Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων
Και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης Και Υπολογιστων

Μηχανισμός Δημιουργίας Επαναχρησιμοποιήσιμων Δομικών Στοιχείων σε Ροές Εργασίας Μηχανικής Μάθησης με Docker στην Πλατφόρμα Kubeflow

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Παναγιώτης Μ. Γιαννούλης

Αθήνα, Ιούλιος 2022

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

Μηχανισμός Δημιουργίας Επαναχρησιμοποιήσιμων Δομικών Στοιχείων σε Ροές Εργασίας Μηχανικής Μάθησης με Docker στην Πλατφόρμα Kubeflow

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Παναγιώτης Μ. Γιαννούλης

Επιβλέπων Καθηγητής: 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

A Mechanism to Create Reusable Machine Learning Workflow Components with Docker and Kubeflow

DIPLOMA THESIS

Panagiotis M. Giannoulis

Athens, July 2022

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

A Mechanism to Create Reusable Machine Learning Workflow Components with Docker and Kubeflow

DIPLOMA THESIS

Panagiotis M. Giannoulis

Supervising Professor: Nectarios Koziris
Professor NTUA

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

. . . . .	. . . . .	. . . . .
Nectarios Koziris	Georgios Goumas	Dionisios Pnevmatikatos
Professor NTUA	Assoc. Professor NTUA	Professor NTUA

Athens, July 2022

. . . . . .

Παναγιώτης Μ. Γιαννούλης
Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

στους γονείς μου,
στ' αδέλφια μου

Σα βγεις στον πηγαιμό για την Ιθάκη,
να εύχεσαι να 'ναι μακρύς ο δρόμος,
γεμάτος περιπέτειες, γεμάτος γνώσεις.

— Κ. Π. Καβάφης

Η στοιχειοθεσία του κειμένου έγινε με το XeTeX 0.999993.

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

Περίληψη

Η Μηχανική Μάθηση (ΜΜ) αποτελεί σήμερα την βάση για την επίλυση σύνθετων προβλημάτων του πραγματικού κόσμου, παρέχοντας αξία σε όλους τους τομείς. Οι επιστήμονες δεδομένων μπορούν να αναπτύξουν και να εκπαιδεύσουν ένα μοντέλο Μηχανικής Μάθησης με αναμενόμενη απόδοση σε ένα γνωστό σύνολο δεδομένων. Ωστόσο, αποτελεί αληθινή πρόκληση η κατασκευή ενός ολοκληρωμένου συστήματος ΜΜ και η συνεχής λειτουργία του σε πραγματικό χρόνο. Το Kubeflow έρχεται να κάνει την ανάπτυξη μοντέλων Μηχανικής Μάθησης στον Κυβερνήτη απλή, άμεση και επεκτάσιμη. Όμως η εκτέλεση ροών εργασίας ΜΜ στο Kubeflow είναι ακόμη μία επίπονη διαδικασία και η αναπαραγωγιμότητα φαντάζει ουτοπία. Ένα από τα σημαντικότερα εμπόδια στην υιοθέτηση των σωληνώσεων Kubeflow εκτενώς και στην παραγωγή είναι οι γνώσεις που απαιτούνται για την εσωτερική λειτουργία του Kubeflow Pipelines DSL, του Κυβερνήτη και του Docker. Τελικά, η ανάπτυξη ενός Kubeflow Pipeline απαιτεί επαναλαμβανόμενες, επαχθείς και άγνωστες στους επιστήμονες δεδομένων λειτουργίες.

Στην παρούσα διπλωματική εργασία, αναπτύσσουμε μια αποτελεσματική προσέγγιση για την αυτόματη κατασκευή επαναχρησιμοποιήσιμων και αναπαραγώγιμων δομικών στοιχείων του Kubeflow Pipeline. Για να το πετύχουμε, εισάγουμε μια νέα βιβλιοθήκη για τη διαφανή αναπαραγωγή του περιβάλλοντος εργασίας του επιστήμονα δεδομένων μέσω της δημιουργίας εικόνων Docker σε μη προνομιούχα συστήματα. Έτσι, είμαστε σε θέση να μετατρέψουμε ένα δομικό στοιχείο του Kubeflow Pipeline, το οποίο είναι ένα αυτοτελές σύνολο κώδικα που εκτελεί ένα βήμα σε μια ροή εργασίας ML, σε μια συσκευασμένη και έτοιμη προς χρήση εικόνα Docker. Αξιοποιώντας την ικανότητα του μηχανισμού μας, οι επιστήμονες δεδομένων θα μπορούν να μοιράζονται, να ανακαλύπτουν, να αποθηκεύουν και να επαναχρησιμοποιούν συστατικά KFP σε μια πληθώρα από συνδυασμούς εξοικονομώντας πολύτιμο χρόνο για την εύρεση της καλύτερης ροής εργασίας ΜΜ για τις ανάγκες τους. Τέλος, θα κάνει την εξυπηρέτηση των μοντέλων τους εύκολη υπόθεση ακόμα και στα πιο απαιτητικά περιβάλλοντα.

Keywords

Machine Learning, MLOps, containers, OCI Images, Docker, layers, Kubernetes, Kubeflow, Kale, Rok, components

Abstract

Data science and Machine Learning are becoming core capabilities for solving complex real-world problems, transforming industries, and delivering value in all domains. Data scientists can implement and train an ML model with predictive performance on an offline holdout dataset, given relevant training data for their use case. However, the real challenge isn't building an ML model, the challenge is building an integrated ML system and to continuously operate it in production. The Kubeflow project is dedicated to making deployments of machine learning (ML) workflows on Kubernetes simple, portable and scalable. Running Kubeflow Pipelines is challenging and reproducibility is only a pipe dream. One of the major obstacles in adopting Kubeflow pipelines extensively and in production is the knowledge required for the inner workings of Kubeflow Pipelines DSL, Kubernetes concepts and Docker expertise to compile and run a workflow. Eventually, the deployment of a Kubeflow Pipeline (ML workflow) requires repetitious, burdensome, and unknown to data scientists operations.

In this diploma thesis, we develop an efficient approach to automatically build and transfer versioned, reusable and reproducible Kubeflow Pipeline components. We introduce a library to transparently reproduce the scientist's working environment by building images in unprivileged environments. Doing so, we are able to transform a Kubeflow Pipeline component, which is a self-contained set of code that performs one step in an ML workflow, into a packaged ready-to-use Docker image. Leveraging our mechanism's ability, ML engineers will be able to share, discover, store and reuse KFP components in various different combinations saving valuable time towards finding the best ML workflow for their needs. Furthermore, packaging a model will make serving a piece of cake even in the most demanding environments. Finally, the creation of reusable components will open the road for splitting ML workflows into distinct parts that will run on different clusters that target different development stages, workflow life-cycles, or provide access to limited services or computing resources.

Keywords

Machine Learning, MLOps, Containers, OCI Images, Docker, Layers, Kubernetes, Kubeflow, Kale, Rok, Components

Αντί Προλόγου

Οφείλω να ευχαριστήσω στο σημείο αυτό, τον επιβλέποντα της διπλωματικής μου, Καθηγητή Νεκτάριο Κοζύρη, ο οποίος πρώτος μου κοινώνησε το ενδιαφέρον του για τον κόσμο τον υπολογιστικών συστημάτων μέσω των διαλέξεών του.

