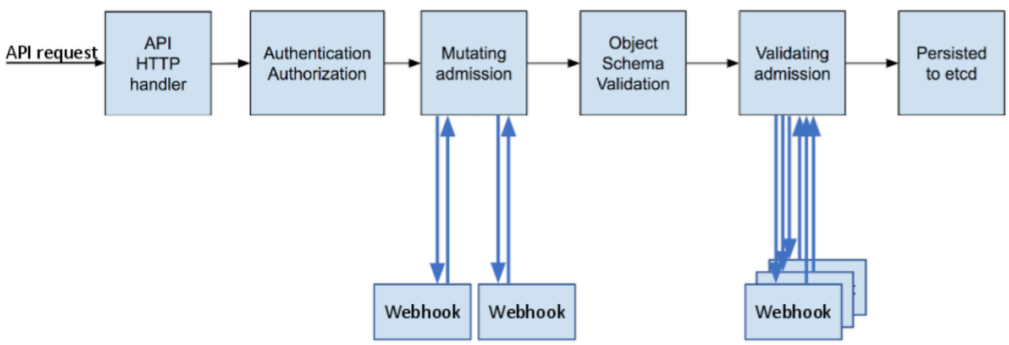


Figure 2.8: *Admission controller phases*

2.4 Kubernetes Admission Webhooks

Admission webhooks are HTTP callbacks that receive admission requests and do something with them. Two types of admission webhooks can be defined: *validating admission* webhook and *mutating admission* webhook.

Mutating admission webhooks are invoked first, and can modify objects sent to the API server to enforce custom defaults. After all object modifications are complete, and after the incoming object is validated by the API server, validating admission webhooks are invoked and can reject requests to enforce custom policies.The admin of the cluster can dynamically configure what resources are subject to what admission webhooks via `ValidatingWebhookConfiguration` or `MutatingWebhookConfiguration` API objects.

`MutatingWebhookConfiguration` Object Each `MutatingWebhookConfiguration` contains a list of webhooks, specified at `webhooks` field. Each of the webhooks defined, may specify the following fields:

- `rules`: A list of rules used to determine if a request to the API server should be sent to the webhook. Each rule specifies one or more operations, apiGroups, apiVersions, and resources, and a resource scope.
- `failurePolicy`: Defines how unrecognized errors and timeout errors from the admission webhook are handled. Allowed values are `Ignore` or `Fail`.
- `namespaceSelector`: Defines whether to run the webhook on a request for a namespaced resource (or a `Namespace` object) based on whether the namespace labels match the selector. If the object is a cluster scoped resource other than a Namespace, `namespaceSelector` has no effect.

2.5 The Kubernetes Operator Pattern

A Kubernetes operator is a custom application-specific controller that extends the functionality of the Kubernetes API to create, configure, and manage instances of complex applications on behalf of a Kubernetes user. It builds upon the fundamental Kubernetes resource and controller concepts but includes domain or application-specific knowledge to automate the entire life cycle of the software it manages. It uses *custom resources* to

manage applications and their components. The user within a custom resource provides high-level configuration and settings. The Kubernetes operator translates the high-level directives into low-level actions based on best practices embedded within the operator's logic.

A `CustomResourceDefinition` object (CRD) defines a custom resource and lists out all the configurations available to users of the operator. The Kubernetes API can handle custom resource definitions just like built-in objects, including interaction via `kubectl` and inclusion in role-based access control (RBAC) policies.

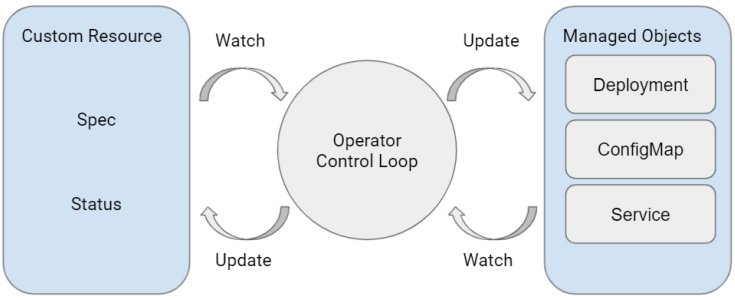


Figure 2.9: *The Kubernetes operator pattern*

2.6 The Kubernetes Scheduler

A scheduler on a Kubernetes cluster is a component that watches for newly created Pods that have no node assigned. For every Pod the scheduler discovers, the scheduler becomes responsible for finding the best node for that Pod to run on. In this section, we will present the background needed to understand how scheduling occurs.

2.6.1 Scheduling Fundamentals

Multiple schedulers A *scheduler* is a component on a Kubernetes cluster that assigns a node for each Pod to run on. Kubernetes clusters ship with the default scheduler, namely `kube-scheduler`, which runs as part of the control plane. However, multiple schedulers may run on a cluster; in this case, each Pod must specify the scheduler's name that shall handle it by setting its `spec.schedulerName` field to the name of the preferred

scheduler. If a scheduler name is not explicitly specified, the Pod will be scheduled using the default scheduler.

Listing 2.3: *A Pod to be scheduled by* `another-scheduler`

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: a-pod
  spec:
    schedulerName: another-scheduler
    containers:
    - name: a-container
      image: nginx
```

Feasible nodes Each Pod has different requirements, e.g., CPU, memory, and node affinity, which affect which nodes the Pod can run on. Factors that need to be taken into account for scheduling decisions include individual and collective resource requirements, hardware / software / policy constraints, affinity and anti-affinity specifications, data locality, and inter-workload interference. The nodes that meet the scheduling requirements for a Pod are called *feasible* nodes. If none of the nodes in the cluster are feasible, the Pod will remain unscheduled, i.e., it will not be assigned any node to run on.

The `kube-scheduler` The Kubernetes default scheduler, `kube-scheduler` selects a node for the Pod in a 3-step operation:

1. *Filtering*: The scheduler finds the set of nodes where it is feasible to schedule the Pod. It executes a series of filtering plugins that evaluate whether the Pod can be placed on the examined node. For example, the `PodFitsResources` filter checks whether a candidate node has enough resources (CPU, RAM, GPU) to meet the Pod's specific resource requests. A node is *feasible* if all the filter plugins consider the node as feasible for the Pod. This step calculates a node list with suitable nodes; if the list is empty, the Pod is *unschedulable*.
2. *Scoring*: The scheduler ranks the feasible nodes to choose the most suitable Pod placement. The scheduler assigns a score to each feasible node based on the active scoring rules.
3. The scheduler assigns the Pod to the node with the highest ranking. If there is more than one node with equal scores, it randomly selects one of these. The

scheduler assigns the Pod to a node by setting the `spec.nodeName` field of the Pod to the name of the selected node.

2.6.2 The Scheduling Framework

The scheduling framework is a pluggable architecture for the Kubernetes scheduler. It adds a set of *plugin* APIs to the existing scheduler. Plugins are compiled into the scheduler. The APIs allow most scheduling features to be implemented as plugins while keeping the scheduling core lightweight and maintainable.

Scheduling & binding cycles The scheduler stores a queue of Pods waiting to be scheduled. It picks a Pod from the queue and attempts to schedule it. Each attempt to schedule a pod is split into two phases:

- *Scheduling cycle*: Selects a node for the Pod. The scheduling cycles of different Pods are run *serially*.
- *Binding cycle*: Applies that decision for the Pod to the cluster. Multiple binding cycles for different Pods are run *concurrently*.

A scheduling cycle and binding cycle together are referred to as the "*scheduling context*". A scheduling cycle or binding cycle can be aborted if the Pod is determined to be unschedulable or if there is an internal error. The Pod will be returned to the queue and retried. If a binding cycle is aborted, it will trigger the `Unreserve` method in the Reserve plugin.

The scheduling framework exposes some extension points. The plugins are registered to be called at one or more of these extension points. One plugin may register at multiple extension points. The extension points are illustrated in Figure 2.10.

Scheduler framework extension points The scheduler framework offers the following extensions points where each plugin can be registered:

- **`Queue sort`**: The scheduler uses these plugins to sort Pods in the scheduling queue. A queue sort plugin essentially will provide a `less(pod1, Pod2)` function. Only one queue sort plugin may be enabled at a time.

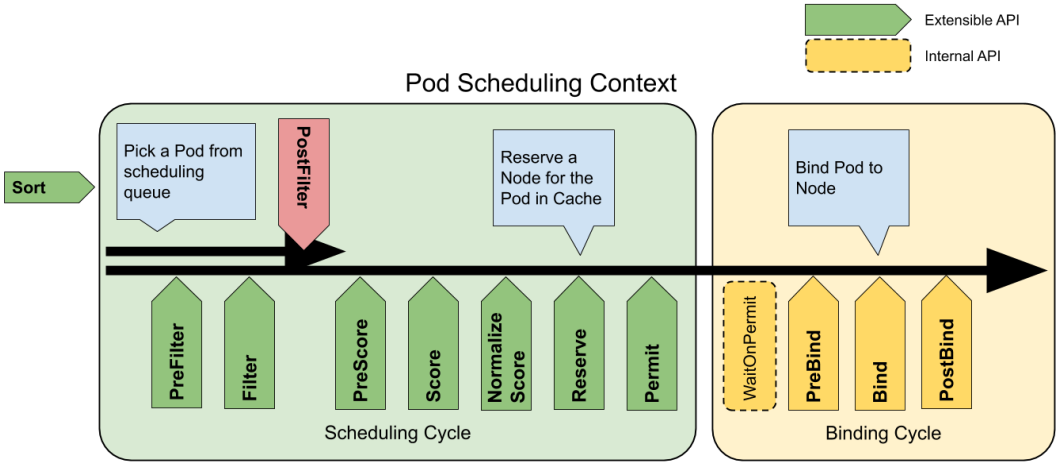


Figure 2.10: *The extension points of the scheduling framework*

- **`PreFilter`**: The scheduler uses these plugins to pre-process info about the Pod or to check certain conditions that the cluster or the Pod must meet. A PreFilter plugin should implement a `PreFilter` function. If `PreFilter` returns an error, the scheduler will abort the scheduling cycle..

- **`Filter`**: The scheduler uses these plugins to *filter out* nodes that cannot run the Pod. For each node, the scheduler will call the filter plugins in their configured order. If any filter plugin marks the node as *infeasible*, the scheduler will not call the remaining plugins for that evaluated node. The scheduler may evaluate nodes concurrently, and thus, it may call a `Filter` plugin more than once in the same scheduling cycle.

- **`PostFilter`**: The scheduler calls these plugins in their configured order after the `Filter` phase, but only if it did not find any feasible nodes for the Pod. If any of them marks the node as schedulable, the scheduler will not call the remaining plugins. A typical PostFilter implementation is the `preemption` plugin, which tries to make the Pod schedulable by preempting other Pods.

- **`PreScore`**: This is an informational extension point for performing pre-scoring work. The scheduler will call the plugins with a list of nodes that passed the filtering phase. A plugin may use this data to update the internal state or to generate logs or metrics.

- **`Scoring`**: This extension point has two phases:

  - The first phase is called "*score*" and is used to rank nodes that have passed the filtering phase. The scheduler will call the `Score` method of each scoring plugin for each node.
  - The second phase is "*normalize scoring*" and is used to modify scores before the scheduler computes a final ranking of nodes.

- **`Reserve`**: A plugin that implements the Reserve extension has two methods, namely `Reserve` and `Unreserve`. Plugins that maintain runtime state, i.e., *stateful plugins*, should use these phases to reserve and unreserve any resources in the internal state of the scheduler.

  The *Reserve* phase exists to prevent race conditions while the scheduler waits for the bind to succeed. The scheduler executes it before it binds a Pod to its designated node. The `Reserve` method of each Reserve plugin may succeed or fail.

  If the `Reserve` method of all plugins succeeds, the scheduler considers the Reserve phase to be successful and executes the rest of the scheduling cycle and the binding cycle.

  If one `Reserve` method call fails, the scheduler will not execute the subsequent phases and will run the `Unreserve` phase instead. The *Unreserve* phase exists to clean up the state associated with the reserved Pod. The scheduler calls the `Unreserve` method of all the `Reserve` plugins in the reverse order of `Reserve` method calls.

- **`Permit`**: The scheduler uses these plugins to prevent or delay the binding of a Pod. It executes them as the last step of a scheduling cycle; however, it waits for the permit phase to execute successfully at the beginning of a binding cycle, before executing the `PreBind` plugins.

The `CycleState` struct The various plugins running in the scheduling context of a single Pod share a common `CycleState` struct. `CycleState` provides a mechanism for plugins to store and retrieve arbitrary data. Data stored in the `CycleState` by one plugin can be read, altered, or deleted by another. `CycleState` does not provide any data protection, as all plugins are assumed to be trusted.

2.6.3 The VolumeBinding Plugin

The `VolumeBinding` plugin of the Kubernetes Scheduler binds Pod volumes in scheduling. The VolumeBinding plugin is registered on the `PreFilter`, `Filter`, `PreBind`, `Reserve`, and `Unreserve` extension points. The `PreFilter`, `Filter`, and `PreBind` phases of the plugin are of particular interest in the context of this thesis. We explain here briefly the operations that take place:

- `PreFilter`: Checks if a Pod has all its immediate (PVCs that request a storage class with `Immediate` binding mode) PVCs bound. If not all immediate PVCs are bound, the plugin returns an `UnschedulableAndUnresolvable` error.
- `Filter`: Evaluates if a Pod fits a node due to the volumes it requests:
  - For *bound PVCs*, it checks that the corresponding node affinity of the PV of each PVC is satisfied by the given node.
  - For each *unbound PVC*, it tries to find an available PVs that satisfies the PVC requirements (access mode, requested capacity) and has node affinity that matches the given node.
  - For each `unbound PVC` that did not find a matching PV (hereafter referred to as the "PVCs to provision"), it checks if there is enough space on the node to provision a volume.
  - The `Filter` method returns true if the following conditions hold:
    1. he PVs the (bound) PVCs are bound to are accessible from the node
    2. the unbound PVCs can be matched to an existing available PV that is accessible from the node or have the storage driver dynamically provision such a PV, if there is enough storage.
- `PreBind`: PreBind updates the API Server with the assumed bindings and waits until the PersistentVolume controller has wholly finished the binding operation. If binding errors, times out or gets undone, then an error will be returned to retry scheduling.

2.7 The Kubernetes Cluster Autoscaler

There are three different types of autoscaling in Kubernetes:

- **Cluster Autoscaler (Autoscaler)**: adjusts the number of nodes in the cluster when Pods fail to schedule or when nodes are underutilized.
- **Horizontal Pod Autoscaler (HPA)**: adjusts the number of replicas of an application.
- **Vertical Pod Autoscaler (VPA)**: adjusts the resource requests and limits of the containers of a Pod.

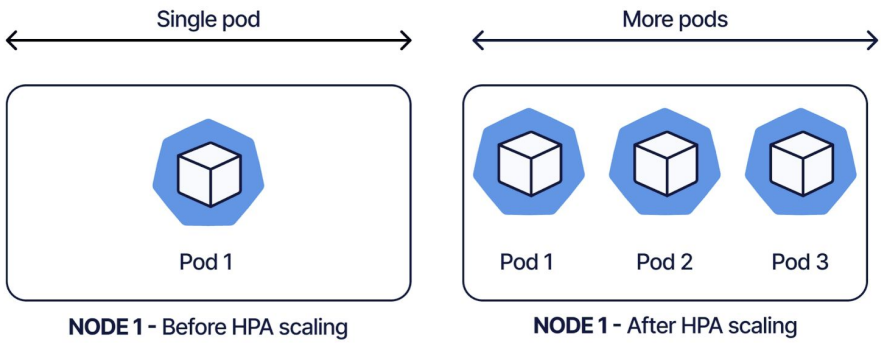


Figure 2.11: *Kubernetes Horizontal Pod Autoscaling*

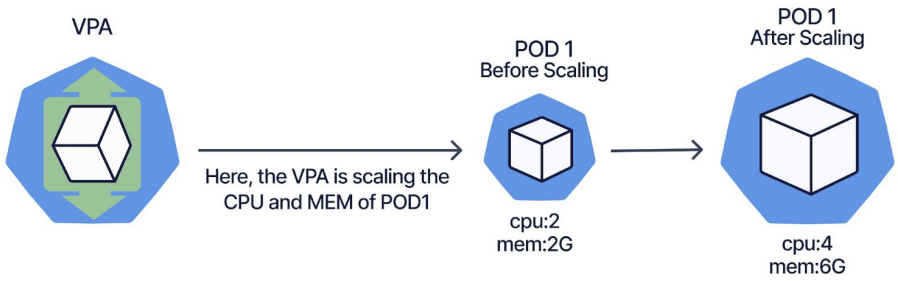


Figure 2.12: *Kubernetes Vertical Pod Autoscaling*

In the context of this thesis, we only discuss the Cluster Autoscaler. We will refer to it as the "Autoscaler", and we will imply that we talk about the Cluster Autoscaler.

2.7.1 Cluster Autoscaling Fundamentals

The Cluster Autoscaler is a component that automatically adjusts the Kubernetes cluster size. It automatically adds or removes nodes based on the Pods' resource requests compared to the nodes' available resources (see section 2.2.5.1); it does not measure the live CPU and memory usage.

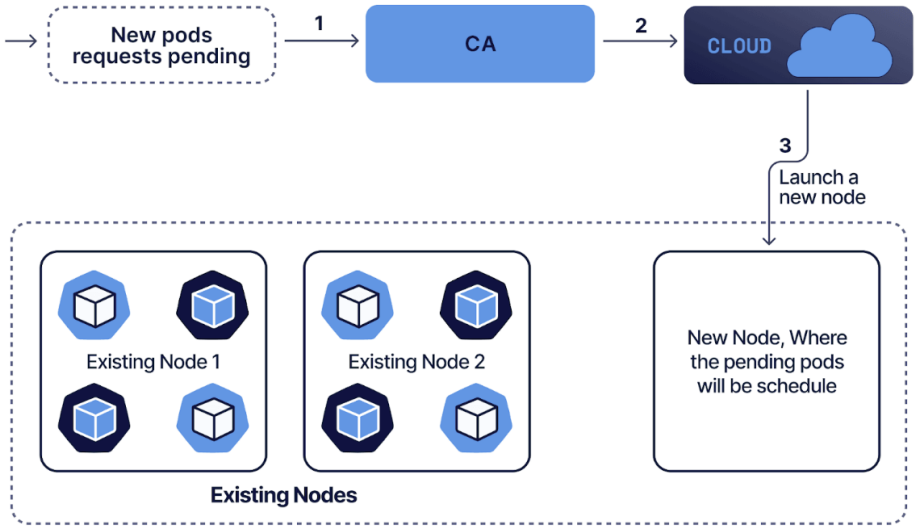


Figure 2.13: *Kubernetes Cluster Autoscaling*

If the scheduler could not schedule any Pods, the Autoscaler will add new nodes in the cluster to help the Pods get scheduled. This action is called *scale-up* of the cluster (or, equivalently, *scale-out*). If any nodes in the cluster are not needed, the Autoscaler will remove these nodes. This action is called *scale-down* of the cluster (or, equivalently, *scale-in*).

The following paragraphs explain the basic principles of scaling up and down a cluster.

2.7.2 Scale-Up Procedure

The Autoscaler periodically checks for any unschedulable Pods on the cluster and tries to find a new place for them to run.

The Autoscaler assumes that the underlying cluster runs on top of some *node groups*. Inside a node group, all machines have the same capacity and assigned labels. Increasing the size of a node group will create a new node that will be similar to those already in the cluster.

Based on the above assumption, the Autoscaler creates template nodes for each node group and checks if any unschedulable Pod would fit on a new node. If there are multiple node groups that, if increased, would help some Pods to run, the administrator can specify different strategies for the Autoscaler to choose which node group it shall

increase. This procedure may require multiple iterations before all the Pods are eventually scheduled.

2.7.3 Scale-Down Procedure

The Autoscaler periodically checks which nodes are unneeded, provided that no scale-up is needed. A node is considered for removal if all the following conditions hold:

- The sum of CPU and memory requests of all Pods running on this node is less than 50% of the node's allocatable resources. This threshold is configurable.
- All Pods running on the node (except Pods created by Daemons) can be moved to any other node in the cluster.

The Autoscaler removes a node if it remains unneeded for more than 10 minutes (configurable duration). It terminates one *non-empty* node at a time to reduce the risk of creating new unschedulable Pods. On the other hand, it terminates *Empty* nodes (nodes that run only DaemonSet Pods) in bulk. When a non-empty node is terminated, as mentioned above, all Pods should be migrated elsewhere. It achieves this by marking the node unschedulable and then evicting the Pods that run on it. As soon as each Pod is successfully evicted, the scheduler shall schedule it on a different node.

2.8 The Container Storage Interface

The *Container Storage Interface* (CSI) is a standard for exposing arbitrary block and file storage systems to containerized workloads on container orchestration systems (COs), such as Kubernetes. Using CSI, third-party storage providers can write and deploy plugins exposing new storage systems in Kubernetes without ever having to touch the core Kubernetes code.

2.8.1 CSI Driver Architecture

Kubernetes interacts with a CSI driver plugin through *Remote Procedure Calls* (RPCs). Each CSI driver consists of the following plugins:

- **Node Plugin**: A gRPC endpoint serving CSI RPCs that must run on the node where the provisioned volume will be published. It consists of the CSI driver that implements the CSI `Node` service and one or more sidecar containers. The kubelet of every node is responsible for issuing the CSI Node service calls. The kubelet issues the calls to the Node service of the driver through a UNIX domain socket on the host shared via a `HostPath` volume. The node plugin has direct access to the host for making block devices and filesystem mounts available to the kubelet.

- **Controller Plugin**: A gRPC endpoint serving CSI RPCs that may run on any node of the cluster. It consists of the CSI driver that implements the CSI `Controller` service and one or more sidecar containers. These controller sidecar containers typically interact with Kubernetes objects and make calls to the driver's CSI Controller service by sharing a UNIX domain socket through an `emptyDir` volume between the sidecars and CSI driver. It generally does not need direct access to the host.

The Google Remote Procedure Call We mentioned earlier that a container orchestrator interacts with the driver through RPCs. The most widely used RPCs are *Google Remote Procedure Calls* (gRPC). gRPC is an open-source, high-performance Remote Procedure Call (RPC) framework that can run in any environment. It uses HTTP/2 for transport, Protocol Buffers as the interface description language, and provides authentication, bidirectional streaming and flow control features, blocking or non-blocking bindings, and cancellation and timeouts. It generates cross-platform client and server bindings for many languages. gRPC clients and servers can run and talk to each other in various environments and can be written in any of gRPC's supported languages.

2.8.2 The CSI Remote Procedure Calls

The Container Storage Interface defines the RPCs a container orchestrator uses in order to interact with the storage driver. Each of the RPCs is an idempotent operation. The order the calls can be issued is show in Figure 2.15. The list of available RPCs is the following:

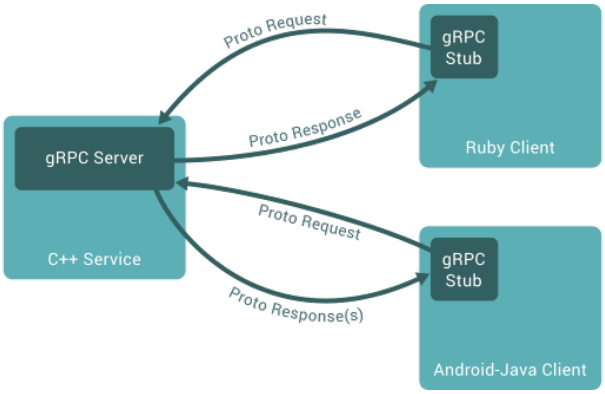


Figure 2.14: *The architecture of gRPC*

- `CreateVolume`: The external provisioner issues this RPC to the CSI Controller service, asking it to provision a new volume on behalf of a user. If the plugin is unable to complete the `CreateVolume` call successfully, it must return a non-OK gRPC code in the gRPC status.

  Of particular interest in the context of the thesis is the `RESOURCE_EXHAUSTED` code; If the controller plugin responds with this code, it indicates that it cannot provision the requested volume with the specified topology constraints, possibly due to insufficient storage capacity.

- `ControllerPublishVolume`: The external attacher issues this RPC to the CSI Controller service when Kubernetes wants to place a workload that uses the (already provisioned) volume onto a node. The plugin should perform the necessary work to make the volume available on the given node.

- `NodeStageVolume`: The kubelet issues this RPC to the CSI Node service when the volume is to be used by the first Pod on the node. It should be issued only after `NodePublishVolume` has succeeded. It is essentially used to format the volume and mount it on a staging directory on the node.

- `NodePublishVolume`: The kubelet issues this RPC to the CSI Node service when a Pod starts to run on a node. It essentially mounts the volume to the directory of the Pod.

- `NodeUnpublishVolume`: The kubelet issue this RPC to the CSI Node service to undo the work done by the corresponding `NodePublishVolume`. It essentially unmounts the volume from the directory of the Pod.

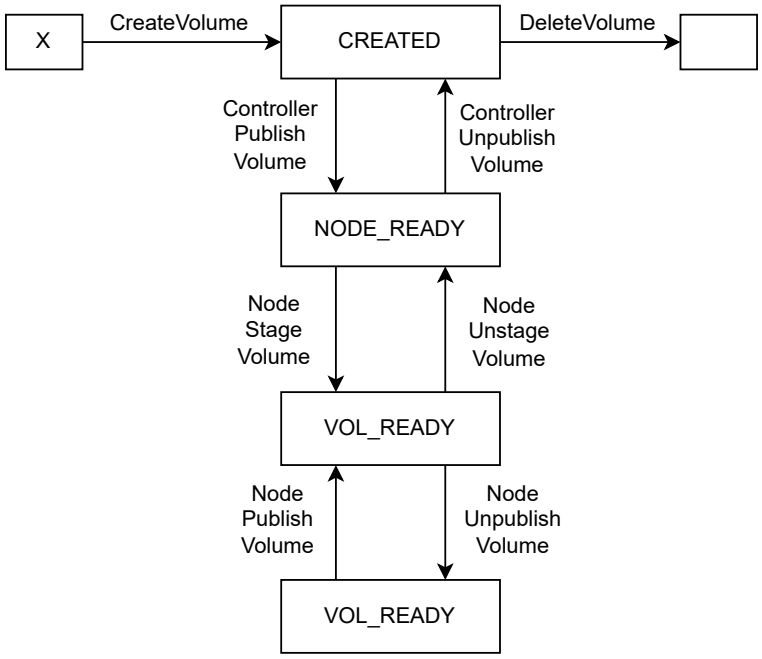


Figure 2.15: *The lifecycle of a dynamically provisioned volume, from creation to destruction.*

- `NodeUnstageVolume`: The kubelet issues the RPC to the CSI Node service to undo the work by the corresponding `NodeStageVolume`. It essentially unmounts the volume from the staging directory of the node.

- `ControllerUnpublishVolume`: The external attacher issues this RPC to the CSI Controller service to perform the work necessary for making the volume ready to be consumed by a different node. It essentially undoes any work done by the `ControllerPublishVolume`.

- `DeleteVolume`: The external provisioner issues this RPC to the CSI Controller service to deprovision a volume. It is the reverse operation of the `CreateVolume`.

2.8.3 Kubernetes CSI Sidecars

The Kubernetes CSI sidecars containers are a set of standard containers that aim to simplify the development and deployment of CSI drivers on Kubernetes. These containers contain common logic to watch the Kubernetes API, trigger appropriate operations against the *CSI driver* container, and update the Kubernetes API as appropriate. The containers are intended to be bundled with third-party CSI driver containers and de-

ployed together as Pods.

2.8.3.1 CSI External Provisioner

The CSI external provisioner is a sidecar container that watches the Kubernetes API server for `PersistentVolumeClaim` objects. If a PVC requests for dynamic provisioning of a volume and has the selected node annotation [2], the external provisioner issues a `CreateVolume` RPC against the CSI Controller service to provision a new volume accessible from the selected node. Suppose the Controller service responds with a `ResourceExhausted` status code. In that case, the external provisioner will remove the selected node annotation from the PVC, to signal back to the scheduler that the provisioning of the volume has failed, and it shall retry the scheduling. Once the external provisioner successfully provisions the volume, it creates a Kubernetes `PersistentVolume` object to represent the volume and binds it to the PVC.

The deletion of a `PersistentVolumeClaim` object bound to a `PersistentVolume` corresponding to this driver with a `delete` reclaim policy causes the external provisioner to trigger a `DeleteVolume` operation against the CSI Controller service to delete the volume. Once the volume is successfully deleted, the sidecar container deletes the `PersistentVolume` object representing the volume.

2.8.3.2 CSI External Attacher

The CSI external attacher is a sidecar container that watches the Kubernetes API server for `VolumeAttachment` objects and triggers `ControllerPublishVolume` and `ControllerUnpublishVolume` operations against a CSI endpoint.

2.8.3.3 CSI Node Driver Registrar

The CSI node driver registrar is a sidecar container that fetches driver information by issuing a `NodeGetInfo`) to the CSI Node service and registers the driver with the kubelet on that node. The registration is necessary because the kubelet is responsible for issuing CSI `NodeGetInfo`, `NodeStageVolume`, `NodePublishVolume` calls. By registering the CSI driver, the kubelet learns which Unix domain socket to issue the CSI calls on.

[2]Selected node annotation: `volume.kubernetes.io/selected-node: <node-name>`

2.8.4 Kubernetes CSI: An End-to-End Story

In this section, we aim to combine all the information by describing an end-to-end story for the CSI. We present the timeline of actions that take place under the hood in order to provision a volume dynamically.

The timeline of actions is the following:

1. The cluster administrator creates a StorageClass (in our case, the `Rok` storage class) that specifies the CSI plug-in name (`provisioner:rok.arrikto.com`):

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: rok
provisioner: rok.arrikto.com
parameters:
  rok/allow-auto-recovery: ”false”
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer
```