Θα ήθελα να εκφράσω την ιδιαίτερη ευγνωμοσύνη μου προς τον Διδάκτορα Βαγγέλη Κούκη για την αδιάλειπτη και καθοριστική συμβολή του, τόσο στην καλλιέργεια του ενδιαφέροντός μου για ποικίλες πτυχές των υπολογιστικών συστημάτων, όσο και στην διαμόρφωση του τρόπου σκέψης μου ως προς την προσέγγιση σχετικών ζητημάτων. Υπήρξε εξαιρετικός μέντορας και μου πρόσφερε ανεκτίμητη βοήθεια για τα μελλοντικά μου βήματα ως μηχανικός. Τον ευχαριστώ επίσης που μου έδωσε την ευκαιρία να γίνω μέλος της οικογένειας Arrikto και να γνωρίσω εξαιρετικούς ανθρώπους και μηχανικούς. Του υπόσχομαι δε θα σταματήσω ποτέ να ρωτώ 'γιατί' κυνηγώντας πάντα και παντού τη βαθύτερη ουσία και γνώση.

Θέλω να ευχαριστήσω επίσης τον Stefano Fioravanzo τον οποίο γνώρισα ως επιβλέποντα στην διπλωματική μου στο περιβάλλον της Arrikto. Η εμπειρία του, τόσο ως μηχανικού λογισμικού και ως επιστήμονα πληροφορικής υπήρξε αρωγός για την ανάπτυξη των ικανοτήτων μου, υποδεικνύοντάς μου πως να είμαι επαγγελματίας στη δουλειά μου. Μου ενστάλαξε πως με ήρεμη και καθαρή ματιά, με απλότητα και η υπομονή μπορείς να αντιμετωπίσεις τα πάντα. Στην Arrikto ήρθα επίσης σε επαφή με τον Ηλία Κατσακιώρη ο οποίος μοιράστηκε μαζί μου τις σκέψεις του και ήταν πάντα εκεί στις δύσκολες στιγμές κατά την εκπόνηση της διπλωματικής εργασίας μου.

Οφείλω ακόμη ένα μεγάλο ευχαριστώ στους φίλους και συμφοιτητές μου Μιχάλη, Παντελή, Γιώργο, Δημήτρη, Χρήστο και Νίκο για την υποστήριξη που μου παρείχαν όλα αυτά τα χρόνια, οδηγώντας με στο να αναπτύξω και να διαμορφώσω την προσωπικότητα και το χαρακτήρα μου και να γίνω αυτός που είμαι σήμερα. Θα τους θυμάμαι για πάντα όπου και να βρίσκομαι.

Κυρίως όμως, θα ήθελα να ευχαριστήσω την οικογενειά μου, τους γονείς μου Ματθαίο και Ελένη, τα αδέλφια μου Νίκο και Μαίρη αλλά και τους Σπύρο και Νίκο. Η υπέρμετρη αγάπη τους υπήρξε, και συνεχίζει να υπάρχει, το μεγαλύτερο στήριγμα στην πορεία μου.

Παναγιώτης Γιαννούλης
Ιούλιος 2022

Περιεχόμενα

1 Εισαγωγή

1.1 Κίνητρο

Τα τελευταία χρόνια η δημιουργία περιεκτών ή αλλιώς containers για την κατάλληλη συσκευασία των εφαρμογών σε καλά ορισμένα, ασφαλή και απομονωμένα πακέτα έχει αναδειχθεί εξαιρετικά ως τεχνολογία. Στην πραγματικότητα, η έννοια του containerization και της απομόνωσης διεργασιών είναι δεκαετίες παλιά, αλλά η εμφάνιση το 2013 της μηχανής ανοικτού κώδικα Docker - ένα βιομηχανικό πρότυπο για containers με απλά εργαλεία ανάπτυξης και μια καθολική προσέγγιση συσκευασμού - επιτάχυνε την υιοθέτηση αυτής της τεχνολογίας. Σήμερα οι οργανισμοί χρησιμοποιούν όλο και περισσότερο την τεχνική του containerization για τη δημιουργία νέων εφαρμογών και για τον εκσυγχρονισμό των υφιστάμενων εφαρμογών στο cloud. Ίσως το πιο σημαντικό είναι πως η τεχνολογία αυτή επιτρέπει στις εφαρμογές να γράφονται μία φορά και να εκτελούνται οπουδήποτε. Η άνοδος των containers και της τεχνολογίας μικρουπηρεσιών είχε ως αποτέλεσμα εφαρμογές που περιλαμβάνουν πλέον εκατοντάδες ή μερικές φορές χιλιάδες containers. Η διαχείριση αυτών των φορτίων σε πολλαπλά περιβάλλοντα μπορεί να είναι πραγματικά δύσκολη και επώδυνη. Ο Κυβερνήτης είναι ένα εργαλείο ενορχήστρωσης containers ανοιχτού κώδικα που ικανοποιεί την ανάγκη αφαίρεσης του ελέγχου, της εποπτείας και της διαχείρισης πολυάριθμων containers.

Τα containers μαζί με τον Κυβερνήτη βοήθησαν τρομερά την κουλτούρα DevOps να εξελιχθεί και να αφαιρέσει τα όποια εμπόδια μεταξύ δύο παραδοσιακά απομονωμένων ομάδων, της ανάπτυξης και των λειτουργιών. Το DevOps είναι ένα σύνολο πρακτικών στον παραδοσιακό κόσμο της ανάπτυξης λογισμικού που επιτρέπει την ταχύτερη και

πιο αξιόπιστη ανάπτυξη λογισμικού στην παραγωγή. Βασίζεται στην αυτοματοποίηση, τα εργαλεία και τις ροές εργασίας για να αφαιρέσει την πολλές φορές χωρίς ουσία πολυπλοκότητα και να επιτρέψει στους προγραμματιστές να επικεντρωθούν σε πιο κρίσιμα ζητήματα. Τελικά, επιτυγχάνει οφέλη όπως η μείωση των κύκλων ανάπτυξης και η αύξηση της αξιοπιστίας των εκδόσεων των λογισμικών με την εισαγωγή δύο εννοιών:

- Συνεχής ενοποίηση (Continuous Integration - CI)
- Συνεχής παράδοση (Continuous Delivery - CD)

Σήμερα, η μηχανική μάθηση (ΜΜ) αποτελεί βασικό κομμάτι της καθημερινότητας των επιχειρήσεων, καθώς αποδεικνύεται πολύτιμη για την επίλυση διαφόρων προβλημάτων του πραγματικού κόσμου. Οι περισσότεροι κλάδοι που εργάζονται με μεγάλες ποσότητες δεδομένων έχουν αναγνωρίσει την αξία της τεχνολογίας αυτής. Με την άντληση πληροφοριών από αυτά τα δεδομένα - συχνά σε πραγματικό χρόνο - οι οργανισμοί είναι σε θέση να εργάζονται πιο αποτελεσματικά ή να αποκτούν πλεονέκτημα έναντι των ανταγωνιστών τους. Η μηχανική μάθηση έχει πάρα πολλές εφαρμογές και υπάρχει στην καθημερινότητά μας περισσότερο απ' ότι ίσως πιστεύουμε. Η κατηγοριοποίηση ηλεκτρονικών μηνυμάτων, η αναγνώριση ομιλίας, τα αυτοκινούμενα αυτοκίνητα, η μετάφραση της Google καθώς και κάθε είδους διαδικτυακές διαφημίσεις είναι μερικές μόνο από αυτές τις εφαρμογές.

Η Μηχανική Μάθηση σαν δραστηριότητα είναι εξαιρετικά δαπανηρή και ύψηλης έντασης απαιτώντας σημαντικό αριθμό πόρων για την υποστηριξή της. Αυτό ισχύει και για τη διαδικασία δημιουργίας ενός μοντέλου μηχανικής μάθησης και για την ανάπτυξη και την εκτέλεσή του σε περιβάλλον παραγωγής. Μπορούμε να αξιοποιήσουμε την τεχνολογία των containers για να διευκολύνουμε τη ροή εργασιών μηχανικής μάθησης και να βελτιώσουμε την απόδοση των μοντέλων; Φυσικά! Ο Κυβερνήτης μπορεί να ανταποκριθεί σε πολλές από τις υπολογιστικές προκλήσεις ενορχηστρώνοντας απαιτητικές ροές εργασίας Μηχανικής Μάθησης μέσω containers. Μια συνήθης πρακτική σήμερα, όπως συμβαίνει με τις περισσότερες εφαρμογές, είναι το πακετάρισμα και η ανάπτυξη εργασιών ΜΜ ως containers που διαχειρίζεται ο Κυβερνήτης. Όμως είναι σίγουρα ένα επαχθές έργο για τον επιστήμονα δεδομένων να μετακινηθεί από τον τοπικό υπολογιστή του όπου αναπτύσσει το μοντέλο του σε ένα Kubernetes Cluster, μια δηλαδή, συστοιχία υπολογιστών στον Κυβερνήτη, για εκπαίδευση και εξυπηρέτηση των

μοντέλων του στον πραγματικό κόσμο.

Το κίνητρό μας και αυτό που επιδιώκουμε σε αυτή τη διπλωματική εργασία είναι να κάνουμε την καθημερινότητα των Επιστημόνων Δεδομένων ευκολότερη γεφυρώνοντας το χάσμα μεταξύ ανάπτυξης και παραγωγής. Θέλουμε να δώσουμε τη δυνατότητα στους επιστήμονες δεδομένων να επικεντρωθούν στην κατασκευή και την ανάπτυξη των μοντέλων τους. Πόσο μαγικό θα ήταν να μπορούν να παρέχουν τα μοντέλα τους στους χρήστες των εκάστοτε εφαρμογών με το πάτημα μόνο ενός κουμπιού!

Η κατασκευή ενός συστήματος ΜΜ που θα λειτουργεί συνεχώς στην παραγωγή είναι μια σημαντικά επίπονη δουλειά για έναν επιστήμονα δεδομένων. Πολλές ομάδες διαθέτουν επιστήμονες δεδομένων και ερευνητές ΜΜ που μπορούν να κατασκευάσουν μοντέλα τελευταίας τεχνολογίας, αλλά η διαδικασία κατασκευής και ανάπτυξης μοντέλων ΜΜ στον Κυβερνήτη είναι εντελώς χειροκίνητη. Ο κύκλος ζωής της μηχανικής μάθησης αποτελείται από πολλά πολύπλοκα στοιχεία, όπως η λήψη δεδομένων, η προετοιμασία δεδομένων, η εκπαίδευση, ρύθμιση, ανάπτυξη μοντέλων και παρακολούθηση μοντέλων και πολλά άλλα. Τι θα γινόταν αν εφαρμόζαμε την τεχνική του DevOps - κατάλληλα προσαρμοσμένη - στις εφαρμογές Μηχανικής Μάθησης;

Machine Learning Operations
Ομοίως με το DevOps, το MLOps ήρθε να αντιμετωπίσει αυτά τα ζητήματα περιλαμβάνοντας τη συνεχή βελτίωση του κύκλου ζωής της μηχανικής μάθησης. Το MLOps προωθεί την επικοινωνία και τη συνεργασία μεταξύ των επαγγελματιών που ασχολούνται με DevOps και των επιστημόνων δεδομένων. Το MLOps σημαίνει Machine Learning Operations (Λειτουργίες Μηχανικής Μάθησης) και επικεντρώνεται στην αυτοματοποίηση της διαδικασίας μεταφοράς των μοντέλων μηχανικής μάθησης στην παραγωγή και, στη συνέχεια, στη συντήρηση και την παρακολούθησή τους.

1.2 Διατύπωση Προβλήματος

Το MLOps είναι αδιαμφισβήτητα το μέλλον. Ωστόσο, δεν βρίσκεται στην φάση όπου οι Data Scientists δεν θα εμπλέκονται με μη σχετικές με την ανάπτυξης των μοντέλων τους λειτουργίες. Πολλά προϊόντα λογισμικού προσπαθούν να αντιμετωπίσουν το πρόβλημα της ανάπτυξης ροών εργασίας μηχανικής μάθησης στον Κυβερντήτη. Παρά τον στόχο τους να κάνουν τη χρήση τους απλή και εύκολη, εξακολουθούν να μην είναι

ελκυστικά για τους επιστήμονες δεδομένων. Κατά συνέπεια, ακόμη και απλές λειτουργίες απαιτούν πολύωρη μελέτη γι' αυτούς σπαταλώντας έτσι πολυτίμο χρόνο απο την εργασία τους.

Η επιστήμη των δεδομένων είναι εγγενώς μια ροή εργασίας με χρήση της τεχνικής σωλήνωσης (pipeline), από την προετοιμασία των δεδομένων, την εκπαίδευση και την ανάπτυξη κάθε έργο ΜΜ οργανώνεται σε αυτά τα λογικά βήματα. Χωρίς την επιβολή μιας αυστηρής δομής pipeline στα έργα της επιστήμης δεδομένων, είναι συχνά πολύ εύκολο να δημιουργηθεί ακατάστατος κώδικας, με περίπλοκες εξαρτήσεις δεδομένων και δύσκολα αναπαραγώγιμα αποτελέσματα.

Μια δημοφιλής εργαλειοθήκη μηχανικής μάθησης που είναι αφιερωμένη στο να κάνει την ανάπτυξη ροών εργασίας ΜΜ στον Κυβερνήτη απλή, φορητή και κλιμακούμενη είναι το Kubeflow. Το Kubeflow Pipelines (KFP) είναι το συστατικό του Kubeflow που είναι υπεύθυνο για την ενορχήστρωση από άκρο σε άκρο των σωληνώσεων ΜΜ. Ένα pipeline αποτελεί περιγραφή μιας ροής εργασίας μηχανικής μάθησης, που περιλαμβάνει όλα τα στοιχεία της ροής εργασίας και τον τρόπο με τον οποίο αυτά τα στοιχεία σχετίζονται μεταξύ τους με τη μορφή γράφου.

Το Kubeflow μαζί με τα Pipelines είναι ένα εξαιρετικό εργαλείο για να οδηγήσει τους επιστήμονες δεδομένων να υιοθετήσουν μια πειθαρχημένη νοοτροπία κατά την ανάπτυξη κώδικα ΜΜ και την κλιμάκωσή του στο Cloud. Το Python SDK του Kubeflow Pipelines βοηθάει στην αυτοματοποίηση της δημιουργίας τέτοιων pipelines, ειδικά όταν πρόκειται για πολύπλοκες ροές εργασίας και περιβάλλοντα παραγωγής. Παρ' όλ' αυτά, όταν παρουσιάζεται αυτή η τεχνολογία σε Data Scientists που δεν έχουν την απαιτούμενη τεχνογνωσία σχετική με ζητήματα που αφορούν σε μηχανικούς λογισμικού, το KFP μπορεί να εκληφθεί ως πολύ περίπλοκο και δύσκολο στη χρήση.