2. A user creates a PersistentVolumeClaim that requests a volume of at least 10 Gi with access mode `ReadWriteOnce` from the `Rok` storage class.

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: rok-pvc
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: rok
  volumeMode: Filesystem
```

   Since the `Rok` StorageClass has `volumeBindingMode: WaitForFirstConsumer`, the volume for the PVC will not be provisioned as long as a Pod requesting the PVC is not scheduled on a node. he PVC to be provisioned.

3. A user creates a Pod that uses the PVC:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - image: nginx
    name: container
    volumeMounts:
    - mountPath: /home/
      name: my-vol
  volumes:
  - name: my-vol
    persistentVolumeClaim:
      claimName: rok-pvc
```

4. The VolumeBinding plugin of the scheduler does not find any PV to match the PVC. It signals the driver to provision the volume dynamically: it annotates the PVC with the selected node annotation..

5. The external provisioner sidecar that runs along with the Rok CSI driver sees the annotation on the PVC and issues a `CreateVolume` call against the Rok CSI Controller service to provision the volume.
6. The Rok CSI controller provisions the volume and returns a successful response to the external provisioner.
7. The external provisioner creates a `PersistentVolume` object on the API Server and binds it (the PV) to the PVC.
8. The PersistentVolume controller completes the bidirectional binding of the PV and the PVC (by binding the PVC to the PV).

2.9 Logical Volume Management

Logical Volume Management enables combining multiple individual hard drives and disk partitions into a single volume group (VG). That volume group can then be subdivided into logical volumes (LV) or used as a single large volume. Regular file systems, such as EXT3 or EXT4, can then be created on a logical volume.

Logical volume manager (LVM) introduces an extra layer between the physical disks and the file system, allowing file systems to:

- Be resized and moved with ease and online without requiring a system-wide outage.
- Use discontinuous space on disk.
- Have meaningful names to volumes, rather than the usual cryptic device names.
- Span multiple physical disks.

The Logical Volume Management consists of the following conceptual layers:

- **Physical Volume**: Each Physical Volume can be a disk partition, whole disk, meta-device, or a loopback file.
- **Volume Group** (VG): A Volume Group gathers a collection of Logical Volumes and Physical Volumes into one administrative unit. A volume group is divided into fixed-size physical extents. VGs are made up of Physical Volumes, which, in turn, are made up of physical extents (PEs).

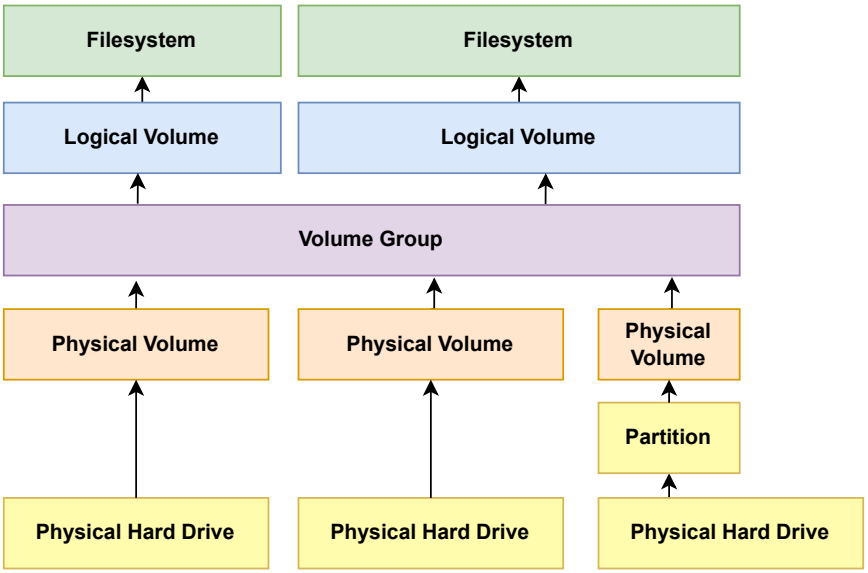


Figure 2.16: *The Logical Volume Management layers*

- **Logical Volume** (LV): A Logical Volume is the conceptual equivalent of a disk partition in a non-LVM system. Logical volumes are block devices that are created from the physical extents present in the same volume group.

2.10 Arrikto's Rok

Rok provides an enterprise data management layer that allows users to instantly snapshot their containers for local and offsite backup, take immutable, consistent snapshots of their apps and keep them in a backup store, e.g., Amazon S3. It allows users to snapshot, version, package, distribute, and clone their entire environment along with its data. It is natively integrated with Kubernetes as one of its supported platforms.

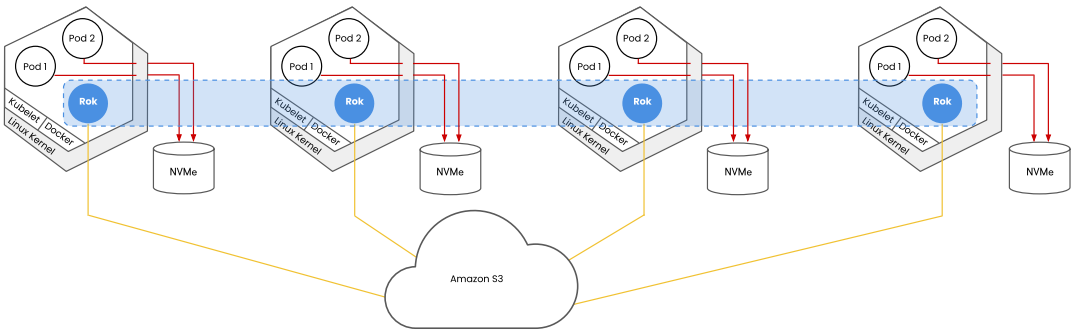


Figure 2.17: *A Rok Cluster*

2.10.1 The Rok Operator

Rok clusters ship with the *Rok operator*, a component that implements the Kubernetes *operator* pattern (see section 2.5) and manages the Rok cluster. Rok Operator watches the `rokCluster` Custom Resource and takes any actions required to bring the state of the cluster to the desired state.

The Rok operator is responsible –among others– for deploying the Rok CSI driver on the cluster and managing the Rok CSI data protection mechanism (Rok CSI Guards), which we will explain in the next chapter.

2.10.2 The Rok Storage System

The Rok storage system gathers the available local NVMe disks attached to a node and, using LVM, aggregates them into a single Volume Group, hereafter referred to as the "*Rok VG*". The Rok VG is the storage place where Rok dynamically provisions the requested volumes.

Rok's storage driver integrates with Kubernetes by implementing the Container Storage Interface. We will refer to the Rok's storage driver as the "*Rok CSI*". The Rok CSI follows the CSI plugins architecture and is deployed on the cluster as the following Kubernetes resources:

- A DaemonSet, running on every node the CSI Node plugin, hereafter referred to as the "*Rok CSI Node*". The Rok CSI Node handles the provisioning and the management of the local volumes on each node. The Pod of the Rok CSI Node consists of the CSI Node service container and the Node Driver Registrar sidecar.
- A StatefulSet, running the Controller plugin on an arbitrary node, hereafter referred to as the "Rok CSI Controller". The Pod of the Rok CSI Controller consists of the CSI Controller service container, the external attacher, and the external provisioner sidecars.

The architecture of the Rok CSI storage system is illustrated in Figure 2.18.

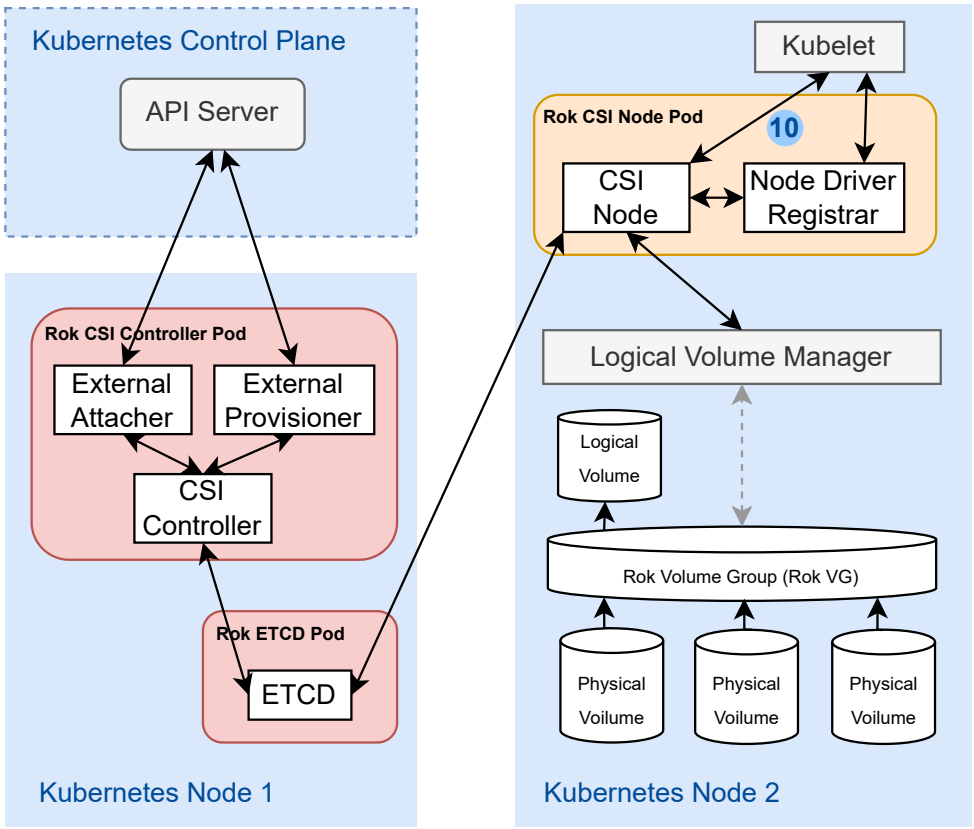


Figure 2.18: *The architecture of the Rok storage system*

3 Design

In this chapter, we describe the design and algorithms that discipline the operation of the Cluster Autoscaler and the Scheduler; we point out their shortcomings concerning local data storage and propose design enhancements to enable seamless scheduling and autoscaling with local persistent volumes.

3.1 Design Rationale & Goals

As explained in section 1.2, our goal is to extend the Scheduler and the Cluster Autoscaler so that they operate seamlessly with workloads that use volumes backed by local storage. More specifically, we aim to:

- Extend the Scheduler to consider the storage capacity of the nodes when scheduling Pods.
- Extend the Autoscaler to scale-down nodes where local volumes live, ensuring that Rok's mechanism snapshots the data before removing the node from the cluster.
- Extend the Autoscaler to check if the Pod's volumes can be placed on any other node (with regards to storage capacity) when evaluating (for a possible scale-down) if a Pod can be moved elsewhere.
- Extend the Autoscaler to consider the storage capacity of nodes, and, when scaling up, add a node with enough storage capacity.

- Extend the Autoscaler to not remove unready nodes in case local volumes live on these nodes.

3.2 Rok's Local Volume Mechanism

Local data on a node need a mechanism to be backed up if the node gets removed from the cluster; otherwise, the data will be permanently lost, and the user will not be able to recover them.