Η Επιστήμη Δεδομένων είναι συχνά θέμα πρωτοτυποποίησης νέων ιδεών, εξερεύνησης νέων δεδομένων και μοντέλων, γρήγορου και επαναληπτικού πειραματισμού. Σε αυτά τα σενάρια θα προτιμούσε κανείς να εκτελεί απλώς κάποιο πρόχειρο κώδικα και να αναλύει τα αποτελέσματα παρά να δημιουργεί πολύπλοκες ροές εργασίας με ένα συγκεκριμένο SDK. Το Kubeflow έχει γίνει η προτιμώμενη πλατφόρμα αυτοματοποίησης MLOps επειδή προσφέρει μια ροή εργασίας σε επίπεδο παραγωγής, δηλαδή στον Κυβερνήτη, που είναι αυτοματοποιημένη, φορητή, αναπαραγώγιμη και ασφαλής. Ωστόσο, υπάρχουν επί του παρόντος δύο σημαντικά κενά μεταξύ των δυνατοτήτων του

MLOps και της πραγματικότητας της ανάπτυξης ΜΜ μοντέλων στο Kubeflow σε κλίμακα παραγωγής:

1. *Κενό αυτοματισμού*: Οι επιστήμονες δεδομένων δεν είναι ειδικοί σε θέματα DevOps ή υποδομών, παρ' όλ' αυτά σήμερα απαιτείται από τους ίδιους να δημιουργούν εικόνες Docker, να τις δημοσιεύουν σε μητρώα, να γράφουν ορισμούς DSL και DAG για pipelines, να γράφουν αρχεία YAML κ.λπ.

2. *Κενό αναπαραγωγιμότητας*: Η δημιουργία πολλαπλών εκδόσεων των δεδομένων αποτελεί μια κόλαση για τους επιστήμονες δεδομένων. Δυστυχώς, τόσο τα δεδομένα όσο και ο κώδικας αλλάζουν συνεχώς, καθιστώντας την αποσφαλμάτωση σχεδόν αδύνατη.

Τα Kubeflow Pipelines αποτελούνται από έναν αριθμό βημάτων, καθένα από τα οποία εκτελείται ως ανεξάρτητη διαδικασία σε σχέση με τα υπόλοιπα. Κάθε βήμα εκτελεί μια συγκεκριμένη εργασία, π.χ.: προεπεξεργασία ενός συνόλου δεδομένων, εκπαίδευση ενός μοντέλου, παραγωγή προβλέψεων σε ένα υποσύνολο δοκιμής κ.λπ. Ένα δομικό στοιχείο (component) του pipeline είναι ένα αυτοτελές σύνολο κώδικα, συσκευασμένο ως εικόνα Docker, που εκτελεί ένα βήμα του pipeline. Για παράδειγμα, ένα δομικό στοιχείο μπορεί να είναι υπεύθυνο για την προεπεξεργασία δεδομένων, τον μετασχηματισμό δεδομένων, την εκπαίδευση μοντέλων κ.ο.κ.

Το Kubeflow δεν έχει καταφέρει ακόμη να διαχωρίσει τους επιστήμονες δεδομένων από έννοιες όπως containers, Κυβερνήτης, αρχεία YAML, ροές εργασίας Argo. Συνεπώς, οι χρήστες καλούνται συχνά να κατασκευάσουν εικόνες Docker για κάθε component ενός Kubeflow Pipeline το οποίο αναπτύσσουν. Η μόνη λύση που παρέχεται από το Kubeflow είναι να χρησιμοποιείται από προεπιλογή μια βασική εικόνα Python και να επεκτείνεται ο κώδικας του χρήστη με τον ελάχιστο αριθμό παραμέτρων, προκειμένου να μπορούν να διαβαστούν οι είσοδοι και έξοδοι του εκάστοτε δομικού στοιχείου του pipeline. Το KFP προτείνει στους χρήστες να δημιουργήσουν τη δική τους εικόνα Docker αν έχουν πολύπλοκες εξαρτήσεις. Αυτό είναι λογικό, επειδή η προεπιλεγμένη προσέγγιση και συμπεριφορά του KFP συνοδεύεται από σοβαρές επιπτώσεις και περιορισμούς. Για παράδειγμα:

1. Δεν θα πρέπει να χρησιμοποιείται κανένας κώδικας που δηλώνεται εκτός του ορισμού της συνάρτησης.

2. Οι δηλώσεις εισαγωγής βιβλιοθηκών πρέπει να προστίθενται μέσα στη συνάρτηση (όπως και οι βοηθητικές συναρτήσεις)

Ταυτόχρονα το Kubeflow αποτελεί εξ' ορισμού ένα μη-προνομιούχο περιβάλλον. Επομένως η χρήση του Docker είναι ουσιαστική αδύνατη, μιας και απαιτεί root δικαιώματα για την επικοινωνία με τον δαίμονα του Docker.

1.3 Προτεινόμενη Λύση

Το Kale μαζί με το Rok έκανε ένα πρώτο τεράστιο βήμα για την αντιμετώπιση αυτών των δυσκολιών παρέχοντας ένα εργαλείο για την απλοποίηση της διαδικασίας του deployment ενός Jupyter Notebook σε Kubeflow Pipeline workflows. Η μετάφραση του Jupyter Notebook απευθείας σε pipeline εξασφαλίζει ότι όλα τα δομικά στοιχεία επεξεργασίας είναι καλά οργανωμένα και ανεξάρτητα το ένα από το άλλο, ενώ παράλληλα αξιοποιεί την παρακολούθηση των πειραμάτων και την οργάνωση των ροών εργασίας που παρέχει out-of-the-box το Kubeflow. Πώς όμως κατασκευάζει το Kale τις απαραίτητες, όπως αναφέραμε προηγούμενως, εικόνες Docker για τα components του KFP; Στην πραγματικότητα δεν το κάνει. Αντίθετα, το Rok παίρνει στιγμιότυπα του περιβάλλοντος Kubeflow του χρήστη (κώδικας + δεδομένα, δηλαδή τα περιεχόμενα του component) και τα συνδέει με τα Pods του Κυβερνήτη όπου θα εκτελεστεί ο κώδικας του επιμέρους στοιχείου του pipeline. Δυστυχώς, αυτή η προσέγγιση έχει ορισμένα μειονεκτήματα:

1. Το Kale δεν μπορεί να παράγει διαμοιραζόμενα και αυτόνομα δομικά στοιχεία KFP, αφού πρέπει να πακεταριστούν μαζί με τις εξαρτήσεις τους σε εικόνες Docker, κάτι που δεν συμβαίνει όταν γίνεται χρήση του Rok για την εκτέλεση των pipelines.

2. Τα pipelines μεταγλωττίζονται και εκτελούνται μόνο υπό την αιγίδα του Rok. Όμως το Rok είναι ένα επιχειρηματικό, βαρύ και ακριβό λογισμικό που εξυπηρετεί ανάγκες για πολύ απαιτητικούς χρήστες, αποκλείοντας έτσι τους πιο συμβατικούς.

3. Το Rok δεν μπορεί να συνδεθεί εύκολα σε πολύ περιορισμένα περιβάλλοντα με ελάχιστους διαθέσιμους πόρους, πράγμα που σημαίνει ότι τα βήματα των pipelines

δεν μπορούν να εκτελεστούν οπουδήποτε.

Ενώ το Kale με το Rok λύνει ορισμένα από τα σημαντικότερα προβλήματα του Kubeflow, εξακολουθούν να υπάρχουν κενά που πρέπει να καλυφθούν. Και αυτός είναι ο απώτερος στόχος μας σε αυτή τη διπλωματική εργασία, ν' αντιμετωπίσουμε, δηλαδή, αυτές τις ελλείψεις που αναφέρθηκαν παραπάνω. Το βλέμμα μας θα είναι πάντα στραμμένο προς την κατεύθυνση της αυτοματοποίησης και της αναπαραγωγιμότητας. Με άλλα λόγια, θα επιδιώξουμε να διευκολύνουμε τους Data Scientists να περάσουν από το περιβάλλον ανάπτυξής τους στην παραγωγή με μερικά κλικ και χωρίς να απαιτείται βαθιά γνώση ούτε του Κυβερνήτη, ούτε των Containers. Επιπλέον, θα επιδιώξουμε να τους δώσουμε τη δυνατότητα να πειραματιστούν αβίαστα με διάφορους συνδυασμούς ροών εργασίας ΜΜ, επαναχρησιμοποιώντας ήδη συσκευασμένα pipeline components που έχουν δημιουργηθεί είτε από τους ίδιους είτε από άλλους μηχανικούς. Η ελπίδα μας είναι να τους εξοικονομήσουμε πολύτιμο χρόνο κατά την διαδικασία εύρεσης του καλύτερου μοντέλου για τις ανάγκες τους και να ενισχύσουμε την μεταξύ τους συνεργασία.

Για να συμβούν όλα αυτά, θα αναζητήσουμε τρόπους για να συσκευάσουμε Kubeflow Pipeline components μαζί με τις πολύπλοκες εξαρτήσεις τους σε μια εικόνα Docker με απρόσκοπτο τρόπο. Για να το πετύχουμε αυτό καλούμαστε να εξερευνήσουμε την τεχνολογία container ειδόλων στον πυρήνα τους. Η απόκτηση βαθιάς κατανόησης του τρόπου με τον οποίο το Docker και το OCI παράγουν έναν πλήρως κλειστό χώρο που μπορεί να χρησιμοποιηθεί για να περιέχει, να αποθηκεύει και να μεταφέρει εφαρμογές, θα μας επιτρέψει να αναπτύξουμε τον τρόπο με τον οποίο χειριζόμαστε τις εικόνες Docker και OCI. Έπειτα, θα είμαστε σε θέση να εισαγάγουμε και να καθιερώσουμε έναν τρόπο για την κατασκευή εικόνων αυτόματα σε μη προνομιούχα περιβάλλοντα. Σημειώνουμε εδώ ότι δεν είναι δυνατή η χρήση του Docker στο Kubeflow μιας και τα εργαλεία που παρέχει αποτελούν containers που τρέχουν στον Κυβερνήτη με την μορφή Pods. Με αυτό τον μηχανισμό στα χέρια μας θα έχουμε δημιουργήσει τις απραίτητες βάσεις ώστε να επεκτείνουμε κατάλληλα το Kale και τις δυνατότητές του για να υποστηρίξουμε τη δημιουργία επαναχρησιμοποιούμενων, διαμοιραζόμενων και αυτόνομων δομικών στοιχείων KFP. Συγκεκριμένα, αρχικά θα ενσωματώσουμε τον νέο μηχανισμό στο Kale με ένα API που αποτελείται από δύο υψηλού επιπέδου λειτουργίες: τη δημιουργία και εισαγωγή δομικών στοιχείων KFP.

Συνολικά, θα επιχειρήσουμε να πακετάρουμε το περιβάλλον εργασίας του χρήστη μαζί

με τον κώδικα και τα δεδομένα του, χρησιμοποιώντας τον μηχανισμό δημιουργίας εικόνων που θα έχουμε αναπτύξει προηγουμένως. Εν συνεχεία, θα επεκτείνουμε το Kale για να παράγουμε κατάλληλα αυτόνομα στοιχεία KFP που μπορούν να εκτελεστούν ακόμα και στα πιο απαιτητικά περιβάλλοντα.

1.4 Δομή Διπλωματικής Εργασίας

Το περιεχόμενο της διπλωματικής εργασίας είναι δομημένο ως εξής:

- Κεφάλαιο 2: μια σύντομη επισκόπηση ορισμένων από τις βασικές έννοιες και τα συστήματα στα οποία βασίζεται η εργασία μας.
- Κεφάλαιο 3: μια ανάλυση της αρχιτεκτονικής της λύσης μας και των σχεδιαστικών αποφάσεων από μια προοπτική υψηλότερου επιπέδου.
- Κεφάλαιο 4: μια σύντομη παρουσίαση ορισμένων από τα κομβικά σημεία της διαδικασίας ανάπτυξής μας, συμπεριλαμβανομένων κυρίως λεπτομερειών σχετικά με την υλοποίηση ορισμένων σχεδιαστικών επιλογών που θα μπορούσαν να είχαν υλοποιηθεί με πολλούς διαφορετικούς τρόπους.
- Κεφάλαιο 5: συμπερασματικές παρατηρήσεις και μελλοντικές βελτιώσεις και επεκτάσεις της προτεινόμενης λύσης μας.

2

Υπόβαθρο

Σε αυτό το κεφάλαιο παρέχουμε κάποιες στοιχειώδεις πληροφορίες σχετικά με διάφορες πτυχές του γνωστικού υπόβαθρου εικόνων περιεκτών και την παραγωγή αυτόνομων και επαναχρησιμοποιήσιμων στοιχείων σε αυτή την εργασία. Ξεκινάμε με την αποδόμηση των containers και της υλοποίησής τους στον πυρήνα του Linux στα κύρια συστατικά τους: Χώροι Ονομάτων και Ομάδες Ελέγχου Αφού περιγράψουμε εν συντομία τη λειτουργικότητα αυτών των συστατικών, συζητάμε για τα είδωλα των containers και τις προδιαγραφές αυτών με βάση το Open Container Initiative. Στη συνέχεια, δείχνουμε πώς όλες αυτές οι έννοιες και οι μηχανισμοί μπορούν να χρησιμοποιηθούν για να διαμορφώσουν συνεργατικά τα θεμέλια μιας μηχανής εκτέλεσης περιεκτών (containers): συζητάμε για ορισμένες πτυχές του Docker και την συμβολή του για το σχεδιασμό και την υλοποίηση ενός τέτοιου συστήματος. Προχωράμε στη συνοπτική ανάλυση ορισμένων πτυχών του Κυβερνήτη και του σκεπτικού του: την οπτική του για τη διαχείριση των container, ορισμένες επιλογές αρχιτεκτονικής και μηχανικής λογισμικού, διάφορες αφαιρέσεις που δημιουργεί και πολλά άλλα. Το κεφάλαιο ολοκληρώνεται με μια σύντομη παρουσίαση των Kubeflow, Kale και Rok και της αξίας τους στις λειτουργίες μηχανικής μάθησης (Machine Learning Operations).