Rok provides a mechanism that enables the functionality of moving volumes around the nodes of a cluster. It leverages an external storage system, such as Amazon's S3, where it snapshots the local volumes and can restore them on a different node. Rok refers to moving a local volume to Amazon S3 as *"Unpinning"* and restoring the volume to a different node as *"Pinning"*. We describe this mechanism in greater depth in the following section.

3.2.1 Rok Volume Pinning and Unpinning

When a local volume is provisioned on a node, the corresponding `PersistentVolume` object on the API Server that represents the volume has node affinity on it (see section 2.2.5.7). In the case of local storage, the Rok CSI driver sets the node affinity of the PV to match only with the node where the local volume is provisioned.

Rok introduces the following terms:

- *Pinned PV*: A PV representing a node's local volume. This PV has node affinity to indicate that it is accessible only from that particular node.
- *Unpinned PV*: A PV representing a local volume migrated to S3. The PV has an empty node affinity to indicate that it is accessible from every cluster node.

A pinned PV can become unpinned with the process of "*unpinning*". An unpinned PV can become pinned with the process of "*pinning*". The process can be repeated multiple times, essentially allowing the volume to move around the cluster nodes as many times as needed.

The Rok CSI Controller implements the following mechanism for the unpinning of a PV:

1. Watches for nodes that are marked unschedulable.
2. Finds volumes on the unschedulable node that are not currently used by any Pods.
3. Starts the unpinning process of the unused PV: it takes snapshots of the volume on Amazon S3.
4. Removes the node affinity from the PV. Note that the `nodeAffinity` field of a PV is immutable, i.e., it is not allowed to change. To overcome this restriction, Rok deletes the PV and instantaneously recreates it.

Rok implements the following mechanism for the pinning of a PV:

1. The Scheduler schedules the Pod that references the unpinned PV (through a PVC).
2. The Kubernetes `attachDetach` controller creates a `VolumeAttachment` object to signal the external attacher to attach the volume on the node.
3. The external attacher sees the VolumeAttachment and issues a `ControllerPublishVolume` call to the Rok CSI controller.
4. The Rok CSI controller creates a logical volume on the Rok VG and restores the data of the PV from the Amazon S3 to the logical volume.
5. The Rok CSI controller sets the appropriate node affinity on the PV to indicate its only accessible from the node the volume was restored to.

3.2.2 Rok's Local Volume Protection Mechanism

The Kubernetes maintenance and upgrade tools rely on the `drain` operation (see 2.2.7). Essentially, before taking any actions to remove or upgrade a node in the cluster, the tools drain the node (`kubectl drain`) in order to mark the node unschedulable and safely evict all the Pods. The Cluster Autoscaler also uses the drain operation before removing a node.

Rok deploys a mechanism to facilitate the upgrades of a cluster and ensure that the nodes are not removed before Rok snapshots all their local volumes.

The mechanism leverages Pods with properly configured PodDisruptionBudgets to block their eviction. It relies on the fact that the drain operation fails as long as the eviction of a Pod fails. The mechanism works as follows:

1. The Rok Operator creates a `Deployment` resource *for each node* in the cluster. The Deployment of each node creates *exactly* one replica Pod with node affinity that matches only the specific node. Rok names these Pods "*CSI Guard Pod*", since they protect the node's local volumes.
2. The Rok Operator creates a `PodDisruptionBudget` object for each Rok CSI Guard Deployment. The PodDisruptionBudget demands at any time to exist at least one Rok CSI Guard Pod of the Deployment. This configuration causes any evictions of the Rok CSI Guard Pod to fail.
3. The drain operation marks the node unschedulable and starts evicting the Pods on the node. The eviction of the CSI Guard fails because of the configured PodDisruptionBudget.
4. The Rok Operator checks if the Rok CSI has unpinned all the volumes of the unschedulable node; if this condition holds, it removes the PodDisruptionBudget that corresponds to the CSI Guard of the node.
5. Since the PodDisruptionBudget does not exist anymore, the eviction of the Rok CSI Guard Pod finally succeeds, and the drain operation completes.

The mechanism is illustrated in Figure 3.1.

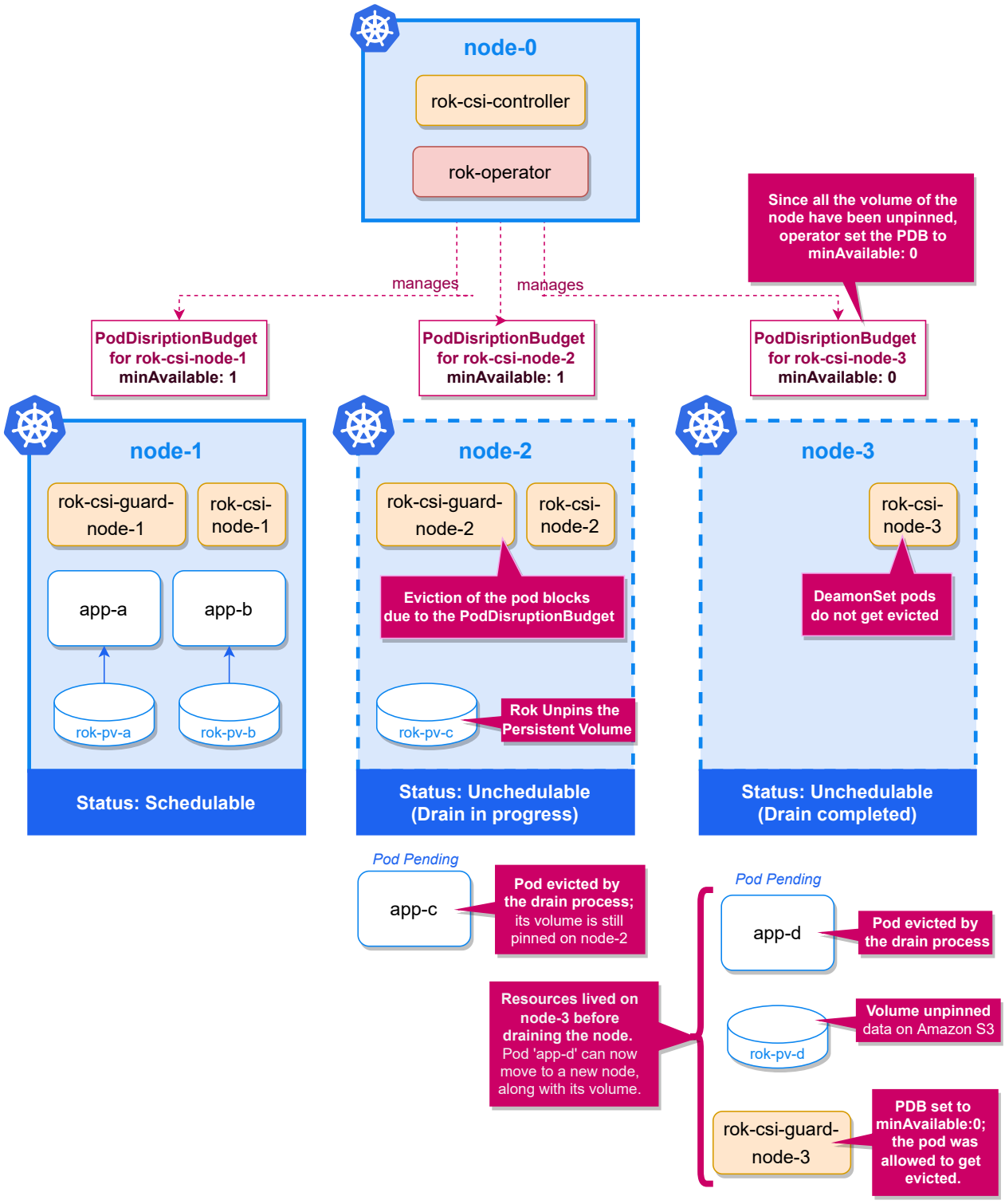


Figure 3.1: *Protecting Local Data with Rok CSI Guard Pods*

3.3 Kubernetes Scheduler

In this section, we will unveil the Kubernetes Scheduler's design. More specifically, we will expose the rationale of its `VolumeBinding` plugin and understand how it decides whether a node is appropriate to run a Pod or not, based on the volumes it requests. Moreover, we will identify the current design's problematic parts and propose enhancements to resolve these shortcomings.

3.3.1 The VolumeBinding Plugin

In this section, we describe the algorithms of the VolumeBinding plugin. Of particular interest in the context of this thesis are the `PreFilter`, `Filter`, and `PreBind` phases of the VolumeBinding plugin (see section 2.6.2), so we will omit the rest of the phases in our analysis.

PreFilter Phase

The `VolumeBinding` plugin's `PreFilter()` method takes as input a Pod to be scheduled and retrieves the PVCs it references. It splits them into three categories:

- **Bound claims**: PVCs that are already bound to a PV. If a PVC is bound, the Kubernetes –by definition– considers that the underlying storage has already been provisioned. Thus it is not taken into consideration when checking for the storage capacity.
- **Claims to bind**: Unbound PVCs with `WaitForFirstConsumer` (delayed) binding mode. The scheduler must bind these PVCs, either with an existing unbound PV or a new, dynamically provisioned PV. In the case of dynamic provisioning, since they specify delayed binding mode, it will occur after the scheduler selects a node for the Pod. The scheduler must select a node with enough storage capacity to accommodate them.
- **Unbound claims immediate**: PVCs with `Immediate` binding mode that are still unbound. PVCs that specify the `Immediate` binding mode must be bound to a PV by the PersistentVolume controller as soon as the PVC is created and before the

scheduler schedules the Pod. If unbound immediate claims exist, the scheduler will abort the current scheduling attempt for the Pod.

For the sake of completeness, we expose the algorithm of the `PreFilter()` method in Algorithm 2.

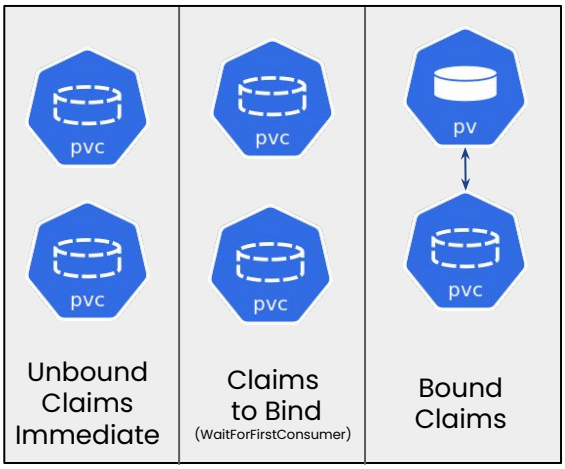


Figure 3.2: *VolumeBinding plugin's PreFilter method splits the PVCs a Pod references into 3 categories*

Filter phase

The `VolumeBinding` plugin's `Filter()` method takes as input a Pod to be scheduled, the bound and unbound claims the Pod references and a node. It checks if the claims of the Pod are compatible with the node. In particular:

- For each *bound claim*, it checks if the bound PV is accessible from the node, according to its node affinity. If a PV is not accessible, the Pod can not be scheduled on the examined node.
- For each *claim to bind*, it checks if there is any unbound PV that matches the claim requests. If such a PV exists, it creates a binding for the claim with the PV in its cache. The remaining claims to bind, for which the plugin did not manage to find a matching PV, are claims that must be dynamically provisioned (hereafter referred to as *"claims to provision"*).
- For each *claim to provision*, it checks if the `StorageClass` the PVC requests supports dynamic provisioning and if there is enough storage capacity accessible

from the node. If not, the Pod can not be scheduled on the node. At this step, the VolumeBinding plugin considers the storage capacity.

The current implementation of the scheduler checks if there is enough capacity for each claim to dynamically provision by calling the `hasEnough()` method with a single claim as input. It lists from the API Server all `CSIStorageCapacity` objects and checks if any of them matches the `StorageClass` of the PVC, is accessible from the examined node, and the reported capacity on the object is greater than the requested capacity of the PVC. If such a CSIStorageCapacity exists, the node has enough space for the given claim to be dynamically provisioned.

For the sake of completeness, we expose the algorithm of the `Filter()` method in Algorithm 3.

PreBind phase

As we explained, the VolumeBinding plugin's `Filter` phase tries to find PVs for the unbound PVCs of the Pod; otherwise, it tries to dynamically provision volumes for them.

During the `PreBind` phase, the VolumeBidning plugin executes the following actions:

1. For each PVC it found a matching PV, writes to the API Server the binding, i.e., it updates the PV to point to the PVC.

2. For each PVC that needs provisioning, it updates the PVC on the API Server with the "selected node annotation" [1] to signal the external provisioner that a volume for the PVC must be dynamically provisioned on a topology segment that is accessible from the selected node.

3. It then polls the API Server till all the unbound PVCs become fully bound. If the selected node annotation of any claim that needs provisioning is removed, the `PreBind` phase fails, and the scheduler cancels the current scheduling attempt.

4. The scheduler will retry scheduling the Pod later.

[1] The selected node annotation: `volume.kubernetes.io/selected-node`

We shall point out that removing the selected node annotation from a dynamically provisioned claim is a mechanism for the external provisioner to signal back to the scheduler that the volume provisioning failed, and the scheduler shall retry scheduling the Pod.

Here is an example of how the mechanism of the selected node removal proves useful:

1. The reported storage capacity information on the node is outdated: it indicates a reasonable quantity of storage, but the actual capacity is zero.
2. The scheduler decides to schedule the Pod on that node based on the outdated capacity report.
3. The scheduler decides to provision the PVC on the node and sets the selected node annotation.
4. The external provisioner tries to provision the volume but fails since the available storage capacity is zero. It responds to the external provisioner with a `ResourceExhausted` status code.
5. The external provisioner handles the `ResourceExhausted` status code and removes the selected node annotation from the PVC.
6. The VolumeBinding plugin fails, and the scheduler cancels the scheduling attempt.
7. The scheduler retries later, taking into consideration the possibly updated storage capacity information.

For the sake of completeness, we expose the algorithm of the `PreBind()` method in Algorithm 1.

3.3.2 Shortcomings & Proposed Extensions

According to the previous analysis of the algorithms, the current design of the Scheduler has the following limitations:

1. The VolumeBinding plugin's Filter method uses the `CSIStorageCapacity` objects to fetch information for the available storage capacity. This API object was introduced as an alpha feature in Kubernetes 1.19 and became beta on Kubernetes version 1.21. The major cloud providers follow the policy not to enable

alpha features on their services. As a result, the `CSIStorageCapacity` object is not available on clusters that run Kubernetes versions earlier than 1.21 on most cloud providers. That is a significant problem since many enterprises do not run the latest Kubernetes versions for stability reasons. In our case, our clients are running Kubernetes 1.19 and 1.20 clusters and, despite that, need to schedule Pods with local storage consideration.

2. The current design rationale of the VolumeBinding plugin's method does not consider the total capacity required for provisioning multiple PVCs of a single Pod. Instead, it checks if there is enough capacity for each PVC separately. That is a crucial problem: if a Pod references multiple unbound PVCs and there is not enough space for all of them, some of them will get provisioned, and the rest will fail. Then all future scheduling decisions will be limited by the already provisioned volumes, and the Pod will be stuck. Considering the total storage required for all the PVCs would minimize the possibility of bumping onto the previous problem (yet, some race conditions may cause this).

Since the upstream design comes with the aforementioned limitations, we propose extending the Kubernetes Scheduler and deploying our extended scheduler on the clusters. The proposed design consists of the following parts:

- Extend the Rok CSI Node component to report the available capacity of each node.
- Extend the Rok CSI Controller component to respond with an `ResourceExhausted` error status code to the `CreateVolume` request when the provisioning of a volume fails due to insufficient storage.
- Extend the Kubernetes Scheduler's VolumeBinding plugin to consider the total storage required by multiple PVCs and compare it against the reported free capacity of each node.
- Deploy the extended scheduler on the cluster.
- Deploy a webhook to mutate Pods to be scheduled by our custom scheduler.

We analyze each part of the design in the following paragraphs.

3.3.2.1 Extend Rok CSI Node

The `CSIStorageCapacity` objects for the capacity report can not be back-ported to previous Kubernetes versions; moreover, adding a similar object using a Custom Resource would be too much of a hassle. For these reasons, we decide to report the capacity of each node as an annotation on the corresponding `Node` object. We use the `rok.arrikto.com/capacity` key for the annotation. Hereafter we refer to it as the "*capacity annotation*".

The Rok CSI Node on each cluster node calculates the available storage and updates the capacity annotation. It issues commands to the Logical Volume Manager (LVM) of the node to fetch the free space of the Rok Volume Group and updates the `Node` object on the API Server.

An illustration of the mechanism is shown in Figure 3.3.

3.3.2.2 Extend Rok CSI Controller

We extend the Rok CSI Controller to return the `GRPCResourceExhausted` status code in response to the `CreateVolume` call of the external provisioner when the creation of a volume fails due to insufficient storage capacity.

3.3.2.3 Extend the VolumeBinding Plugin

We propose the extension of the `VolumeBidning` plugin's `Filter` phase as follows:

1. When checking the PVCs of the Pod that need provisioning (`checkVolumeProvisions()` method), select all the Rok PVCs [2] (hereafter referred to as the "*Rok claims to provision*") and check if there is enough capacity for the total storage they request.
2. Check if there is enough capacity for the Rok claims to dynamically provision, as follows:
   (a) Calculate the total capacity requested for Rok claims that need dynamic provisioning by summing their requests.
   (b) Check if the examined node has the Rok capacity [3] annotation.

[2] PVCs provisioned by the `rok.arrikto.com` provisioner.
[3] The Rok capacity annotation: `rok.arrikto.com/capacity`

(c) If the annotation *does not exist*, or if it exists but is not a valid integer number, the Rok claims can not be provisioned on the node. The absence of the annotation indicates that the Rok CSI driver is not running on the node; consequently, the claims can not be provisioned there.

(d) If the annotation *exists*, check if the reported available capacity is greater or equal to the total capacity the Rok claims request. If the condition holds, there is enough capacity, and the Rok claims of the Pod can be provisioned on the node. Otherwise, there is insufficient capacity; the Rok claims can not be provisioned, and the Pod can not be scheduled on the node.

3. Maintain backward compatibility by not modifying the handling of non-Rok PVCs, which may not be local. Our design splits the PVCs into local Rok PVCs and non-Rok PVCs; it only extends how the scheduler handles the Rok PVCs. PVCs provisioned by other storage providers will not be affected by our changes.

3.3.2.4 Deploy the Custom Rok Scheduler

The `kube-cheduler` runs by default on each cloud provider as part of the Kubernetes control plane and is the default scheduler used for scheduling Pods. The cloud providers hide the control plane from the end-user of their services, so there is no option to replace the instance of the running scheduler.

Due to this limitation, we deploy our scheduler on the cluster that runs the extended VolumeBinding plugin. Hereafter we refer to our custom scheduler as the "Rok Scheduler".

3.3.2.5 Deploy the Rok Scheduler Webhook

Since we deploy the Rok Scheduler without replacing the default Kubernetes Scheduler of the cluster, each Pod shall specify which scheduler shall schedule it by setting its `spec.schedulerName` field. If the field is not set, the default scheduler is used.

We do not want each user to manually set the name of the scheduler on the Pod; this would allow users to bypass the scheduling policy we set, is prone to errors, and is a tedious process. We need an automatic way to achieve this. The solution for automating the task is a mutating webhook.

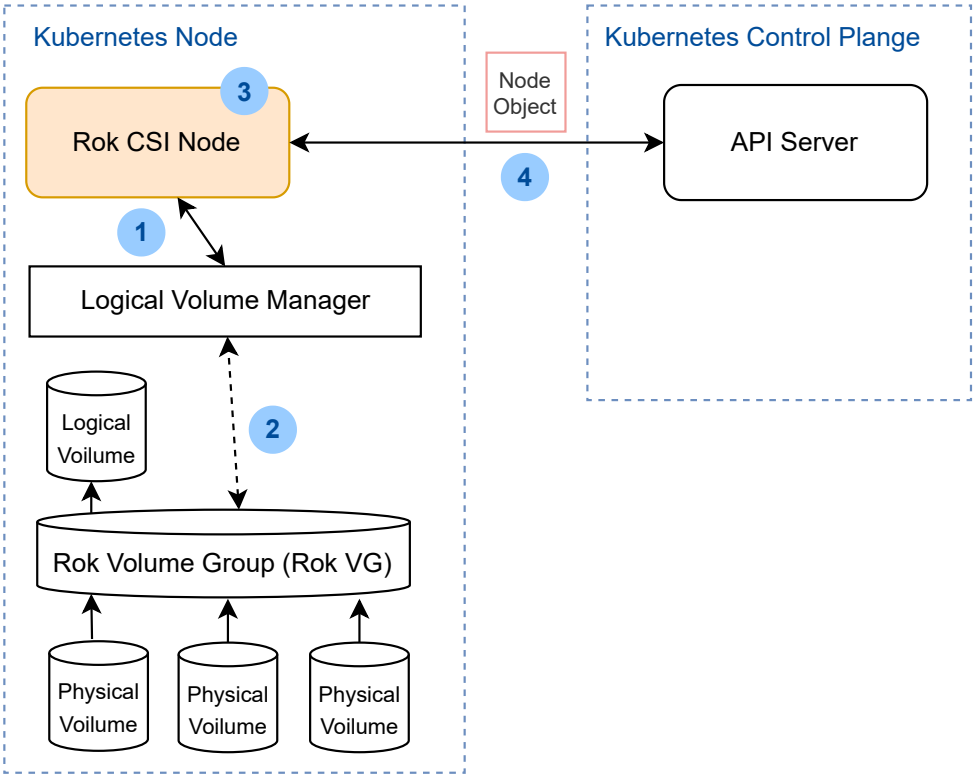
We deploy a mutating webhook on the cluster, hereafter referred to as the "Rok Scheduler webhook", which admits newly created Pods in specific namespaces of the cluster and adds the name of the Rok Scheduler on the Pod's spec. The proposed design is illustrated in Figure 3.4.

3.3.2.6 End-to-End Story of the Proposed Design

With the new design, this is the succession of interactions that will take place between the various components:

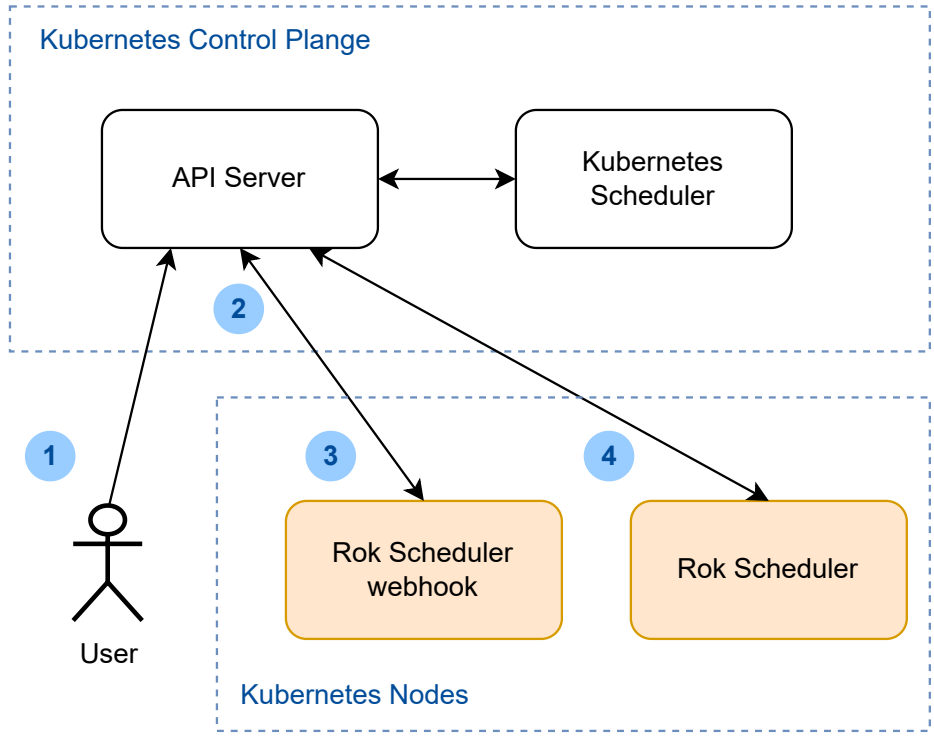
1. A user creates a Pod that references unbound Rok PVCs.
2. The Kubernetes MutatingAdmissionWebhook admission controller sends a request to the Rok Scheduler webhook to mutate the Pod.
3. The Rok Scheduler webhook adds the name of the Rok Scheduler on the spec of the Pod and responds to the API Server with the mutated Pod.
4. The Rok Scheduler sees the Pod's spec specifies `scedhulerName: rok-scheduler` and handles its scheduling.
5. The VolumeBinding plugin `Filter` method of the Rok Scheduler calculates the total capacity requested by the unbound Rok PVCs. It compares it against the value of the capacity annotation of the node it examines. It considers feasible nodes only these with enough capacity for the total requested storage.
6. The Rok Scheduler selects a node to place the Pod and adds the selected node annotation on the PVCs of the Pod to be provisioned.
7. The external provisioner of the Rok CSI sees the selected node annotation on the unbound Rok PVC and submits a `CreateVolume` GRPC call to the Rok CSI Controller component to provision a volume accessible from the selected node.
8. The Rok CSI Controller handles the request and submits a job for volume creation.
9. The Rok CSI Node component on the selected node executes the job. To provision a logical volume, it issues a `lvcreate` command to the LVM.
10. If there is not enough space on the Rok Volume Group, the `lvcreate` command will fail with `ResourceExhausted` status code, the job will fail, and the Rok CSI Node will report the reason and message of failure of the job to Rok.

11. The Rok CSI Controller component sees that the job failed and parses the error message; it determines that it is due to insufficient storage capacity and responds with `ResourceExhausted` code status to the external provisioner's `CreateVolume` call.
12. The external provisioner handles the `ResourceExhausted` error by removing the selected node annotation from the Rok PVC.
13. The `PreBind` method of the VolumeBinding plugin of the Rok Scheduler notices that the selected node annotation was removed from the PVC and aborts the current scheduling attempt.
14. The Rok Scheduler will attempt to scheduler the Pod again in the future.



1. The Rok CSI Node (running on each node) periodically issues command to the LVM to retrieve the free storage on the Rok VG.
2. The LVM will get the **free** and the **max** storage capacity of the Rok VG.
3. The Rok CSI Node will calculate the storage needed for metadata of the free and the max space. The reported free capacity will be equal to the available space on the VG reduced by the amount of space needed for metadata of the system.
4. The Rok CSI Node will update the Node object on the API Server with the `rok.arrikto.com/capacity` annotation to indicate the space available for volumes and the `rok.arrikto.com/max-instance-capacity` label to indicated the max capacity the node has.

Figure 3.3: *Rok CSI Node's capacity report mechanism*



1. A user creates a Pod.
2. The API Server is configured with a *MutatingWebhookConfiguration* to admit the Pod and send a request to the Rok Scheduler Webhook, so it contact the Rok Scheduler Webhook.
3. The Rok Scheduler webhook sets `spec.schedulerName: rok-scheduler` on the Pod and sends the mutated Pod to the API Server.
4. The Rok Scheduler watches that a Pod specifying its name in `spec.schedulerName` has been created, so it tries to schedule it. It sets the selected node name on the Pod's `spec.nodeName` field and writes it back to the API Server.

Figure 3.4: *Deployment of Rok Scheduler along with its webhook*

Algorithm 1: The scheduler's VolumeBinding plugin: PreBind() method

Input: `pod`: the Pod to be scheduled
`claimsToBind`: PVCs to bind
`claimsToProvision`: PVCs to provision
`node`: the selected node
Result: Binds unbound PVCs of the Pod with PVs.

1. Initiate the binding of `claimsToBind` by updating the API Server with the binding of the PV to its matching PVC.
2. Trigger the provisioning of `claimsToProvision` by updating the API Server object of each PVC with the selected node annotation.
3. Wait for PVCs to be completely bound by the PV controller:
   (a) For each claim in `claimsToProvision`:
      i. Check its selected node annotation.
      ii. **if** *the annotation was removed* **then** cancel the scheduling attempt.
   (b) For each claim in `claimsToProvision`, check if the bidirectional binding of the claim to the corresponding PV has completed.
   (c) **if** *the operation times out* **then** return error.

Algorithm 2: The scheduler's VolumeBinding plugin: PreFilter() method

Input: `pod`: the Pod to be scheduled

Result: Update the CycleState of the scheduler with the PVCs the Pod uses.

1. **if** *the Pod does not reference any PVCs* **then** return nil.
2. Otherwise, split the PVCs into 3 categories:
   (a) `boundClaims`: PVCs bound with a PV.
   (b) `claimsToBind`: Unbound PVCs with `WaitForFirstConsumer` (delayed) binding mode.
   (c) `unboundClaimsImmediate`: Unbound PVCs with `Immediate` binding mode.
3. **if** *`unboundClaimsImmediate` is non-empty* **then** return error; abort the scheduling attempt.
4. Store the `boundClaims` and `claimsToBind` in the `CycleState` of the scheduler.

Algorithm 3: The scheduler's VolumeBinding plugin: Filter() method

Input: `pod`: the Pod to be scheduled
`claimsToBind`: PVCs of the Pod to bind
`boundClaims`: PVCs of the Pod that are bound
`node`: the `Node` to check against.
Result: Checks if the PVCs of the Pod are compatible with the node

1. Call `FindPodVolumes(pod, state.boundClaims, state.claimsToBind, node)` to check if all of the Pod's PVCs can be satisfied by the node:
   (a) For each (bound) PVC in `boundClaims`:
      i. Get the PV it is bound to.
      ii. **if** *the PV's node affinities do not match the node* **then** the node is not feasible for the Pod.
   (b) For each (unbound) PVC in `claimsToBind`:
      i. **if** *the PVC has the selected node annotation and the selected node is not equal to the name of the examined Node* **then** return false (pod cannot be scheduled on the node.).
      ii. Try to find a matching PV for the PVC, to bind it to (by calling `findMatchingVolumes()`.
      iii. **if** *any matching PV was not found* **then** add the claim in list of volumes that must be provisioned (by calling `claimsToProvision()`).
2. Call `checkVolumeProvisions(claimsToProvision)`, to check the claims that need provisioning:
   (a) For each claim in `claimsToProvision` :
      i. Get the `StorageClass` that the PVC requests.
      ii. **if** *the `StorageClass` does not support dynamic provisioning* **then** return error.
      iii. Check if there is enough capacity on the node, by calling `hasEnoughCapacity()`.
      iv. **if** *there is not enough capacity* **then** return error.

Algorithm 4: The scheduler's VolumeBinding Plugin: hasEnough() method

Input: `claim`: the PVC to check
`node`: the `Node` to examine
Result: Checks if there is enough capacity for PVC on the node.

1. request ←Storage size (in bytes) the PVC requests.
2. Get the name of the provisioner from `StorageClass` the PVC requests.
3. Get the `CSIDriver` object with the same name as the provisioner.
4. **if** *no such `CSIDriver` object exists* **then** return true (capacity tracking is not enabled).
5. `capacities` ←List all the `CSIStorageCapacity` objects from the API Server.
6. For each `capacity` in `capacities`:
   (a) **if** *capacity.StorageClassName != storageClass.Name* **then** go to the next `capacity`.
   (b) **if** *request* $>$ *capacity.Capacity* **then** go to next `capacity`.
   (c) Check if node has access to the specific topology, by checking the `capacity.NodeTopology` against the node's labels.
   (d) **if** *node does not have access to the topology* **then** go to next `capacity`.
   (e) Return `true` (the node has access to enough capacity for the volume to be provisioned).
7. Return `false`, no CSIStorageCapacity with enough capacity for the PVC accessible from the node was found.

3.4 Kubernetes Cluster Autoscaler

In this section, we are going to expose the design of the Cluster Autoscaler (Autoscaler), describe its main principles of operation, identify its limitations and propose extensions that will enable its seamless operation with local persistent volumes.

3.4.1 Fundamental terms

Before we describe the algorithms of operations of the Autoscaler, it is essential to understand some fundamental structures and terminology it uses.

3.4.1.1 The Node Group Abstraction

The Autoscaler uses the abstraction of a "*node group*". A node group is not an actual Kubernetes resource but rather an abstraction for a group of nodes within a cluster. The Autoscaler expects that nodes found within a single node group have the same resources (CPU, memory, storage) and share several common properties such as labels and taints. However, they can still differentiate in some details, e.g., they may consist of more than one availability zone.

Each node group has the following important properties:

- `minSize`:minimum size of the node group.
- `maxSize`: maximum size of the node group.
- `targetSize`: the target size of the node group.

3.4.1.2 The `CloudProvider` Interface

The Autoscaler operates with various cloud providers, e.g., GCE, AWS. To achieve this, it specifies two important interfaces that each cloud provider that aims to integrate its services with the Autoscaler must implement:

- The `CloudProvider` interface: it contains configuration info and functions for interacting with the cloud provider.

- The `NodeGroup` interface: it contains configuration info and functions to control a node group.

The `NodeGroup` interface builds upon the node group abstraction. Each cloud provider may choose its interpretation of what is a node group on its service, as long as it conforms with the abstraction's definition.

For example, in the case of AWS EKS, the implementation of the `NodeGroup` interface maps each node group to an AWS Auto Scaling Group (ASG). An Auto Scaling group contains a collection of Amazon EC2 instances that are treated as a logical grouping for automatic scaling and management purposes. An EC2 instance is a virtual server in Amazon Web Services terminology. A cluster administrator configures the Auto Scaling groups of the EKS cluster and sets their `minSize`, `maxSize` accordingly. The Autoscaler interacts with the AWS cloud provider through the `CloudProvider` interface, which (the CloudProvider interface) lists the configured Auto Scaling groups and maps each of them to a node group. Only the `CloudProvider` interface knows about ASGs; the rest components of the Autoscaler are unaware of the underlying implementation and only see node groups.

3.4.1.3 The `ClusterSnapshot` Interface

The Autoscaler runs simulations on the cluster to make decisions. It takes a snapshot of the current cluster, adds or removes nodes in the snapshot, and simulates the scheduling decisions on the modified snapshot. A cluster snapshot contains a fixed view of the cluster's nodes and the Pods that run on each node. The `ClusterSnapshot` interface describes methods for taking a snapshot of the cluster nodes and their Pods.

Note that the cluster's PVCs and PVs are not contained in the snapshot. Instead, they are fetched from the API Server by the VolumeBinding plugin when the `PredicateChecker` checks if a Pod can be placed on a node. We will explain more about this later on.

3.4.1.4 The `PredicateChecker` interface

The Autoscaler defines the `PredicateChecker` interface, which offers methods to check whether all required predicates pass for a given Pod and node.

A Predicate is equivalent to a `Filter` plugin (see section 2.6.2) and it is used to filter out nodes that can not run a Pod.

These are the methods of the interface:

- `CheckPredicates()`: checks if the given Pod can be placed on the given node.
- `FitsAnyNode()`: checks if the given Pod can be placed on any of the given nodes.
- `FitsAnyNodeMatching()`: checks if the given Pod can be placed on any of the given nodes matching the provided function.

The Autoscaler implements this interface. The implementation is called `SchedulerBasedPredicateChecker` and leverages the Kubernetes Scheduler code. In particular, the Autoscaler imports the code of the Kubernetes Scheduler and constructs a list of predicates from the `Filter` plugins of the Scheduler. The Autoscaler uses the `SchedulerBasedPredicateChecker` in its simulations to determine whether a Pod can be placed on a node or not. The methods of the interface it implements follow this basic flow:

1. Create a new scheduler `CycleState`.
2. Run the `preFilter` method of all the plugins. Note that in the case of the VolumeBinding plugin, this step fetches the PVCs and PVs of the Pod from the API Server and stores them in the `CycleState`.
3. Runs all the `Filter` plugins to determine if the Pod can be placed on the node.

At this point, we shall highlight the fact that the Autoscaler imports the code of the Kubernetes Scheduler and uses the `Filter` plugins it provides in the `SchedulerBasedPredicateChecker` to run a simulation. However, it **never** interacts with the live instance of the Kubernetes Scheduler that runs on the cluster.

3.4.1.5 The `Estimator` & `Strategy` Interfaces

The Autoscaler specifies two interfaces that are used in the scale-up procedure:

- `Estimator`: interface for calculating the number of nodes of a given type needed to schedule Pods.

- `Strategy`: interface for selecting the best option to scale up.

The estimator currently used by the Autoscaler is `BinPackingNodeEstimator`. This estimator implements the First Fit Decreasing bin packing approximation algorithm.

3.4.1.6 Template Nodes

As we have explained, the Autoscaler assumes that every node in a node group will have the same resources (CPU, memory, storage) and labels. It constructs a template node for each node group to add it to the cluster snapshot and run its simulations. As the name suggests, a template node represents the details of a new node of the given node group. A template node is a `NodeInfo` struct and contains the details of a real `Node` object and information about the DaemonSet Pods that would run on the node if it was an actual node in the cluster. The Autoscaler tries to build a template node of a node group as follows:

1. First, look for a ready and schedulable node of the node group in the cluster and use it to generate the template.
2. If the previous step failed, look for a template in the Cluster Autoscaler's cache.
3. If the previous step failed, call the cloud provider's compiled plugin to generate a template for the given node group.
4. If the previous step failed, look for any unready or unschedulable node of the given node group in the cluster and use it to generate the template.

The constructed template nodes are *sanitized*: the sanitization is a mechanism that removes irrelevant or undesired details from the constructed node template, such as the name of the node, specific labels, etc.

The complete algorithm for template node creation is shown in Algorithm 5 and the

algorithm for the node template sanitization in Algorithm 6.

Algorithm 5: Cluster Autoscaler: GetNodeInfoForGroup() method

Input: node group: A NodeGroup struct.

Output: A template node for the Node Group (NodeInfo struct).

1. **if** *a ready and schedulable node of the node group exists in the cluster* **then**
    (a) Build the template from that node.
    (b) Store the template in the template's cache.
    (c) Return the template.

   end
2. **if** *a template node for the node group exists in the Autoscaler's cache* **then**
    (a) Return the cached template node.

   end
3. Call `TemplateNodeInfo()` of the `NodeGroup` interface to get the cloud provider defined template for the node group.
4. **if** *the `TemplateNodeInfo()` generated the template successfully* **then**
    (a) Return the template.

   end
5. **if** *an unready or unschedulable node of the node group exists in the cluster* **then**
    (a) Build the template from that node.
    (b) Return the template.

   end
6. Return error, the template node could not be constructed.

Algorithm 6: Cluster Autoscaler: sanitizeNodeInfo() method

Input: `node`: A template node (`NodeInfo` struct)

Output: A sanitized template node (`NodeInfo` struct).

1. `nodeName` ←"`template-node-for-<nodegroup-name>-<random-suf>`".
2. Set the `kubernetes.io/hostname` label of the node to `nodeName`
3. Remove the following taints of the node:
   - `ToBeDeletedByClusterAutoscaler`
   - `DeletionCandidateOfClusterAutoscaler`
   - any taints that indicate the node's condition, e.g, `node.kubernetes.io/not-ready`
   - taints starting with the `ignore-taint.cluster-autoscaler.kubernetes.io/` prefix.
4. Remove any taints of the node, as specified by the `-ignore-taints` flag of the Cluster Autoscaler.
5. Return the sanitized node.

3.4.1.7 Node Utilization

As for scale-down, the Autoscaler acts based on a metric called the utilization of a node; it calculates this metric using the *resource requests* of the Pods that run on the node instead of any actual (live) resource metrics.

Each Kubernetes node may have multiple resources, such as CPU, memory, etc. The cluster Autoscaler computes the utilization of every node in the cluster. For a given resource and node, the utilization is the ratio of the total resource requests from the Pods running on the node over the resource allocatable of the node. The utilization is a float number ranging from 0 to 1, where 1 indicates full utilization and 0 no utilization.

For example, the CPU utilization is defined as:

$$node\_cpu\_utilization = \frac{total\ CPU\ requests\ of\ Pods\ running\ on\ the\ node}{allocatable\ cpu\ of\ the\ node}$$

The steps for calculating the utilization of a node are shown in Algorithm 7.

Algorithm 7: Cluster Autoscaler: CalculateUtilization() method

Input: `node`: A Node API object

`pods`: the Pods running on the node

`resource`: a specific resource type, e.g, cpu, memory, etc

Output: The utilization of node for a given resource

1. Get the node allocatable resource from the `Node` object:
   `nodeAllocatable` ← `node.Status.Allocatable[resource]`.
2. **if** *nodeAllocatable == 0* **then return** 0.
3. Initialize: `daemonSetRequests` ←0, `podRequest` ←`0`.
4. Calculate the Pods' total resource requests. For each `pod` in `pods`:
   (a) `request` ←Calculate the resource request of the Pod by summing the `container.Resources.Requests[resourceName]` of each container of the `pod`.
   (b) **if** *Autoscaler is configured to ignore DaemonSet Pods in node utilization AND the Pod is a DaemonSet Pod* **then** `daemonSetRequests+= request`.
   (c) **if** *the Pod is long terminating* **then** continue to next Pod.
   (d) `podRequest += request`.
5. Calculate the utilization:

$$utilization = \frac{podRequest - daemonSetRequests}{nodeAllocatable - daemonSetRequests}$$

6. **return** *utilization*

3.4.2 The Main Loop

The Autoscaler runs continuously a loop, called the *main loop*, which executes two basic operations on the cluster:

- *Scale-up*: adding new nodes to cluster to help unschedulable Pods.
- *Scale-down*: removing unneeded nodes from a cluster.