Σε αυτό το σημείο, πρέπει να σημειώσουμε ότι το περιεχόμενο αυτού του κεφαλαίου δεν αποτελεί σε καμία περίπτωση μια πλήρη ανάλυση ή επεξήγηση όλων των εννοιών και συστημάτων που εμπλέκονται στη διπλωματική εργασία. Αποτελεί μάλλον μια υψηλού επιπέδου επισκόπηση της βασικής λειτουργικότητας και αρχιτεκτονικής ορισμένων από αυτά. Ως εκ τούτου, το κεφάλαιο αυτό είναι ασφαλές να παραλειφθεί από τους αναγνώστες που είναι ήδη εξοικειωμένοι με τα θέματα που αναφέρονται στην παραπάνω

παράγραφο, ενώ παρέχει μόνο στοιχειώδεις, υψηλού επιπέδου πληροφορίες σχετικά με αυτά στους αναγνώστες που δεν είναι ήδη εξοικειωμένοι με αυτά, καθώς σκόπιμα απουσιάζουν οι λεπτομέρειες. Πρόσθετες βασικές έννοιες και γνώσεις που μπορεί να απαιτούνται για την πλήρη κατανόηση της συλλογιστικής των επιχειρημάτων της διατριβής παρουσιάζονται ενίοτε και στα επόμενα κεφάλαια. Τέτοιες έννοιες μπορεί να αποτελούν αποφάσεις σχεδιασμού ή υλοποίησης, οπότε ταιριάζουν καλύτερα στο κεφάλαιο 3 ή στο κεφάλαιο 4.

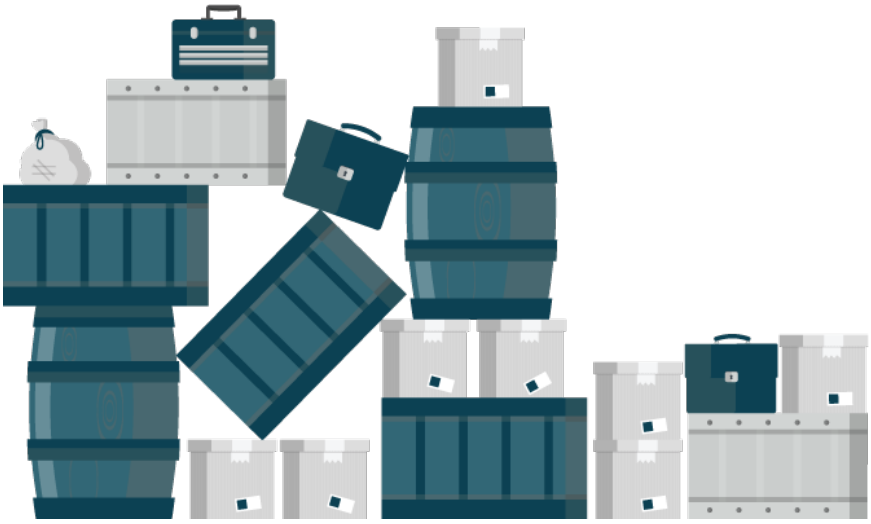
2.1 Εικονικοποίηση σε επίπεδο Λειτουργικού Συστήματος

2.1.1 Ο Κόσμος των Περιεκτών

Τι είναι container;
Σύμφωνα με τη Βικιπαίδεια, ένα εμπορευματοκιβώτιο είναι οποιοδήποτε δοχείο ή περίβλημα για τη συγκράτηση ενός προϊόντος που χρησιμοποιείται στην αποθήκευση, τη συσκευασία και τη μεταφορά, συμπεριλαμβανομένης της ναυτιλίας. Τα πράγματα που φυλάσσονται μέσα σε ένα εμπορευματοκιβώτιο προστατεύονται από πολλές πλευρές, καθώς βρίσκονται μέσα στη δομή του [1]. Ο όρος εφαρμόζεται συχνότερα σε συσκευές που κατασκευάζονται από υλικά που είναι ανθεκτικά και συχνά είναι μερικώς ή πλήρως άκαμπτα. Μπορούμε να θεωρήσουμε ένα δοχείο ως ένα βασικό εργαλείο που δημιουργεί έναν μερικώς ή πλήρως κλειστό χώρο που μπορεί να χρησιμοποιηθεί για να περιέχει, να αποθηκεύει και να μεταφέρει αντικείμενα ή υλικά.

Γιατί να χρησιμοποιήσουμε έναν container;
Τα χαρακτηριστικά του προϊόντος που δημιουργούν χρησιμότητα για ένα δοχείο υπερβαίνουν την απλή προστασία του περιεχομένου από κραδασμούς και υγρασία. Ένας καλά σχεδιασμένος container παρουσιάζει επίσης ευκολία χρήσης, δηλαδή είναι εύκολο για τον εργαζόμενο να ανοίξει ή να κλείσει, να εισάγει ή να αφαιρέσει το περιεχόμενο και να χειριστεί το κιβώτιο αυτό κατά τη μεταφορά. Επιπλέον, ένας καλός container θα έχει βολικές και ευανάγνωστες θέσεις σήμανσης, σχήμα που ευνοεί την αποτελεσματική στοίβαξη και αποθήκευση και εύκολη ανακύκλωση στο τέλος της ωφέλιμης ζωής του [1]. Φανταστείτε πως προσπαθείτε να μεταφέρετε και να στοιβάσετε ακανόνιστα

σχήματα και μεγέθη κουτιών. Αυτή είναι η εικόνα που πρέπει να σας έρχεται αμέσως στο μυαλό όταν λείπουν τα πρότυπα και οι προδιαγραφές.



Σχήμα 2.1: *Κουτιά που φορτώνονται χειροκίνητα σε τρένα και πλοία το 1921*

Τι είναι ένας Linux Container;

Στον κόσμο των μηχανικών υπολογιστών -κατά καιρούς- οι όροι μιας νέας τεχνολογίας αντικατοπτρίζουν την ίδια την τεχνολογία και περιγράφουν τη λειτουργικότητά της. Αυτή είναι και η περίπτωση με τους Linux Containers. Ο ορισμός του φυσικού εμπορευματοκιβωτίου είναι σχεδόν ταυτόσημος αυτού για τον κόσμο των Linux Containers. Ένας Linux container είναι ένα σύνολο από 1 ή περισσότερες διεργασίες που είναι απομονωμένες από το υπόλοιπο σύστημα [2] (προστατευμένες από πολλές πλευρές). Όλα τα αρχεία που είναι απαραίτητα για την εκτέλεσή τους παρέχονται από ένα είδωλο του container ή αλλιώς εικόνα/image (πακέτο), πράγμα που σημαίνει ότι τα Linux containers είναι φορητά και συνεπή καθώς μετακινούνται (μεταφέρονται) από την ανάπτυξη, στις δοκιμές και τελικά στην παραγωγή.

Όπως και ένα κανονικό πρόγραμμα Linux, οι containers έχουν πραγματικά δύο καταστάσεις - ανάπαυση και εκτέλεση. Όταν βρίσκεται σε κατάσταση ηρεμίας, ο container είναι ένα αρχείο (ή ένα σύνολο αρχείων) που αποθηκεύεται στο δίσκο. Αυτό αναφέρεται ως εικόνα ή είδωλο του container (Container Image). Όταν ένας container εκκινείται, η μηχανή Container Engine αποσυμπιέζει τα απαιτούμενα αρχεία και μεταδεδομένα και στη συνέχεια τα παραδίδει στον πυρήνα του Linux. Μόλις εκτελεστεί, αποτελεί πια μια διεργασία του Linux. Η διαδικασία για την εκκίνηση των containers, καθώς και η μορφή της εικόνας στο δίσκο, ορίζονται και διέπονται από πρότυπα.