The steps of the Autoscaler's main loop are shown in Algorithm 5.

Algorithm 8: Cluster Autoscaler: The main loop - RunOnce() method

1. `unschedulablePods` ←Select Pods that do not have `spec.nodeName` set.
2. `scheduledPods` ←Select Pods that have `spec.nodeName` set.
3. `allNodes` ←List all the nodes of the cluster, by calling `ObtainNodesList()`.
4. `readyNodes` ←List the Ready nodes of the cluster, by calling `ObtainNodesList()`.
5. `nodeGroups` ←List the registered node groups of the cluster from the cloud provider.
6. Take a snapshot of the cluster.
7. For every node group in `nodeGroups`, generate its template node.
8. For each node group in `nodeGroups` calculate the number of upcoming nodes (nodes that the Autoscaler has asked to be added but are not yet in the cluster) and add the same number of the node group's template nodes in the cluster snapshot.
9. Run a scheduling simulation with the current cluster snapshot to determine if any of the `unschedulablePods` can be scheduled on the upcoming nodes.
10. **if** *any Pod in `unschedulablePods` is considered as schedulable in the simulation* **then** disable the scale-down for the current loop.
11. `unschedulablePodsToHelp` ←Pods from `unschedulablePods` that remained unschedulable in the simulation.
12. **if** *`unschedulablePodsToHelp` is empty* **then** do not scale-up.
13. Else, try to scale-up, by calling `ScaleUp()`.
14. **if** *no scale-up was attempted* **then** proceed with the scale-down evaluation.

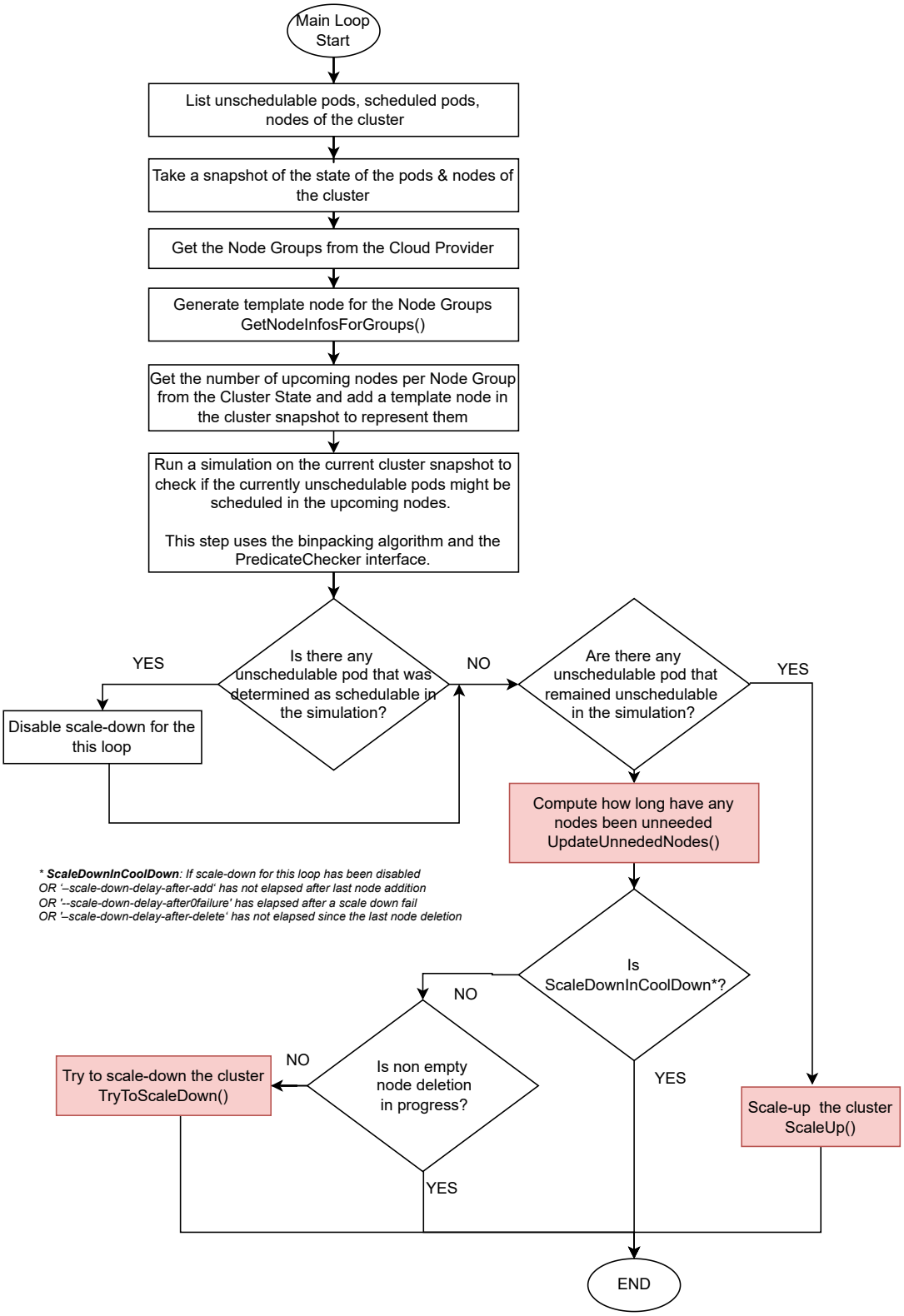


Figure 3.5: *Cluster Autoscaler:The main loop*

3.4.3 Scale-Down

The Autoscaler tries to scale down the cluster if it did not attempt any scale-up in the current run of the main loop. The scale-down procedure consists of two distinct procedures:

1. *Update unneeded nodes*: calculates which nodes have been unneeded and for how long.
2. *Try to scale down*: attempts to scale down the cluster by removing unneeded nodes.

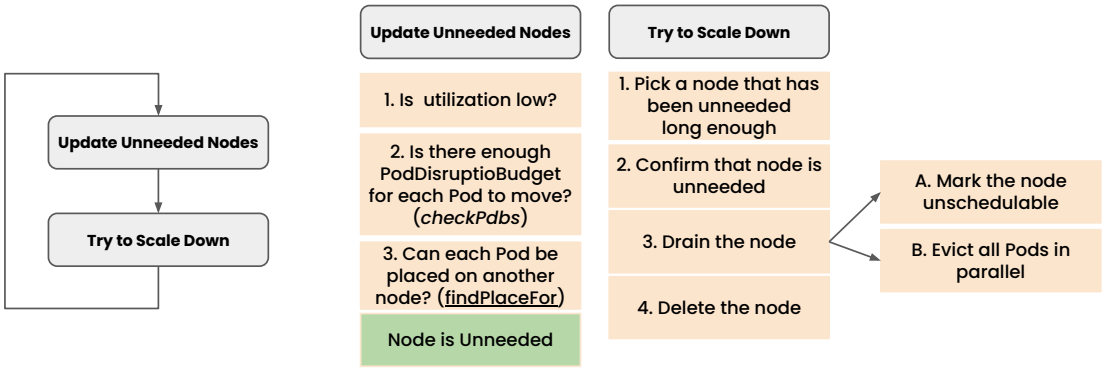


Figure 3.6: *Cluster Autoscaler: Scale-down procedure*

We will use symbolic names to refer to various parameters of the Autoscaler in our analysis, presented in Table 3.1.

3.4.3.1 Update Unneeded Nodes Procedure

The *update unneeded nodes* procedure calculates which nodes of the cluster have been unneeded and updates the Autoscaler's internal state with the duration they have been unneeded. The Autoscaler considers a node *unneeded* if it meets all the following criteria:

- It is *underutilized*, i.e., it has resource utilization below a specific threshold.
- The Pods that run on the node can be evicted (see section 2.2.6), i.e., their eviction is not blocked by any PodDisruptionBudgets.

Symbolic Name	Description
scanInterval	How often cluster is reevaluated for scale up or down.
scaleDownDelayAfterAdd	How long after scale up that scale down evaluation resumes. Defaults to 10 minutes. Configurable via the `–scale-down-delay-after-add` flag.
scaleDownDelayAfterDelete	How long after node deletion that scale down evaluation resumes. Defaults to scanInterval. Configureable via the `–scale-down-delay-after-delete` flag.
scaleDownDelayAfterFailure	How long after scale down failure that scale down evaluation resumes. Defaults to 10 minutes. Configurable via the `–scale-down-delay-after-failure` flag.
scaleDownUtilizationThreshold	Utilization threshold below which a node can be considered for scale down. Defaults to 0.5. Configurable via the `–scale-down-utilization-threshold` flag.
scaleDownUnneededTime	How long a ready node should be unneeded before it is eligible for scale down. Defaults to 10 minutes. Configurable via the `–scale-down-unneeded-time` flag.
scaleDownUnreadyTime	How long an unready node should be unneeded before it is eligible for scale down. Defaults to 20 minutes. Configurable via the `–scale-down-unready-time` flag.

Table 3.1: *Symbolic names for various parameters of the Autoscaler used in our analysis*

- The Pods that run on the node can be moved to a different cluster node.

If a node does not meet the criteria, it is considered *unremovable*.

To determine if an underutilized node is unneeded, the Autoscaler runs these steps:

1. Calculate the Pods that must be moved if it removes the node.
2. Check if any PodDisruptionBudgets block the eviction of any Pod. If so, the node is unremovable.
3. Call `FindPlaceFor()` to find place for the Pods on a different node. `FindPlaceFor()` uses the `SchedulerBasedPredicateChecker` interface to determine if the Pod can be placed on any other node. It checks if the Pod can fit a node due to other scheduling constraints (CPU, memory), as well as if the Pod's volumes can be accessed from the node.

The steps for calculating the unneeded nodes are shown in Algorithm 10.

3.4.3.2 Try to Scale Sown Procedure

If a *Ready* node remains unneeded for longer than `scaleDownUnneededTime`, or an *Unready* node remains unneeded for longer than `scaleDownUnreadyTime` (see Table 3.1 for the symbolic names) the Autoscaler will consider the node as a candidate for deletion.

The Autoscaler makes a distinction between non-empty and empty nodes:

- *Empty nodes*: nodes that run *only* DaemonSet Pods. The Autoscaler removes them in bulk
- *Non-empty nodes*: nodes that do not run only DaemonSet Pods. The Autoscaler removed them one by one to ensure that no Pods would be made unschedulable.

The algorithm for `TryToScaleDown()` is shown in Algorithm 11.

Node removal The node removal is executed as follows:

1. Add the `ToBeDeletedByClusterAutoscaler:NoSchedule` taint on the Node, essentially marking the node as unschedulable.

2. Start draining the node by evicting in parallel all the Pods of the node. If any Pod can not be evicted due to a configured PDB, retry until the `MaxPodEviction-Timeout` exceeds.
3. When all Pods are successfully evicted, ask the cloud provider to delete the node instance.

The algorithm for the node removal is shown in Algorithm 12.

Algorithm 9: Cluster Autoscaler: Scale-down evaluation procedure

1. Take a snapshot of the cluster Nodes, Pods, and PodDisruptionBudgets.
2. `allNodes` ← List all the `Node` objects from the API Server.
3. `scaleDownCandidates` ← Select nodes from `AllNodes` that belong to node groups that have not reached their minimum size.
4. `podDestinations` ← `AllNodes`; `podDestinations` represents the nodes that that may accept Pods in case a node is removed.
5. Call `UpdateUnneededNodes()` with `podDestinations`, and `scaleDownCandidates` as input, to calculate which nodes are unneeded an which ones are unremovable.
6. **if**
   the scale-down has been disabled for this loop
   OR `scaleDownDelayAfterDelete` *interval has not elapsed*
   OR `scaleDownDelayAfterAdd` *interval has not elapsed*
   OR `scaleDownDelayAfterFailure` *interval has not elapsed*
   then
   | Don't scale-down the cluster. Autoscaler Status: `ScaleDownInCooldown`
   else if *there is non empty node deletion in progress*
   then
   | Don't scale-down the cluster. Autoscaler Status: `ScaleDownInProgress`
   else Try to scale down the cluster..

Algorithm 10: Cluster Autoscaler: UpdateUnneededNodes() method

Input: `scaleDownCandidates`: a list of nodes that belong to node groups that have not reached their minimum size

Result: Update the state of the Autoscaler with information about which nodes are unneeded

1. For each node in `scaleDownCandidates`, call `checkNodeUtilization()`:
   (a) **if** *node has the "ToBeDeletedByClusterAutoscaler" taint* **then** it is currently deleted, continue to next node.
   (b) **if** *node has "cluster-autoscaler.kubernetes.io/scale-down-disabled: true" annotation* **then** continue to next node.
   (c) Calculate the utilization of the node.
   (d) **if** *utilization is above threshold* **then** continue to next node.
   (e) Add the node `currentlyUnneededNodes`.
2. `currentlyUnneededNonEmptyNodes` ←From `currentlyUnneededNodes` select nodes that are not empty, i.e., they do not run only DaemonSet Pods.
3. Call `findNodesToRemove(currentlyUnneededNonEmptyNodes)` to determine nodes that can be removed. For each node:
   (a) Get the Pods that are running on the node and for each Pod:
      i. **if** *the Pod has a Pod disruption budget that prevents its eviction* **then** the node is unremovable.
      ii. Call `findPlaceFor(pod)` to determine if it can be moved elsewhere.
      iii. **if** *the Pod can not be moved elsewhere* **then** the node is unremovable **else** the node can be removed, add it to `nodesToRemove`.
4. For each node in `nodesToRemove` update the state of the Autoscaler with the duration the node has been unneeded.

Algorithm 11: Cluster Autoscaler: TryToScaleDown() method

Input: A list of unneeded nodes, as computed by UpdateUnneded() method
Result: Scales-down the cluster

1. For each node in the unneeded nodes:
   (a) **if** *If the node has the* `cluster-autoscaler.kubernetes.io/scale-down-disabled` **then** mark the node unremovable, reason `ScaleDownDisabledAnnotation`, go to next node..
   (b) **if** *the node is* `Ready`*, and it has been underutilized for less than* `ScaleDownUnneededTime` **then** mark the node unremovable, reason `NotUnneededLongEnough`, continue to next node.
   (c) **if** *the node is* `Unready`*, and it has been underutilized for less than* `ScaleDownUnreadyTime` **then** mark the node unremovable, reason `NotUnreadyLongEnough`, continue to next node.
   (d) Get the NodeGroup the node belongs to, get its `minSize` and current `size`, the number of node deletions in progress for the node group (`deletionsInProgress`).
   (e) **if** *size* - *deletionsInProgress* $\leq$ *minSize* **then** mark the node unremovable, reason `NodeGroupMinSizeReached`, continue to next node.
2. `candidates` $\leftarrow$All the unneeded node that were not marked unremovable
3. From `candidates`, try to scale-down as many as possible empty nodes.
4. `nodesToRemove` $\leftarrow$From the remaining candidates find nodes to remove (call `FindNodesToRemove()`).
5. Pick a node from `nodesToRemove` and delete it (call `deleteNode()`) .

Algorithm 12: Cluster Autoscaler: deleteNode() method

Input: An unneeded Node of the Cluster to be deleted
Result: Deletes the Node from the cluster and the Cloud Provider

1. Add `ToBeDeletedByClusterAutoscaler:NoSchedule` taint on the Node to make the Node unschedulable.
2. Drain the node; For each Pod (except for the DaemonSet Pods), in parallel:
   (a) Send Eviction request
   (b) **while** *the Eviction fails and for duration up to* `MaxPodEvictionTimeout` **do** retry the Eviction.
   Note: `MaxPodEvictionTimeout` is a hard-coded value equal to 2 minutes.
3. **if** *any of the Pods was not evicted successfully* **then** return error.
4. **if** *the node has any annotation with prefix* `delay-deletion.cluster-autoscaler.kubernetes.io/` **then** wait for up to `nodeDeletionDelayTimeout` for the annotation to be removed.
5. Request from the Cloud Provider to delete the Node.
6. **if** *the Cloud Provider deletion fails* **then** return error.

3.4.4 Scale-Up

If the cluster has unschedulable Pods, the Autoscaler will try to help them by adding new nodes to the cluster (*scale-up*). A scale-up, essentially, is the increase of the target size of one or more node groups. If multiple node groups exist in the cluster, the cluster has to decide the following:

- Which node groups can help the unschedulable Pod run.
- How many nodes of the node group do the Pods need.
- If different node group scale-ups are feasible, which node group shall scale up.

As soon as the Autoscaler increases the target size of a node group, the cloud provider will spin up new node instances, the new nodes will join the Kubernetes cluster, and the scheduler will gradually scheduler the so far unschedulable Pods to the new nodes.

The full algorithm for the `ScaleUp()` method is shown in Algorithm 13.

Scale-up options To decide whether the scale-up of a node group would help the unschedulable Pod, the Autoscaler runs the (roughly) following steps:

1. Take a snapshot of the cluster.
2. Add a template node of the node group to the snapshot.
3. Run a simulation, using the `SchedulerBasedPredicateChecker`, whether the unschedulable Pod can be scheduled on the modified snapshot of the cluster.
4. If the simulation determines that the Pod can be scheduled on the modified snapshot, use the `BinPackingNodeEstimator` to calculate how many nodes of that node group are needed.

The option to scale up a specific node group with the number of needed nodes is referred to as a "*scale-up option*".

The complete algorithm to calculate a scale-up option is shown in Listing 14.

Scale-up strategy If multiple scale-up options, i.e., different node group scale-ups, can help the unschedulable Pods, the Autoscaler decides which one is best using the `Strategy` interface. There are various strategies, and the administrator can configure the Autoscaler to use a desired one, e.g., the least cost option, random strategy, etc.

Algorithm 13: Cluster Autoscaler: ScaleUp() method

Input: `pods`: the unschedulable Pods
`snapshot`: the cluster snapshot
Result: Adds extra nodes to accommodate the unschedulable Pods

1. Build Pod equivalence groups - each Pod equivalence group consists of Pods that are managed by the same controller (same UUID) and have the same spec and labels.
2. For each node group registered:
   (a) Get its target size.
   (b) If the target size $\geq$ max size, go to the next node group.
   (c) Create a template node for the node group.
   (d) Compute the expansion option for the node group, see `ComputeExpansionsOption()`.
   (e) If any unschedulable Pod can be helped by adding a new node of the node group, add the node group in the expansion options list.
3. If there are not any expansion options list, then do not trigger any scale-ups.
4. Else, from the expansions options select one, according to the configured expansion strategy.
5. Execute the selected scale up option: increase the target sizes of the corresponding node groups.

Algorithm 14: Cluster Autoscaler: ComputeExpansionsOption() algorithm

Input: `pods`: the unschedulable Pods
`snapshot`: the cluster snapshot
`template`: the template node of the node group
Result: Computes if the scale-up of the node group would help any of the unschedulable Pods.

1. For each Pod equivalence group:
   (a) Get the sample Pod of the Pod equivalence group.
   (b) Add the template node in the cluster snapshot.
   (c) Call the Predicate Checker to check if any of the unschedulable Pods can be scheduled in the new cluster snapshot.
   (d) If the sample Pod fits the new node in the cluster simulation, append all the equivalent Pods in the list of Pods that got helped (`options.Pods`).
2. Call the bin-packing estimator to estimate how many nodes of the node group would be needed to help all the equivalent Pods.
3. Return the `option`: a struct that indicates how many nodes of the node group are needed and which Pods would be helped.

3.4.5 Shortcomings & Proposed Extensions

In previous sections, we described the algorithms that govern the operations of the Autoscaler; we will now identify their shortcomings.

3.4.5.1 Scale-Down: Rok Volumes Can Be Migrated

When evaluating the scale-down of a node, the Autoscaler tries to find a place for the Pods that run on the node in other cluster nodes. To do so, it calls the `FindPlaceFor()` method, which in turn leverages the `PredicateChecker` interface methods to determine if a Pod fits a node. The `SchedulerBasedPredicateChecker` implementation of the interface runs the VolumeBinding plugin's `Filter()` method to check if the volumes of the Pod can be accessed from another node.

The PVs of the Rok storage class have node affinity that matches only with the node where the volume was provisioned. Since the node affinity of the volumes does not match any other in the cluster, the Autoscaler considers that the Pod and its volume can not be moved on a different node, thus, marking the current node as unremovable. The `SchedulerBasedPredicateChecker` does not know that the Rok volumes have a mechanism to snapshot and recover them on a different node [by unpinning them (snapshot + remove volume's node affinity, see section 3.2.1) and then pinning them (restoring the data) on another node].

We propose the extension of the Autoscaler to simulate the Rok volumes as unpinned (as if they do not have node affinity) when evaluating a scale-down (and only then; in other cases, the volumes are retaining their node affinity). With this extension, the Autoscaler will comprehend that the Rok volumes can move anywhere in the cluster, and it can remove the node safely.

3.4.5.2 Scale-Down: Coordinate With the Rok CSI Guard Mechanism

As part of the Rok volume protection mechanism (see section 3.2.2), we deploy a `Deployment` object for each node of the cluster, which creates a Pod per node (Rok CSI Guard) with strict node affinity that matches only the node it protects. The Autoscaler tries *to find place* to move this Pod. Since the Pod has strict node affinity that matches

only the current node, SchedulerBasedPredicateChecker assumes that the Pod cannot be moved to a different node. Because of that, the Autoscaler marks the node as unremovable. Of course, the Rok Operator will remove the Pod after the Autoscaler removes the node, but the Autoscaler is unaware of this fact.

Moreover, the Autoscaler checks the PDB of the Guard Pod. The PDB of the Guard Pod is configured to cause any evictions to fail. The Autoscaler notices that and assumes that the Guard Pod will not be able to get safely evicted, thus, marking the node unremovable. It is unaware that the Rok Operator will remove them as soon as the scale-down starts and the Rok CSI unpins all the local volumes of the node.

We propose the extension of the Autoscaler so that it does not try to find a place for the operator-managed ephemeral Guard Pods. Moreover, we extend the Autoscaler to ignore the PDB of the Guard Pod. Still, the Autoscaler will be aware that the Guard Pod exists, evicting it when it drains the node. This eviction will fail as long as the PDB exists and the Autoscaler will retry, delaying the deletion of the node.

To make things more obvious, here is the procedure that will take place with the new design:

1. The Autoscaler evaluates a node for removal:
   (a) It checks if the PDB allows the eviction of the Pods running on the node, but it ignores the PDB of the Guard Pod.
   (b) It tries to find a place for each Pod on a different node, but it ignores the Guard Pod.
2. The Autoscaler decides to remove the node.
3. The Autoscaler adds the deletion taint on the node, effectively marking it as unschedulable for Pods.
4. The Autoscaler sends eviction requests for each Pod to the API Server.
5. The eviction of all the Pods –except for the Guard Pod– succeeds.
6. The Autoscaler keeps retrying to evict the Guard Pod, but the API Server responds that the eviction is not allowed due to the configured PDB.
7. The Rok CSI Controller notices that the node is unschedulable and that no workload mounts the volumes, so it starts unpinning them.
8. The Rok CSI Controller finishes the unpinning of the PVs.

9. The Rok Operator removes the PodDisruptionBudget of the Guard Pod.
10. The Autoscaler's request to evict the Guard Pod succeeds since the PDB was removed.
11. The Autoscaler asks the cloud provider to delete the node.
12. The Rok Operator removes the Rok CSI Guard `Deployment` object that corresponds to the removed node.

Let us notice that the Autoscaler keeps retrying the eviction of the Guard Pod for up to 2 minutes. That duration is a hard-coded timeout that might not be enough in most cases. Taking a snapshot of the volume may last more than 2 minutes, depending on the size of the volume. It would be wise to use more sane values and allow the user to cluster's admin to configure the value when deploying the Autoscaler. To do so, we propose the extension of the Autoscaler with a flag to configure the max pod eviction timeout.

3.4.5.3 Scale-Down: Consider Storage Capacity

The Autoscaler shall check if the Rok volumes of a Pod can fit a node concerning their requested storage capacity when evaluating a scale-down. As we have explained, the Autoscaler used the `SchedulerBasedPredicateChecker` interface in order to check if a Pod fits a node, which –among others– calls the VolumeBinding plugin's `Filter()` method.

We propose the extension of the SchedulerBasedPredicateChecker's VolumeBinding plugin's `Filter` method: When evaluating if a Pod can be moved to a different node, check if there is enough available storage on the node to move the volumes.

Moreover, since the snapshotting and migration of a volume is a procedure that costs in terms of time, the Autoscaler shall not remove nodes with high storage utilization, similarly to how it handles the CPU and memory resources. To achieve this, we propose the extension of the Autoscaler with a new metric, called the (Rok) "*storage utilization*", defined as the ratio of the used storage over the max storage capacity of the node. The Autoscaler will compare this metric against a threshold configurable by the admin via a corresponding flag; if the storage utilization exceeds the threshold, the node will be considered unremovable.

3.4.5.4 Scale-Down: Do Not Remove Unready Nodes

A node that with status `Ready` can become `Unready` (or `NotReady`) if a system problem on the node arises. Common reasons include lack of resources on the node, a problem with the kubelet, an error related to kube-proxy, or a networking problem in general.

The Autoscaler removes any unneeded Unready node after the `scaleDownUnreadyTime` elapses. In the case of local volumes, we assume that the node will always be in good condition, with all the systems up and running and having network access, so Rok snapshots the local volumes to Amazon's S3 remote storage. If that does not hold, removing a node will probably cause any local data to be permanently lost.

The Autoscaler must not remove Unready nodes. The Unready nodes shall remain in the cluster so that an administrator takes action to recover them from the Unready state. We propose the extension of the Autoscaler with a flag to explicitly disable the scale-down for nodes in `Unready` state and consider them unremovable.

3.4.5.5 Scale-Up: Consider Storage Capacity

The Autoscaler does not know how much local storage is available when a new node is spanned up and added to the cluster. The template node it creates from a live node contains information only for the *currently* free storage (of the live node), reported on the capacity annotation by the storage driver (see the proposed scheduler design, section 3.3.2.1). We need a mechanism to know how much free storage a new node of a node group will have, and the Autoscaler shall consider it in its simulations.

Report max capacity Assuming that all the nodes in a node group have the same disk configuration and max storage capacity, we can use the storage driver to report what the new node's storage capacity would be on the `Node` objects. We propose the extension of the Rok CSI Node component to report the max capacity of a node as a label on the `Node` object. This label will be referred to as the "*max capacity label*" [4].

We use a label instead of an annotation because various cloud providers give the cluster admins the option to pass labels to the node group node templates the cloud provider plugin constructs. In case no live node for the node group exists, the Autoscaler will

[4]The *Rok max capacity label*: rok.arrikto.com/max-instance-capacity

construct the template from the cloud provider plugin and have the configured admin labels on it.

At this point, let us distinguish the two reported quantities:

- *capacity annotation*: the remaining free storage of a live node. The storage driver reports it, and the scheduler considers it when scheduling Pods.
- *max capacity label*: the max storage capacity of the live node. i.e., the total storage capacity. That is a time constant value that depends on the node's disks.

For example, a node might have 200 Gi total storage (reported on the max capacity label), and only 100 Gi out of them are free (reported on the capacity annotation).

Pass the max capacity information to the template For the Autoscaler to simulate the scheduling with capacity considerations, we will import in the SchedulerBasedPredicateChecker the extended VolumeBinding (see section 3.3.2.3).

The extended VolumeBinding plugin will look for the capacity of a template node on the capacity annotation and not on the label. Since the template will get the annotation from a live node, it will represent the currently free storage on the live node instead of the max storage capacity. We need to sanitize the value and set it to the actual max capacity. To do so, we propose the extension of the sanitization mechanism of the Autoscaler to copy the max capacity label's value to the capacity annotation. In this way, the template node's capacity annotation will indicate the max capacity (total) storage of the new node of the node group.

The sanitization mechanism shall set the capacity annotation to an infinitely large value if the max capacity label does not exist. This design choice offers the following advantages:

1. The Autoscaler treats the node as if it had infinite storage capacity and will add a live node of the node group (scale-up). If the decision was wrong (*false scale-up*), i.e., the added node does not have enough storage capacity for the Pod that triggered the scale-up, the scheduler of the cluster will not assign the Pod to the newly added node. As a result, the node will remain unneeded, and the Autoscaler will remove it after some time. The system will gradually fix the wrong decision.

2. The wrong node addition allows the Autoscaler to learn the actual max capacity of the node. The Rok CSI driver gets the chance to run on the node, and the Autoscaler generates a template node from the live node, which contains accurate information for the max available capacity reported by the Rok CSI driver.

Wait for Rok CSI to run If a Pod triggers a scale-up because of the storage it requests, it may take a reasonable time from the node addition till the Rok CSI driver starts running on the node. As long as the Rok CSI is not running on the node, the corresponding capacity annotation is not set on the `Node` object. The scheduler does not schedule the Pod on the node since the absence of the annotation indicates the storage is unavailable (see Section 3.3.1). The Autoscaler runsthe scheduling simulation and decides that the Pod does not fit the newly added node (since the live node has no capacity annotation), so it triggers a new scale-up.

To resolve this issue, the Autoscaler must wait for the Rok CSI driver to run on the newly added node. As long as the driver does not run on a new node (i.e., the node does not have the capacity label set), the Autoscaler shall replace the node with an *Unready* copy.

The Autoscaler treats Unready nodes as upcoming nodes (for a duration of up to 15 minutes): it replaces them with template nodes of the node group they belong to in its simulations. The template node will have the capacity annotation set as if the Rok CSI was running. The Autoscaler's simulation will assume that Pod will be scheduled on the node when the Rok CSI is ready and running and will not trigger any further scale-up.

We mentioned that the Autoscaler treats Unready node as upcoming for up to 15 minutes. That needs a bit of explanation. The Autoscaler gives the nodes a reasonable amount to become fully Ready; after this duration, it will stop replacing the Unready nodes with their template and will consider them unschedulable in the simulation. If any Pods that relied on the node becoming ready (in order to run there) still exist, they will now trigger another scale-up. Of course, in the case of Rok, 15 minutes are more than enough for the Rok CSI driver to become ready and start running.

4 Implementation

In this chapter, we describe the implementation of the proposed design changes and the technologies used.

4.1 Software Stack

The proposed design involves many parts that we had to extend:

- Kubernetes Scheduler, written in Go.
- Kubernetes Cluster Autoscaler, written in Go.
- Rok CSI driver, written in Python.

It also introduces a new component, the Rok Scheduler webhook, which we wrote in Go.

We build the components in a reproducible manner, using Docker containers for the target language of each component. To describe and automate the build process, we used Dockerfiles and Makefiles.

In order to deploy the components (Cluster Autoscaler, Rok Scheduler, Rok Scheduler webhook) on the cluster, we write YAML manifests that use the declarative API of Kubernetes to describe the necessary resources. To ease out the manifests management, we use the *Kustomize* tool. Kustomize is a configuration management solution that leverages layering to preserve the base settings of the applications and components by

overlaying declarative YAML artifacts (called patches) that selectively override default settings without changing the original files.

4.2 Extending the Rok CSI driver

In order to extend the Rok CSI driver's node component with the capacity reporting functionality, we introduce a new thread that periodically calculates the capacity and updates the capacity on the `Node` object on the API Server. The Python thread issues commands to the underlying Logical Volume Manager to fetch the Rok VG size. We introduce an argument `-capacity-poll-interval` to configure how long the thread waits before updating the storage capacity.

Listing 4.1: *The thread of Rok CSI driver that updates the available capacity*

```
class CSINodeStorageCapacityThread(threading.Thread):
    """Report free storage capacity."""

    def run(self):
        timeout = self.ctx.args.capacity_poll_interval
        while True:
            try:
                capacity, max_instance_capacity = self.get_storage_capacity()
                self.add_capacity_info(capacity, max_instance_capacity)
            except Exception as e:
                log.exception("Failed to update capacity info: %s", str(e))

            try:
                self.event_queue.get(timeout=timeout)
            except six.moves.queue.Empty:
                # Timeout
                pass
            else:
                # Signaled for exit
                return

    def add_capacity_info(self, capacity, max_capacity):
        """Add capacity information on the \co{Node} object.

        - Annotate the node with the currently available capacity
        - Add a label on the node with the max available capacity
        """
        node = self.ctx.args.node_name
        label_key = "rok.arrikto.com/max-instance-capacity"
        annotation_key = "rok.arrikto.com/capacity"

        patch = {
            "metadata": {
                "annotations": {
                    annotation_key: str(capacity)
                },
                "labels": {
                    label_key: str(max_capacity)
                }
            }
        }

        n = self.node_client.get(node)
        annotations = n.metadata.annotations or {}
        current_capacity = annotations.get(annotation_key, None)
        if current_capacity:
            try:
                current_capacity = int(current_capacity)
            except ValueError:
                log.warning("Annotation '%s' on node '%s' is not an integer"
                            " value: '%s'", annotation_key, node,
                            current_capacity)
                current_capacity = None

```

```
55          labels = n.metadata.labels or {}
56          current_max_capacity = labels.get(label_key, None)
57          if current_max_capacity:
58              try:
59                  current_max_capacity = int(current_max_capacity)
60              except ValueError:
61                  log.warning("Value of label '%s' on node '%s' is not an"
62                              " integer value: '%s'", label_key, node,
63                              current_max_capacity)
64                  current_max_capacity = None
65
66          if (((capacity != current_capacity)
67               or (max_capacity != current_max_capacity))):
68              log.info("Updating capacity information on Node '%s': %s", node,
69                       str(patch))
70              self.node_client.update(node, patch)
71              log.info("Successfully updated capacity information on Node '%s",
72                       node)
73          else:
74              log.info("Capacity information on Node '%s' is up to date", node)
```

4.3 Extending the Kubernetes Scheduler

The VolumeBinding plugin of the Kubernetes Scheduler imports and uses the `scheduling` package located at `pkg/controller/volume/scheduling/scheduler_binder.go`, in the Kubernetes repo [1]. We extend the package as follows:

- Introduce a `hasRokEnoughCapacity(claims []*v1.PersistentVolumeClaim, node *v1.Node)` method, which checks if there is enough capacity on the given `node` to provision all the specified Rok PVCs (`claims`). This method executes the following steps:
  1. Iterate through the given `claims`, and sum their storage requests in `totalRequestedCapacity`
  2. Check if the given node has `rok.arrikto.com/capacity` annotation.
  3. If the annotation does not exist, or if it exists but is not a valid int, returns `false`, which indicates the PVCs can not be provisioned on the examined node.
  4. If the annotation exists, fetch its value as `nodeCapacityInBytes`.
  5. If `totalRequestedCapacity` $\leq$ `nodeCapacityInBytes` return true, otherwise false.

  The implementation of the method is exposed in listing 4.2.
- Extend the `checkVolumeProvisions()` method of the `VolumeBinding` plugin to gather all the Rok PVCs, (PVCs provisioned by `rok.arrikto.com`), and pass

[1] `https://github.com/kubernetes/kubernetes`

them to `hasRokEnoughCapacity()`, in order to check if there is enough capacity for all of them to be provisioned on a selected node. The implementation is show at Listing 4.3.

- Treat the case that the `rok.arrikto.com/capacity` does not exist as `zero` capacity, i.e., the volumes can not be provisioned.

Listing 4.2: *Implementation of the hasRokEnoughCapacity() method*

```
// hasRokEnoughCapacity checks whether Rok has enough capacity left for all the
// provided volumes on the given node.
func (b *volumeBinder) hasRokEnoughCapacity(claims []*v1.PersistentVolumeClaim, node *v1.Node)↩
     (bool, error) {
    // Rok specific capacity tracking
    claimNames := []string{}

    // Sum the requested capacities
    totalRequestedCapacity := int64(0)
    for _, claim := range claims {
        pvcName := getPVCName(claim)
        quantity, ok := claim.Spec.Resources.Requests[v1.ResourceStorage]
        if !ok {
            // If !ok no capacity requested
            klog.V(4).Infof("PVC %q has no capacity request", pvcName)
            continue
        }
        sizeInBytes := quantity.Value()
        klog.V(4).Infof("PVC %q capacity request: %d bytes", pvcName, sizeInBytes)
        claimNames = append(claimNames, pvcName)
        // Check for overflow
        if (totalRequestedCapacity + sizeInBytes) < totalRequestedCapacity {
            klog.V(4).Infof("Overflow while calculating total requested capacity for Rok PVCs ↩
                 %q", strings.Join(claimNames, ", "))
            return false, fmt.Errorf("integer overflow")
        }
        totalRequestedCapacity += sizeInBytes
    }

    // Get the free space from the node API object
    nodeCapacity, ok := node.ObjectMeta.Annotations[RokCapacityAnnotation]
    if !ok {
        // No annotation found, treat this as no available capacity
        klog.V(4).Infof("Node %q has no '%s' annotation: Assuming Rok is not available on this↩
             node", node.ObjectMeta.Name, RokCapacityAnnotation)
        return false, nil
    }
    nodeCapacityInBytes, err := strconv.ParseInt(nodeCapacity, 10, 64)
    if err != nil {
        klog.V(4).Infof("Error while converting capacity string %q to bytes: invalid format", ↩
             nodeCapacity)
        return false, err
    }

    klog.V(4).Infof("Rok PVCs %q total capacity request: %d bytes, free storage capacity on ↩
         node %q: %d bytes", strings.Join(claimNames, ", "), totalRequestedCapacity, node.↩
         ObjectMeta.Name, nodeCapacityInBytes)
    if nodeCapacityInBytes >= totalRequestedCapacity {
        // Enough capacity found.
        klog.V(4).Infof("Sufficient free storage capacity for PVCs %q on node %q", strings.↩
             Join(claimNames, ", "), node.ObjectMeta.Name)
        return true, nil
    }
    klog.V(4).Infof("Insufficient free storage capacity for PVCs %q on node %q", strings.Join(↩
         claimNames, ", "), node.ObjectMeta.Name)
    return false, nil
}
```

Listing 4.3: *Extension of the checkVolumeProvisions() method*

```
func (b *volumeBinder) checkVolumeProvisions(pod *v1.Pod, claimsToProvision []*v1.↩
     PersistentVolumeClaim, node *v1.Node) (provisionSatisfied, sufficientStorage bool, ↩
     dynamicProvisions []*v1.PersistentVolumeClaim, err error) {
    podName := getPodName(pod)
    dynamicProvisions = []*v1.PersistentVolumeClaim{}
    // Rok PVCs
    rokClaims := []*v1.PersistentVolumeClaim{}
```

```
6
7       for _, claim := range claimsToProvision {
8               ...
9           class, err := b.classLister.Get(className)
10              ...
11          provisioner := class.Provisioner
12              ...
13          // PVCs provisioned by rok.arrikto.com
14          // Accumulate Rok PVCs in an array and skip hasEnoughCapacity
15          if provisioner == RokProvisioner {
16              klog.V(4).Infof("Unbound claim %q is provisioned by 'rok.arrikto.com'", pvcName)
17              rokClaims = append(rokClaims, claim)
18              continue
19          }
20
21          // Check storage capacity.
22          sufficient, err := b.hasEnoughCapacity(provisioner, claim, class, node)
23              ...
24          if !sufficient {
25              // hasEnoughCapacity logs an explanation.
26              return true, false, nil, nil
27          }
28
29          dynamicProvisions = append(dynamicProvisions, claim)
30      }
31
32      // Check Rok storage capacity.
33      sufficient, err := b.hasRokEnoughCapacity(rokClaims, node)
34      if err != nil {
35          return false, false, nil, err
36      }
37      if !sufficient {
38          // hasRokEnoughCapacity logs an explanation.
39          return true, false, nil, nil
40      }
41
42      // Apend Rok volumes to dynamicProvisions
43      dynamicProvisions = append(dynamicProvisions, rokClaims...)
44      klog.V(4).Infof("Provisioning for %d claims of pod %q that has no matching volumes on node↩
            %q ...", len(claimsToProvision), podName, node.Name)
45      return true, true, dynamicProvisions, nil
46  }
47
48  const (
49      // RokCapacityAnnotation is the key of the annotation that exposes the free capacity of ↩
            Rok on the node
50      RokCapacityAnnotation = "rok.arrikto.com/capacity"
51      // RokProvisioner is the name of the Rok provisioner
52      RokProvisioner = "rok.arrikto.com"
53  )
```

We compile the Rok Scheduler and build its Docker image using the Makefile the upstream project provides. We use YAML manifests and the Kustomize tool to deploy the Rok Scheduler as a `Deployment` along with any other RBAC resources it needs for its operation.