Γιατί να χρησιμοποιήσουμε έναν Linux container;

Τα χαρακτηριστικά ενός Linux container είναι σχεδόν ίδια με τα χαρακτηριστικά ενός φυσικού εμπορευματοκιβωτίου. Το ίδιο ισχύει και για τα πλεονεκτήματά τους. Φανταστείτε ότι αναπτύσσετε μια εφαρμογή. Κάνετε τη δουλειά σας σε ένα φορητό υπολογιστή και το περιβάλλον σας έχει μια συγκεκριμένη διαμόρφωση. Άλλοι προγραμματιστές μπορεί να έχουν ελαφρώς διαφορετικές διαμορφώσεις και σίγουρα όχι πανομοιότυπο χώρο εργασίας. Η εφαρμογή που αναπτύσσετε βασίζεται σε αυτή τη διαμόρφωση και εξαρτάται από συγκεκριμένες βιβλιοθήκες και αρχεία. Θέλετε να μιμηθείτε αυτά τα περιβάλλοντα όσο το δυνατόν περισσότερο τοπικά, αλλά χωρίς όλα τα γενικά έξοδα της αναδημιουργίας περιβαλλόντων προκειμένου να κάνετε την εφαρμογή σας να λειτουργεί σε αυτά. Πώς διασφαλίζετε την ποιότητα και πώς μπορείτε να αναπτύξετε την εφαρμογή σας χωρίς επίπονες και χρονοβόρες διαδικάσιες ή και εξ' ολοκλήρου επανεγγραφή της εφαρμογής αλλά και επιδιόρθωση σφαλμάτων;

Ο container που περιέχει την εφαρμογή μας έχει τις απαραίτητες βιβλιοθήκες, εξαρτήσεις και αρχεία, ώστε να μπορούμε να την μετακινήσουμε από έναν τοπικό υπολογιστή σε ένα περιβάλλον παραγωγής χωρίς δυσάρεστες παρενέργειες. Στην πραγματικότητα, τα περιεχόμενα ενός container image μπορούν να θεωρηθούν ως εγκατάσταση μιας διανομής Linux, επειδή έρχονται πλήρη με πακέτα RPM, αρχεία ρυθμίσεων κ.λπ.

Αυτό είναι ένα συνηθισμένο παράδειγμα των Linux Containers που αναδεικνύει την ικανότητά τους να προσαρμόζονται ακόμη και στους πιο απαιτητικούς χώρους. Ταυτόχρονα αυτή τους η ικανότητα αποδεικνύει αποδοτικότητα χώρου που έχουν. Οι containers μπορούν να εφαρμοστούν σε πολλά διαφορετικά προβλήματα όπου απαιτείται φορητότητα, διαμορφωσιμότητα και απομόνωση. Ανοίγεις το κουτί, προσθέτεις κάποιο περιεχόμενο και κλείνεις το κουτί. Κανένα από τα πιο περίεργα, παράξενα και απαιτητικά περιβάλλοντα δεν μπορεί να μας εμποδίσει από το να επεκτείνουμε/συρρικνώσουμε και να μεταφέρουμε την εφαρμογή σας απρόσκοπτα. Το νόημα των Linux containers είναι να αναπτύσσουμε ταχύτερα τις εφαρμογές μας και να καλύπτουμε τις επιχειρηματικές ανάγκες καθώς προκύπτουν. Σε ορισμένες περιπτώσεις, οι containers είναι απαραίτητοι επειδή είναι ο μόνος τρόπος για να παρέχεται η επεκτασιμότητα που χρειάζεται μια εφαρμογή. Ανεξάρτητα από την υποδομή-στο χώρο μας, στο cloud ή σε ένα υβρίδιο των δύο-οι containers καλύπτουν τη ζήτηση [2].

Η τεχνική του containerization είναι μια μορφή εικονικοποίησης σε επίπεδο λειτουργικού συστήματος (OS), απλοποιεί την ανάπτυξη και τη διαχείριση των εφαρμογών παρέχοντας μια ευέλικτη, χαμηλού κόστους λύση εικονικοποίησης. Τα τελευταία χρόνια, οι containers θεωρούνται επάξιος αντικαταστάτης της εικονικοποίησης που βασίζεται σε VirtualMachine (VM). Είναι ομως; Σίγουρα προσφέρουν καλύτερες επιδόσεις και ευελιξία αλλά το κάνουν σε βάρος της απομόνωσης και της ασφάλειας σε σύγκριση με την παραδοσιακή εικονικοποίηση με hypervisor.

Σχήμα 2.2: *Τέλεια στοιβαγμένα εμπορευματοκιβώτια που δείχνουν την ευκολία χρήσης τους και τα οφέλη τους στην αποστολή.*

2.1.2 Εικονικοποίηση

Η παραδοσιακή λύση για την ενεργοποίηση της απομόνωσης και του διαμοιρασμού των πόρων χρησιμοποιεί την εικονικοποίηση που βασίζεται σε επόπτη. Ο επόπτης ή αλλιώς hypervisor είναι υπεύθυνος για την εκτέλεση ενός μηχανισμού αφαίρεσης, στοχευμένο στην απόκρυψη λεπτομερειών της υλοποίησης και της κατάστασης ορισμένων υπολογιστικών πόρων από πελάτες των πόρων αυτών (π.χ. εφαρμογές, άλλα συστήματα, χρήστες κλπ). Η εν λόγω αφαίρεση μπορεί είτε να αναγκάζει έναν πόρο να συμπεριφέρεται ως πλειάδα πόρων (π.χ. μία συσκευή αποθήκευσης σε διακομιστή τοπικού δικτύου), είτε πολλαπλούς πόρους να συμπεριφέρονται ως ένας (π.χ. συσκευές αποθήκευσης σε κατανεμημένα συστήματα). Η εικονικοποίηση δημιουργεί μία εξωτερική διασύνδεση η οποία αποκρύπτει την υποκείμενη υλοποίηση (π.χ. πολυπλέκοντας την πρόσβαση από διαφορετικούς χρήστες) [3]. Αυτές οι αφαιρέσεις ονομάζονται Εικονικές Μηχανές (Virtual Machines). Κάθε VM έχει το δικό του λειτουργικό σύστημα που εκτελείται εντελώς απομονωμένο από άλλα VM. Κατά συνέπεια, διαφορετικά λειτουργικά συστήματα μπορούν να εικονικοποιηθούν σε έναν μόνο κεντρικό υπολογιστή. Οι hypervisors προσφέρουν καλύτερη απομόνωση μεταξύ των πόρων σε σχέση με άλλες τεχνολογίες, όπως τα containers, επειδή οι εφαρμογές σε ξεχωριστά VM εκτελούνται σε διαφορετικό Guest OS. Η αυξημένη απομόνωση που προσφέρουν τα VMs επηρεάζει

σημαντικά τις επιδόσεις: Πρώτον, η εκκίνηση και επανεκκίνηση ενός VM μπορεί να διαρκέσει δεκάδες δευτερόλεπτα, καθώς πρέπει να ξεκινήσει ένα πλήρες λειτουργικό σύστημα. Δεύτερον, επειδή μιμείται μια πλήρης μηχανή, οι εντολές εξομοιώνονται ή μεταγλωττίζονται just in time (JIT), γεγονός που μπορεί να επιφέρει πλήγμα στις επιδόσεις.