4.4 Implementing the Rok Scheduler Webhook

We implement the Rok Scheduler webhook that will mutate the Pods to use the Rok Scheduler, in a manner it can be reused and easily configured.

We expose the following configuration options:

- `-annotation-optout`: Annotation key that if present on the Pod, the Pod will not be mutated. The default value is `arrikto.com/skip-rok-scheduler-webhook`.

This parameters allows the user to skip the mutation of a Pod in the webhook server, even though the API server admitted that Pod for mutation. We will refer to it as the "*opt-out annotation*".

- `-namespaces-optin`: A comma-separated list of namespaces or namespaces globs. If a Pod matches against one of these namespaces it will get mutated. The default value is **"*"**, which matches against all namespaces. We will refer to it as the "*opt-in namespaces*".

- `-scheduler-name`: The name of the scheduler that will be set on the Pod. The default value is `rok-scheduler`.

For a complete list of arguments, see the `main()` method of the webhook in Listing 4.4.

The `Handle()` method of the webhook handles a single admission request as follows:

1. If the Pod it has the opt-out annotation, do not mutate it.
2. Check the namespace of the Pod against each namespace glob. If the namespace does not match any glob, do not mutate it.
3. In all other cases, mutate the Pod by adding the scheduler name on its `spec.SchedulerName` field.

For the full implementation of the method, see Listing 4.5.

We implement the Rok Scheduler Webhook using the `webhook` package of the controller-runtime library of Go. The Kubernetes controller-runtime is a set of go libraries for building controllers. For implementing the glob functionality of the `-namespaces-optin` flag, we used the `glob` module of Go.

Finally, in order for the Pods to be admitted and sent to the webhook, we instruct the API Server to do so by creating a `MutatingWebhookConfiguration` object (see Listing 4.6). The MutatingWebhookConfiguration we specify admits any newly created Pods in namespaces that match the specific namespace selector. The namespace selector matches against any namespaces that have the label `control-plane: kubeflow`. We chose to admit Pods only in this namespace since the workload we want to admit is created in that namespace, but of course, the Pods in any other namespace can be admitted. The MutatingWebhookConfiguration specifies that the API server contacts the

webhook server at the `/mutate` endpoint. It also specifies a `Fail` failure policy so that if the webhook crashes or stops responding, the creation of new Pods will fail. That is important to ensure every single Pod is admitted and mutated with the scheduler name.

Listing 4.4: *The main() method of the Rok Scheduler Webhook*

```
func main() {
    ...
    // Get command line parameters
    var port int
    var tlsDir, certName, keyName, optOutAnnotation, optInNamespaces, schedulerName string

    flag.IntVar(&port, "port", 443, "Webhook server port.")
    flag.StringVar(&tlsDir, "tls-dir", "/etc/webhook/certs", "Folder containing the X509 certs for the webhook.")
    flag.StringVar(&certName, "cert-name", "cert.pem", "File containing the x509 Certificate for HTTPS.")
    flag.StringVar(&keyName, "key-name", "key.pem", "File containing the x509 private key.")
    flag.StringVar(&optOutAnnotation, "annotation-optout", "arrikto.com/skip-rok-scheduler-webhook", "Annotation key that if present, the pod will not get mutated.")
    // Default namespaces-optin: "*" allows mutation in all namespaces
    flag.StringVar(&optInNamespaces, "namespaces-optin", "*", "A comma separated list of namespaces. If a pod matches against these namespaces it will get mutated. Globs can be provided.")
    flag.StringVar(&schedulerName, "scheduler-name", "rok-scheduler", "The name of the scheduler that will be set on the pod.")
    flag.Parse()
    ...
    // Compile globs from the namespaces-optin comma separated list
    optInNamespacesGlobs, err := compileGlobsFromCommaSeparatedList(optInNamespaces)
    if err != nil {
        logger.Error(err, "failed to parse the namespace globs list")
        os.Exit(1)
    }

    // Setup webhooks
    logger.Info("setting up webhook server") hookServer := mgr.GetWebhookServer()
    hookServer.Port = port
    hookServer.CertDir = tlsDir
    hookServer.CertName = certName
    hookServer.KeyName = keyName

    logger.Info("registering webhook to the webhook server")
    handler := &podHandler{
        Client:           mgr.GetClient(),
        optOutAnnotation: optOutAnnotation,
        optInNamespaces:  optInNamespacesGlobs,
        schedulerName:    schedulerName,
    }
    hookServer.Register("/mutate", &webhook.Admission{Handler: handler})

    logger.Info("starting manager")
    ...
}
```

Listing 4.5: *The Handle() method the Rok Scheduler Webhook*

```
/*
 * This file is part of Rok.
 *
 * Copyright © 2022 Arrikto Inc.  All Rights Reserved.
 */

package main

const LoggerName = "rok-scheduler-webhook"
const PodMutatedLabel = "rok-scheduler-webhook.arrikto.com/mutated"

// podHandler handles Pods
type podHandler struct {
    Client           client.Client
    decoder          *admission.Decoder
    optOutAnnotation string
    optInNamespaces  []glob.Glob
    schedulerName    string
}

// podHandler implements admission.DecoderInjector.
// A decoder will be automatically injected.
// InjectDecoder injects the decoder.
```

```go
func (p *podHandler) InjectDecoder(d *admission.Decoder) error {
    p.decoder = d
    return nil
}

func (p *podHandler) Handle(ctx context.Context, req admission.Request) admission.Response {
    pod := &corev1.Pod{}
    err := p.decoder.Decode(req, pod)
    if err != nil {
        return admission.Errored(http.StatusBadRequest, err)
    }
    // Set Pod namespace to AdmissionRequest namespace as Pod namespace might be empty.
    pod.Namespace = req.Namespace
    podName := pod.Name
    podGenerateName := pod.GenerateName

    if p.skipAdmission(pod) {
        log.Log.WithName(LoggerName).WithValues("namespace", pod.Namespace,
            "name", podName, "generateName", podGenerateName).Info("skipping pod mutation")
        return admission.Allowed("Admission skipped")
    }

    // Mutate the pod
    addSchedulerName(pod, p.schedulerName)
    addLabel(pod, PodMutatedLabel, "true")
    log.Log.WithName(LoggerName).WithValues("namespace", pod.Namespace,
        "name", podName, "generateName", podGenerateName).Info("adding scheduler name to pod",↩
             "schedulerName", p.schedulerName)

    marshaledPod, err := json.Marshal(pod)
    if err != nil {
        return admission.Errored(http.StatusInternalServerError, err)
    }

    // JSON patches are generated automatically
    return admission.PatchResponseFromRaw(req.Object.Raw, marshaledPod)
}

// skipAdmission checks if admission should be skipped for the specific Pod.
// This can happen because:
// - The Pod has an opt-out annotation
// - The Pod is not in the opt-in namespaces
func (p *podHandler) skipAdmission(pod *corev1.Pod) bool {
    podName := pod.Name
    podGenerateName := pod.GenerateName
    if hasAnnotationKey(pod, p.optOutAnnotation) {
        log.Log.WithName(LoggerName).WithValues("namespace", pod.Namespace, "name", podName,
            "generateName", podGenerateName).Info("pod has skip mutation annotation key", "↩
                 annotation", p.optOutAnnotation)
        return true

    }
    if matchesAnyGlob(p.optInNamespaces, pod.Namespace) {
        return false

    }
    log.Log.WithName(LoggerName).WithValues("namespace", pod.Namespace,
        "name", podName, "generateName", podGenerateName).Info("pod's namespace didn't match ↩
             any of the namespaces the webhook mutates")
    return true
}
```

Listing 4.6: *The Rok Scheduler's MutatingWebhookConfiguration*

```yaml
apiVersion: admissionregistration.k8s.io/v1
kind: MutatingWebhookConfiguration
metadata:
  name: rok-scheduler-webhook
webhooks:
- admissionReviewVersions:
  - v1
  clientConfig:
    caBundle: ""
    service:
      name: rok-scheduler-webhook
      namespace: rok-system
      path: /mutate
  failurePolicy: Fail
  name: kubeflow.rok-scheduler-webhook.arrikto.com
  namespaceSelector:
    matchLabels:
      control-plane: kubeflow
  reinvocationPolicy: IfNeeded
  rules:
  - apiGroups:
    - ""
    apiVersions:
```

```
    - v1
    operations:
    - CREATE
    resources:
    - pods
  sideEffects: None
```

To build the Rok Scheduler Webhook and its Docker image, we create a Dockerfile that instructs the docker to build the binary inside a container that has the required `Golang` dependencies.

We deploy the Rok Scheduler and the Rok Scheduler as `Deployment` resources (see 2.2.5.4). The manifests also specify other necessary resources, such as Roles, RoleBindings, ServiceAccounts, ConfigMaps.

4.5 Extending the Cluster Autoscaler

4.5.1 Scale-Down: Rok Volumes Can Be Migrated

As explained in the design proposal (see 3.4.5.1), we extend the Cluster Autoscaler to treat the local volumes of the Rok Storage class as unpinned, i.e., as if they have no affinities, when evaluating a possible scale-down. In all other cases, the local volumes shall be evaluated as is, pinned, i.e., having their existing node affinities.

To implement the design, we extend the CheckPredicates interface's method with an extra boolean argument, called `simulateUnpinnedVolumes`. We pass down information from the PredicateChecker methods to the VolumeBinding plugin's `checkBoundClaims()` method. The SchedulerBasedPredicateChecker creates a `cycleState` (see section 2.6.2 struct that the plugins it runs can use for storing data. We extend the `cycleState` with the same boolean `simulateUnpinnedVolumes` field to pass down to the VolumeBinding plugin information. The full flow of the information whether to simulate unpinned volumes or not is illustrated in Figure 4.1.

We extend the `checkBoundClaims()` method, so that if the Rok volumes are simulated as unpinned (`simulateUnpinnedVolumes` is set to `true`), it gathers all the Rok local volumes, and appends them to `claimsToProvision`, i.e, it treats them as if they were unbound volumes, in order to check if there is enough capacity (see (`checkVolumeProvision()`) for the volumes to be provisioned on the examined node. Of course, this approach only

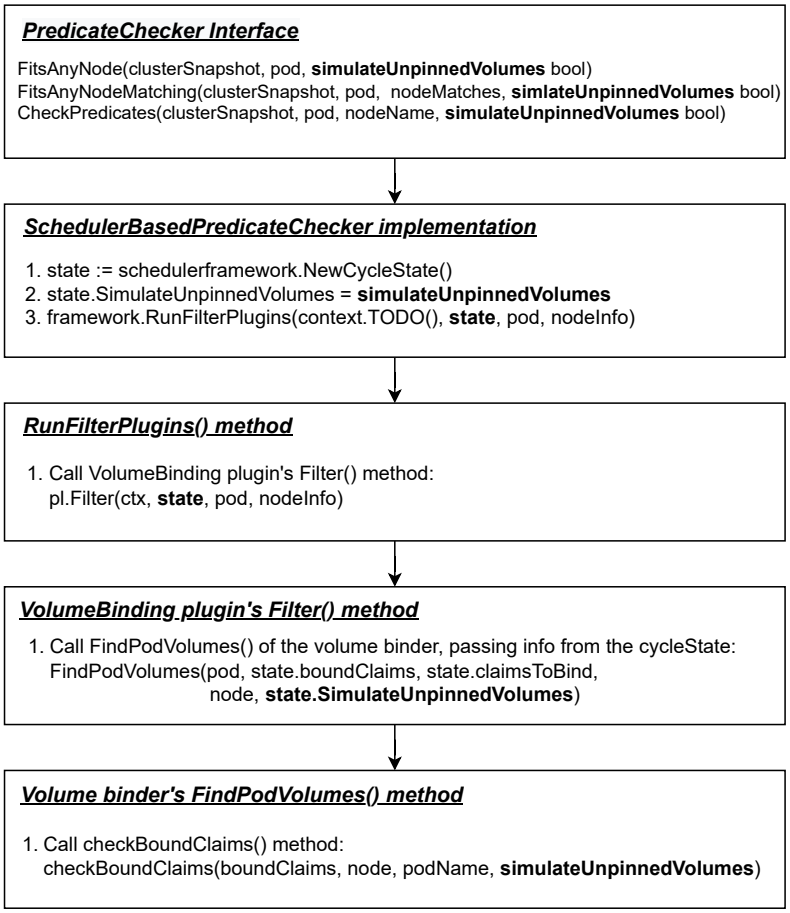


Figure 4.1: *The flow of simulateUnpinnedVolumes information*

checks if the Rok volumes of a Pod alone can be moved to a different node; it does not ensure that all the Pods of the node can fit on a different node with regards to their local storage requests. Listing 4.7 presents the code lines that extend the functionality of `checkBoundClaims`.

Listing 4.7: *Extending the logic of checkBoundClaims() when volumes are simulated as unpinned*

```
1  func (b *volumeBinder) FindPodVolumes(pod *v1.Pod, boundClaims, claimsToBind []*v1.↩
       PersistentVolumeClaim, node *v1.Node, simulateUnpinnedVolumes bool) (podVolumes *↩
       PodVolumes, reasons ConflictReasons, err error) {
2      ...
3      rokClaimsSimulatedUnpinned := []*v1.PersistentVolumeClaim{}
4      // Check PV node affinity on bound volumes
5      if len(boundClaims) > 0 {
6          // Bound PVCs provisioned by rok.arrikto.com that are simulated as unpinned,
7          // will skip checkBoundClaims check, and instead will be treated as volumes
8          // that need to be provisioned.
9          // Thus, we accumulate them in rokClaimsSimulatedUnpinned array,
10         // and then append then to claimsToProvision.
11         if simulateUnpinnedVolumes {
12             // boundClaimsFiltered is the boundClaims after filtering out Rok PVCs simulated ↩
                   as unpinned.
13             boundClaimsFiltered := []*v1.PersistentVolumeClaim{}
14
15             for _, claim := range boundClaims {
16                 if isRok, _ := b.isRokClaim(claim); isRok {
17                     rokClaimsSimulatedUnpinned = append(rokClaimsSimulatedUnpinned, claim)
18                     klog.V(4).Infof("Bound claim %q is provisioned by 'rok.arrikto.com' and ↩
```

```
                        simulated as unpinned: adding it to the list of volumes that need ↩
                        provisioning", getPVCName(claim))
19                  } else {
20                      boundClaimsFiltered = append(boundClaimsFiltered, claim)
21
22                  }
23              }
24              boundClaims = boundClaimsFiltered
25          }
26
27          boundVolumesSatisfied, boundPVsFound, err = b.checkBoundClaims(boundClaims, node, ↩
                podName)
28          if err != nil {
29              return
30          }
31      }
32
33      // Find matching volumes and node for unbound claims
34      if (len(claimsToBind) > 0) || (len(rokClaimsSimulatedUnpinned) > 0) {
35          ...
36          // Find matching volumes
37          if len(claimsToFindMatching) > 0 {
38              var unboundClaims []*v1.PersistentVolumeClaim
39              unboundVolumesSatisfied, staticBindings, unboundClaims, err = b.↩
                    findMatchingVolumes(pod, claimsToFindMatching, node)
40              if err != nil {
41                  return
42              }
43              claimsToProvision = append(claimsToProvision, unboundClaims...)
44          }
45
46          // Check for claims to provision. This is the first time where we potentially
47          // find out that storage is not sufficient for the node.
48          claimsToProvision = append(claimsToProvision, rokClaimsSimulatedUnpinned...)
49          if len(claimsToProvision) > 0 {
50              unboundVolumesSatisfied, sufficientStorage, dynamicProvisions, err = b.↩
                    checkVolumeProvisions(pod, claimsToProvision, node)
51          }
52      }
53      return
54  }
```

4.5.2 Scale-Down: Coordinate With the Rok CSI Guard Mechanism

To implement the design proposal (see section 3.4.5.2) we implement the following changes, according to the proposed design:

- Extend the `findPlaceFor()` method to ignore the Rok CSI Guard Pods and not try to find a place for them on a different node.
- Extend the `checkPdbs()` method to not check the PodDisruptionBudgets of the Rok CSI Guard Pods.
- We introduce a flag `-max-pod-eviction-time` so that the cluster admins can configure the maximum time Autoscaler tries to evict a Pod before giving up.

Listing 4.8: *Ignore Rok CSI Guard Pods and their PDBs in scale-down evaluation*

```
1  func findPlaceFor(removedNode string, pods []*apiv1.Pod, nodes map[string]bool,
2      clusterSnapshot ClusterSnapshot, predicateChecker PredicateChecker, oldHints map[string]↩
           string, newHints map[string]string, usageTracker *UsageTracker,
3      timestamp time.Time) error {
4      ...
5      for _, podptr := range pods {
6          ...
7          klog.V(5).Infof("Looking for place for %s/%s", pod.Namespace, pod.Name)
```

```
8
9           // Skip rok-csi-guard
10          csiGuardSelector, _ := labels.Parse("app=rok-csi-guard")
11          if csiGuardSelector.Matches(labels.Set(pod.Labels)) {
12              klog.V(2).Infof("Skipping findPlaceFor for %s pod", pod.Name)
13              continue
14          }
15          ...
16      return nil
17  }
18
19  func checkPdbs(pods []*apiv1.Pod, pdbs []*policyv1.PodDisruptionBudget) (*drain.BlockingPod, ↩
        error) {
20      for _, pdb := range pdbs {
21          // Ignore PDBs for rok-csi-guard pods
22          csiGuardSelector, err := labels.Parse("app=rok-csi-guard")
23          if err != nil {
24              return nil, err
25          }
26          if csiGuardSelector.Matches(labels.Set(pdb.Labels)) {
27              klog.V(2).Infof("Ignoring pod disruption budget  %s/%s", pdb.Namespace, pdb.Name)
28              continue
29          }
30          ...
31  }
```

4.5.3 Scale-Down: Consider Storage Capacity

We already covered in section 4.5.1 how we extended the Autoscaler to simulate the Rok volumes as unpinned when scaling down and also check if there is enough capacity for each Pod on a different node.

We also extend the Autoscaler to calculate storage utilization and take it into consideration when scaling down, by introducing a new flag "–scale-down-rok-storage-utilization-threshold" flag with default value "0.5" and a `CalculateUtilizationOfRokStorage()` method. This method fetches the values from the capacity annotation and the max capacity label of the `Node` object and calculates the storage utilization. Moreover, we extend the `checkNodeUtilization()` method of the Autoscaler to mark any nodes that have storage utilization over the storage threshold as unremovable. The implementation can is shown in Listings 4.9 and 4.10.

Listing 4.9: *Calculate Rok storage utilization*

```
1   // Based on CalculateUtilization(), see simulator/cluster.go
2   func CalculateUtilizationOfRokStorage(node *apiv1.Node) (utilInfo UtilizationInfo, err error) ↩
        {
3       capacity, foundCapacity := node.ObjectMeta.Annotations[rok.RokCapacityAnnotation]
4       maxCapacity, foundMaxCapacity := node.Labels[rok.RokInstanceCapacityLabel]
5       if !foundCapacity || !foundMaxCapacity {
6           klog.V(3).Infof("Rok doesn't run on node %s", node.Name)
7           return UtilizationInfo{RokStorageUtil: 0, ResourceName: rok.RokStorageResource, ↩
                Utilization: 0}, nil
8       }
9
10      // TODO: Restructure this code, used in many places
11      nodeCapacityInBytes, err := strconv.ParseInt(capacity, 10, 64)
12      if err != nil {
13          klog.V(4).Infof("Error while converting capacity string %q to bytes: invalid format", ↩
                capacity)
14          return UtilizationInfo{}, err
15      }
```

```
    // TODO: Restructure this code, used in many places
    nodeMaxCapacityInBytes, err := strconv.ParseInt(maxCapacity, 10, 64)
    if err != nil {
        klog.V(4).Infof("Error while converting max capacity string %q to bytes: invalid format", maxCapacity)
        return UtilizationInfo{}, err
    }

    util := (float64(nodeMaxCapacityInBytes) - float64(nodeCapacityInBytes)) / float64(nodeMaxCapacityInBytes)
    return UtilizationInfo{RokStorageUtil: util, ResourceName: rok.RokStorageResource, Utilization: util}, nil
}
```

Listing 4.10: *Mark nodes with high Rok storage utilization as unremovable*

```
func (sd *ScaleDown) checkNodeUtilization(timestamp time.Time, node *apiv1.Node, nodeInfo *schedulerframework.NodeInfo) (simulator.UnremovableReason, *simulator.UtilizationInfo) {
        ...
    if !sd.isNodeBelowUtilizationThreshold(node, utilInfo) {
        klog.V(4).Infof("Node %s is not suitable for removal - %s utilization too big (%f)", node.Name, utilInfo.ResourceName, utilInfo.Utilization)
        return simulator.NotUnderutilized, &utilInfo
    }

    klog.V(4).Infof("Node %s - %s utilization %f", node.Name, utilInfo.ResourceName, utilInfo.Utilization)

    // Get Rok storage utilization
    if rok.NodeHasRokStorageCapacity(node) {
        utilInfo, err := simulator.CalculateUtilizationOfRokStorage(node)
        if err != nil {
            klog.Warningf("Failed to calculate Rok storage utilization for %s: %v", node.Name, err)
        }
        klog.V(4).Infof("Node %s - %s utilization %f", node.Name, utilInfo.ResourceName, utilInfo.Utilization)

        if !sd.isNodeRokStorageBelowUtilizationThreshold(utilInfo) {
            klog.V(4).Infof("Node %s is not suitable for removal - %s utilization too big (%f)", node.Name, utilInfo.ResourceName, utilInfo.Utilization)
            return simulator.NotUnderutilized, &utilInfo
        }
    }
    return simulator.NoReason, &utilInfo
}
```

4.5.4 Scale-Down: Do Not Remove Unready Nodes

To implement the proposed design and configure the Autoscaler to not removed unready nodes, we extend the `-scale-down-unready-time` of the Autoscaler to accept negative values; if a negative value is provided, then the scale-down of unready nodes will be disabled.

4.5.5 Scale-Up: Consider Storage Capacity

Pass the max capacity information to the template To implement the scale-up design, we introduce a method `sanitizeRokStorageAnnotations()` that copies the value of the max capacity label of the `Node` object to its capacity annotation. If the label does

not exist, it set the capacity annotation to the max 64 bit number.We extend the `sanitizeTemplateNode()` method to call `sanitizeRokStorageAnnotations` as part of the sanitization process.

The implementation of this functionality is shown in Listings 4.11 and 4.12.

Listing 4.11: *sanitizeRokStorageAnnotations() method*

```
const (
    // RokCapacityAnnotation is the key of the annotation that exposes the free capacity of Rok on the node
    RokCapacityAnnotation = "rok.arrikto.com/capacity"
    // RokInstanceCapacityLabel is the key of the label that exposes the max capacity of Rok on the node
    RokInstanceCapacityLabel = "rok.arrikto.com/max-instance-capacity"
    // NodeMaxCapacity is the maximum possible storage capacity a node can provide.
    NodeMaxCapacity = math.MaxInt64
)

// SanitizeRokStorageAnnotations sets appropriate storage annotations for Rok
func sanitizeRokStorageAnnotations(labels map[string]string, annotations map[string]string, nodeName string) map[string]string {
    capacity := strconv.Itoa(NodeMaxCapacity)
    if val, found := labels[RokInstanceCapacityLabel]; found {
        klog.V(6).Infof("Found label '%s: %q' on template node %q", RokInstanceCapacityLabel, val, nodeName)
        capacity = val
    } else {
        klog.V(6).Infof("Label %q not found on template node %q: Setting capacity to maximum possible node capacity", RokInstanceCapacityLabel, nodeName)
    }
    klog.V(6).Infof("Setting annotation '%s: %q' on template node %q", RokCapacityAnnotation, capacity, nodeName)

    // We expect the annotations to be nil in case the template is created from a cloud provider plugin
    if annotations == nil {
        annotations = map[string]string{}
    }
    annotations[RokCapacityAnnotation] = capacity
    return annotations
}
```

Listing 4.12: *Extend sanitizeTemplateNode() to sanitize the Rok storage annotations*

```
 func sanitizeTemplateNode(node *apiv1.Node, nodeGroup string, ignoredTaints taints.TaintKeySet) (*apiv1.Node, errors.AutoscalerError) {
    newNode := node.DeepCopy()
    nodeName := fmt.Sprintf("template-node-for-%s-%d", nodeGroup, rand.Int63())
    ...
    newNode.Name = nodeName
    newNode.Spec.Taints = taints.SanitizeTaints(newNode.Spec.Taints, ignoredTaints)
    newNode.ObjectMeta.Annotations = sanitizeRokStorageAnnotations(newNode.Labels, newNode.ObjectMeta.Annotations, newNode.Name)
    return newNode, nil
}
```

Wait for Rok CSI to run We implement this design change by introducing a `FilterOutNodesWithUnreadyCSI()` method to check if a node has the capacity annotation set. If not, it is implied that the Rok CSI driver is not running on the node, and it replaces the node with an unready copy. We extend the `obtainNodeLists()` method, to call the `FilterOutNodesWithUnreadyCSI`.

Listings 4.13 and 4.14 present the code lines that implement this functionality.

Listing 4.13: *FilterOutNodesWithUnreadyCSI() method*

```
const (
    // RokCapacityAnnotation is the key of the annotation that exposes the free capacity of Rok on the node
    RokCapacityAnnotation = "rok.arrikto.com/capacity"
 )

func FilterOutNodesWithUnreadyCSI(allNodes, readyNodes []*apiv1.Node) ([]*apiv1.Node, []*apiv1.Node) {
    newAllNodes := make([]*apiv1.Node, 0)
    newReadyNodes := make([]*apiv1.Node, 0)
    nodesWithUnreadyCSI := make(map[string]*apiv1.Node)
    for _, node := range readyNodes {
        hasCapacityAnnotation := false
        if node.Annotations != nil {
            _, hasCapacityAnnotation = node.Annotations[RokCapacityAnnotation]
        }

        if !hasCapacityAnnotation {
            klog.V(3).Infof("Overriding status of node %v, which seems to have unready CSI", node.Name)
            nodesWithUnreadyCSI[node.Name] = kubernetes.GetUnreadyNodeCopy(node)
        } else {
            newReadyNodes = append(newReadyNodes, node)
        }
    }

    // Override any node with unready CSI with its "unready" copy
    for _, node := range allNodes {
        if newNode, found := nodesWithUnreadyCSI[node.Name]; found {
            newAllNodes = append(newAllNodes, newNode)
        } else {
            newAllNodes = append(newAllNodes, node)
        }
    }
    return newAllNodes, newReadyNodes
}
```

Listing 4.14: *Extend ObtainNodesLit() method to return nodes with unready Rok CSI*

```
func (a *StaticAutoscaler) obtainNodeLists(cp cloudprovider.CloudProvider) ([]*apiv1.Node, []*apiv1.Node, errors.AutoscalerError) {
    allNodes, err := a.AllNodeLister().List()
    readyNodes, err := a.ReadyNodeLister().List()
        ...
    allNodes, readyNodes = gpu.FilterOutNodesWithUnreadyGpus(cp.GPULabel(), allNodes, readyNodes)
    allNodes, readyNodes = taints.FilterOutNodesWithIgnoredTaints(a.ignoredTaints, allNodes, readyNodes)
    allNodes, readyNodes = FilterOutNodesWithUnreadyCSI(allNodes, readyNodes)
    return allNodes, readyNodes, nil
}
```

5 Conclusion

Our journey has finally reached its end. In this chapter, we will restate our contributions and summarize what our mechanism offers. Finally, we will close this thesis by mentioning future work that can be done to enrich our mechanism and bring it to its full potential.

5.1 Concluding Remarks

All in all, the primary goal of this thesis was to implement a design that would enable seamless cluster autoscaling and scheduling with local persistent storage. Not only did we achieve this goal, but our implementation was successfully deployed to large production clusters of enterprises that requested the feature.

Although the design we implemented is coupled with the Rok software –since it provides an efficient mechanism for migrating local volumes around a cluster–, the concepts and the design can be generalized to work with any other local storage system. Our long-term goal, which extends beyond the context of this thesis, is to generalize the design and push it upstream. That is a process that we started to be involved in; we attend the meetings of the Kubernetes Storage [1] and Autoscaling [2] Special Interest Groups, interacted with them on GitHub and plan to contribute the whole design upstream actively. At the moment this text is written, we have a few first Pull Requests

[1] https://github.com/kubernetes/community/blob/master/sig-storage/README.md
[2] https://github.com/kubernetes/community/blob/master/sig-autoscaling/README.md

merged [3] [4].

5.2 Future Work

So far, we have implemented various enhancements for the Kubernetes Scheduler and the Cluster Autoscaler, but there is always room for improvement. Since this is an iterative process, in next iterations, we would like to offer these enhancements:

- Extend the Scheduler to reserve the storage (in the `Reserve` phase of the scheduling cycle) when scheduling a Pod to prevent race conditions.
- Extend the Cluster Autoscaler to consider the storage needed for the PVCs of multiple Pods when scaling down. The current design only checks if a single Pod's PVs can fit a node, but not if the PVs of multiple Pods fit a node. Although a wrong decision to scale down will be reverted by a subsequent scale-up, taking the decision would be much more effective.
- Extend the current implementation of the `Estimator` interface, i.e., the `BinPackingEstimator`, to consider the storage and calculate how many nodes are needed for the storage requests of multiple Pods of a StatefulSet. The current design adds nodes one by one till all the Pods get the requested storage. It would be much more efficient to know the number of nodes needed beforehand and add them to the cluster all at once.

Finally, as we already mentioned, our high-priority goal is to merge this work upstream.

[3] https://github.com/kubernetes/autoscaler/pull/4877
[4] https://github.com/kubernetes/autoscaler/pull/4842

Bibliography

[1] Arrikto. Arrikto mlops | scalable machine learning models, delivered. https://www.arrikto.com/. (Accessed on 06/29/2022).

[2] The Kubernetes Authors. Confirm your account recovery settings. https://github.com/kubernetes-sigs/controller-runtime. (Accessed on 07/08/2022).

[3] The Kubernetes Authors. The kubernetes api | kubernetes. https://kubernetes.io/docs/concepts/overview/kubernetes-api/. (Accessed on 06/15/2022).

[4] The Kubernetes Authors. Kubernetes api reference docs. https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.24/#podcondition-v1-core. (Accessed on 06/15/2022).

[5] The Kubernetes Authors. kubernetes-sigs/controller-runtime: Repo for the controller-runtime subproject of kubebuilder (sig-apimachinery). https://github.com/kubernetes-sigs/controller-runtime. (Accessed on 07/08/2022).

[6] The Kubernetes Authors. Pod lifecycle | kubernetes. https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/. (Accessed on 06/15/2022).

[7] The Kubernetes Authors. Storage capacity tracking reaches ga in kubernetes 1.24 | kubernetes. https://kubernetes.io/blog/2022/05/06/storage-capacity-ga/. [Online; Accessed on 06/11/2022].

[8] The Kubernetes Authors. A guide to kubernetes admission controllers | kubernetes. https://kubernetes.io/blog/2019/03/21/a-guide-to-kubernetes-admission-controllers/, 2019. (Accessed on 06/15/2022).

[9] The Kubernetes Authors. Configure multiple schedulers | kubernetes. https://kubernetes.io/docs/tasks/extend-kubernetes/configure-multiple-schedulers/, December 2021. [Online; Accessed on 06/11/2022].

[10] The Kubernetes Authors. kubernetes-csi/external-provisioner: Sidecar container that watches kubernetes PersistentVolumeClaim objects. https://github.com/kubernetes-csi/external-provisioner, 2021. [Online; Accessed on 06/11/2022].

[11] The Kubernetes Authors. CSI attacher. https://github.com/kubernetes-csi/external-attacher/blob/master/README.md, 2022. [Online; Accessed 26/4/2022].

[12] The Kubernetes Authors. Dynamic Admission Control. https://kubernetes.io/docs/reference/Access-authn-authz/extensible-admission-controllers, 2022. [Online; Accessed 26/4/2022].

[13] The Kubernetes Authors. Kubernetes Architecture. https://www.aquasec.com/cloud-native-academy/kubernetes-101/kubernetes-architecture/, 2022. [Online; Accessed 27/4/2022].

[14] The Kubernetes Authors. Kubernetes Architecture. https://kubernetes.io/docs/concepts/overview/components/, 2022. [Online; Accessed 27/4/2022].

[15] The Kubernetes Authors. Kubernetes CSI Documentation. https://kubernetes-csi.github.io/docs/, 2022. [Online; Accessed 26/4/2022].

[16] The Kubernetes Authors. Storage capacity | kubernetes. https://kubernetes.io/docs/concepts/storage/storage-capacity/, March 2022. [Online; Accessed on 06/11/2022].

[17] The Kubernetes Authors. What is Kubernetes? https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/, 2022. [Online; Accessed 26/4/2022].

[18] The Stork Authors. libopenstorage/stork: Stork - storage orchestration runtime for kubernetes. https://github.com/libopenstorage/stork, 2022. [Online; Accessed on 06/11/2022].

[19] The Kubernetes Authos. Deploying a csi driver on kubernetes - kubernetes csi developer documentation. https://kubernetes-csi.github.io/docs/deploying.html. (Accessed on 06/29/2022).

[20] Roman Bessonov. Microservices and containerization: 4 things every IT manager needs to know. https://nordcloud.com/microservices-and-containerisation-four-things-to-know, 2018. [Online; Accessed 19/5/2022].

[21] Aniket Bhattacharyea. Debugging your kubernetes nodes in the 'not ready' state | nodenotready. https://www.containiq.com/post/debugging-kubernetes-nodes-in-not-ready-state. (Accessed on 06/15/2022).

[22] Brendan Burns, Joe Beda, and Kelsey Hightower. *Kubernetes: Up and Running*. O'Reilly Media, Inc., 06 2022.

[23] Wikipedia Contributors. Bin packing problem - wikipedia. https://en.wikipedia.org/wiki/Bin_packing_problem. [Online; Accessed on 06/11/2022].

[24] Wikipedia Contributors. grpc - wikipedia. https://en.wikipedia.org/wiki/GRPC. [Online; Accessed on 06/11/2022].

[25] Densify. Kubernetes taints and tolerations - guide and examples | densify. https://www.densify.com/kubernetes-autoscaling/kubernetes-taints. (Accessed on 06/15/2022).

[26] Densify. Kustomize tutorial with instructions & examples | densify. https://www.densify.com/kubernetes-tools/kustomize. (Accessed on 06/17/2022).

[27] Maria Deutscher. New study shows rapid growth in microservices adoption among enterprises. https://aws.amazon.com/microservices/, 2018. [Online; Accessed 19/5/2022].

[28] Mayuri Dhote. Let's learn about kubernetes cordon with it's illustration: - knoldus blogs. https://blog.knoldus.com/lets-learn-about-kubernetes-cordon-with-its-illustration/. (Accessed on 06/15/2022).

[29] The Geek Diary. LVM Architecture. https://www.thegeekdiary.com/redhat-centos-a-beginners-guide-to-lvm-logical-volume-manager/, 2021. [Online; Accessed 27/4/2022].

[30] Sandeep Dinesh. Kubernetes requests vs limits: Why adding them to your pods and namespaces matters | google cloud blog. https://cloud.google.com/blog/products/containers-kubernetes/kubernetes-best-practices-resource-requests-and-limits. (Accessed on 06/15/2022).

[31] Nigel Poulton [Online; Telecommunications engineer]. *The Kubernetes Book*. Shroff/Poulton, 07 2017.

[32] Seyi Ewegbemi. Kubernetes storage classes: An introduction | kubermatic. https://www.kubermatic.com/blog/keeping-the-state-of-apps-5-introduction-to-storage-classes/, July 2021. [Online; Accessed on 06/11/2022].

[33] HashiCorp. kubernetes_deployment | resources | hashicorp/kubernetes | terraform registry. https://registry.terraform.io/providers/hashicorp/kubernetes/latest/docs/resources/deployment. (Accessed on 06/27/2022).

[34] IBM. Containerization. https://www.ibm.com/cloud/learn/containerization, 2022. [Online; Accessed 19/5/2022].

[35] IBM. What is container orchestration? . https://www.vmware.com/topics/glossary/content/container-orchestration.html#:~:text=Container%20orchestration%20is%20the%20automation,networking%2C%20load%20balancing%20and%20more., 2022. [Online; Accessed 19/5/2022].

[36] Vaquar Khan. Overview of Kubernetes architecture and main concepts. https://github.com/vaquarkhan/Techies-Notes-wiki/blob/master/Overview-of-Kubernetes-architecture-and-main-concepts.md, 2019. [Online; Accessed 28/4/2022].

[37] Komodor. How to fix kubernetes node not ready error | komodor. https://komodor.com/learn/how-to-fix-kubernetes-node-not-ready-error/#:~:text=Common%20reasons%20for%20a%20Kubernetes,networking%20agent%20on%20the%20node). (Accessed on 07/08/2022).

[38] Mengying Li. Why Do We Use Kubernetes, Anyway? https://betterprogramming.pub/why-do-we-use-kubernetes-anyway-644544082f43, 2022. [Online; Accessed 19/5/2022].

[39] Misc. grpc. https://grpc.io/. [Online; Accessed on 06/11/2022].

[40] Misc. Documentation - the go programming language. https://go.dev/doc/, 2021. [Online; Accessed on 06/11/2022].

[41] Misc. Get to know container storage interface [online; csi] - alibaba cloud community. https://www.alibabacloud.com/blog/get-to-know-container-storage-interface-csi_598094, September 2021. [Online; Accessed on 06/11/2022].

[42] Misc. The guide to kubernetes cluster autoscaler by example. https://www.kubecost.com/kubernetes-autoscaling/kubernetes-cluster-autoscaler/, 2021. [Online; Accessed on 06/11/2022].

[43] Misc. Managed node groups - amazon eks. https://docs.aws.amazon.com/eks/latest/userguide/managed-node-groups.html, 2021. [Online; Accessed on 06/11/2022].

[44] Misc. Using the cluster autoscaler - the cluster api book. https://cluster-api.sigs.k8s.io/tasks/cluster-autoscaler.html, 2021. [Online; Accessed on 06/11/2022].

[45] Misc. What is a kubernetes operator? https://www.redhat.com/en/topics/containers/what-is-a-kubernetes-operator, May 2022. [Online; Accessed on 06/11/2022].

[46] opensource.com. LVM Architecture. https://opensource.com/business/16/9/linux-users-guide-lvm, 2022. [Online; Accessed 27/4/2022].

[47] Daniele Polencic. Allocatable memory and cpu in kubernetes nodes. https://learnk8s.io/allocatable-resources. (Accessed on 06/15/2022).

[48] Fatima Silveira. Kubernetes for dummies: Life of a pod. https://itnext.io/kubernetes-for-dummies-life-of-a-pod-fc8158e27aa, 2017. [Online; Accessed on 06/11/2022].

[49] Autoscaler Team. autoscaler/faq· kubernetes/autoscaler. https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md, 2021. [Online; Accessed on 06/11/2022].

[50] Kubernetes Team. spec/csi.proto - storage-interface/spec. https://github.com/container-storage-interface/spec/blob/master/csi.proto. [Online; Accessed on 06/11/2022].

[51] Kubernetes Team. Api-initiated eviction | kubernetes. https://kubernetes.io/docs/concepts/scheduling-eviction/api-eviction/, Jube 2021. [Online; Accessed on 06/11/2022].

[52] Kubernetes Team. Disruptions | kubernetes. https://kubernetes.io/docs/concepts/workloads/pods/disruptions/, November 2021. [Online; Accessed on 06/11/2022].

[53] Kubernetes Team. Kubernetes scheduler | kubernetes. https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/, December 2021. [Online; Accessed on 06/11/2022].

[54] Kubernetes Team. Pod lifecycle | kubernetes. https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/, March 2022. [Online; Accessed on 06/11/2022].

[55] Kubernetes Team. Safely drain a node | kubernetes. https://kubernetes.io/docs/tasks/administer-cluster/safely-drain-node/, March 2022. [Online; Accessed on 06/11/2022].

[56] Kubernetes Team. Storage classes | kubernetes. https://kubernetes.io/docs/concepts/storage/storage-classes/, April 2022. [Online; Accessed on 06/11/2022].

[57] Velotio Technologies. Kubernetes csi in action: Explained with features and use cases | by velotio technologies | velotio perspectives | medium. https://medium.com/velotio-perspectives/kubernetes-csi-in-action-explained-with-features-and-use-cases-4f966b910774, September 2019. (Accessed on 07/12/2022).

[58] Ivy Wigmore. What is an amazon ec2 instance? https://www.techtarget.com/searchaws/definition/Amazon-EC2-instances, July 2021. [Online; Accessed on 06/11/2022].

[59] Sun Zhiheng. Kubernetes persistent storage process - alibaba cloud community. https://www.alibabacloud.com/blog/kubernetes-persistent-storage-process_596505. (Accessed on 06/15/2022).