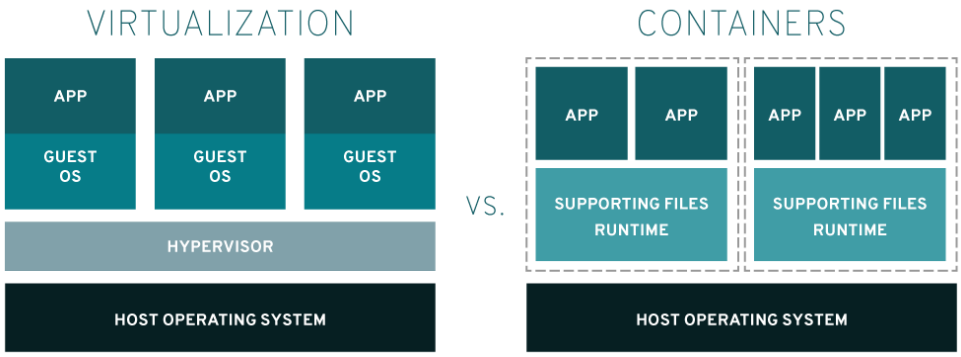
2.2 Containers και VMs

Είναι τέλικα οι containers ένα υποκατάστατο των Εικονικών Μηχανών ή έστω ένα υποσύνολο των VMs; Κανένα από αυτά δεν είναι στην πραγματικότητα αληθινό. Εξυπηρετούν παρόμοιο σκοπό; Όχι ακριβώς. Ποια είναι η σχέση μεταξύ αυτών των δύο; Η πραγματική μαγεία, και η γοητεία των μηχανικών είναι ότι τίποτα δεν αντιμετωπίζεται ως δυαδικό. Οι μηχανικοί κατασκευάζουν την τεχνολογία κατ' εικόνα της φύσης. Αναμφισβήτητα αυτή η προσέγγιση έχει και τα μειονεκτήματά της, αφού η φύση και οι άνθρωποι δεν είναι αλάνθαστοι. Η τεχνολογία προσαρμόζεται και εξελίσσεται ανάλογα με τις ανάγκες, τις συνθήκες και το περιβάλλον κάθε εποχής, ώστε να αξιοποιεί στο έπακρο κάθε κατάσταση. Ως εκ τούτου, μπορούμε να θεωρήσουμε ότι τα containers και τα VMs αλληλοσυμπληρώνονται. Η εικονικοποίηση επιτρέπει την ταυτόχρονη εκτέλεση πολλαπλών λειτουργικών συστημάτων σε ένα μόνο σύστημα υλικού. Οι containers μοιράζονται τον ίδιο πυρήνα του λειτουργικού συστήματος και απομονώνουν τις διεργασίες της εφαρμογής από το υπόλοιπο σύστημα. Λειτουργούν σε διαφορετική στοίβα της πυραμίδας του υπολογιστή και αν χρησιμοποιηθούν μαζί παρέχουν μεγάλη ευελιξία στην ανάπτυξη και διαχείριση εφαρμογών.

Όσο ανεξήγητο και μυστηριώδες και αν φαίνεται έχουν εντελώς διαφορετικές τεχνικές υλοποίησης, προς έναν παρόμοιο τελικό στόχο. Στη συνέχεια θα αναλύσουμε τον τρόπου που υλοποιούνται οι containers και θα αντιληφθούμε τις όποιες διαφορές σε σχέση με τις εικονικές μηχανές.

2.3 Πυρήνας Linux και Containers

Οι Containers δεν είναι τίποτε παραπάνω από διεργασίες Linux με κάποιες επιπλέον ρυθμίσεις. Είναι ένα συνονθύλευμα από λειτουργίες του Linux όπως τα namespaces



Σχήμα 2.3: *Containers vs Virtual Machines*

και τα cgroups. Η τρέχουσα φιλοσοφία της κοινότητας του πυρήνα Linux είναι ότι ο πυρήνας Linux θα πρέπει να παρέχει ένα σωρό διαφορετικές τεχνολογίες, από πειραματικές έως πολύ ώριμες, επιτρέποντας στους χρήστες να αναμειγνύουν αυτές τις τεχνολογίες με δημιουργικούς, νέους τρόπους. Και, αυτό ακριβώς κάνει μια μηχανή container (Docker, Podman, CRI-O, κ.λπ.) - αξιοποιεί τις τεχνολογίες του πυρήνα για να δημιουργήσει, αυτό που εμείς οι άνθρωποι αποκαλούμε, containers.

Οι containers δεν υπάρχουν στον κόσμο του Linux. Κανείς δεν όρισε ποτέ τι είναι ένας container ή δημιούργησε μια δομή που ονομάζεται container στον πυρήνα. Είναι απλώς ένας συνδυασμός διαφορετικών τεχνολογιών που παρέχονται από τον πυρήνα του Linux. Η έννοια του container είναι μια κατασκευή δική μας, όχι μια κατασκευή του πυρήνα. Είναι σημαντικό να τονίσουμε ότι οι διεργασίες των containers είναι κανονικές διεργασίες Linux οι οποίες απομονώνονται χρησιμοποιώντας τεχνολογίες του πυρήνα όπως τα namespaces, το selinux και τα cgroups. Αυτό περιγράφεται μερικές φορές ως ”sand boxing” ή ”απομόνωση” ή ως ”ψευδαίσθηση” της εικονικοποίησης. Όλες οι διεργασίες ζουν δίπλα-δίπλα, είτε πρόκειται για κανονικές διεργασίες Linux, είτε για διεργασίες container.

2.3.1 Χώροι Ονομάτων

Οι χώροι ονομάτων είναι ένα χαρακτηριστικό του πυρήνα Linux που εισήχθη το 2002 με το Linux 2.4.19. Ένας χώρος ονομάτων περιβάλλει έναν καθολικό πόρο του συστήματος με μια αφαίρεση που κάνει τις διεργασίες εντός του χώρου ονομάτων να πιστεύουν ότι έχουν τη δική τους απομονωμένη περίπτωση του πόρου αυτού. Οι αλλαγές στον καθολικό πόρο είναι ορατές σε άλλες διεργασίες που είναι μέλη του ίδιου χώρου

ονομάτων, αλλά είναι αόρατες σε άλλες διεργασίες. Μια χρήση των χώρων ονομάτων είναι η υλοποίηση των containers [4]. Ένα από τα θεμελιώδη μέρη ενός container είναι τα namespaces. Η έννοια των χώρων ονομάτων είναι να περιορίζουν τι μπορούν να δουν οι διεργασίες και σε ποια μέρη του συστήματος μπορούν να έχουν πρόσβαση, όπως άλλες διεπαφές δικτύου ή διεργασίες. Η αφαίρεση του χώρου ονομάτων του πυρήνα επιτρέπει σε διαφορετικές ομάδες διεργασιών να έχουν διαφορετικές απόψεις του συστήματος.

Για παράδειγμα, ο χώρος ονομάτων Network ενθυλακώνει πόρους συστήματος που σχετίζονται με τη δικτύωση, όπως διασυνδέσεις δικτύου (π.χ. wlan0, eth0), πίνακες δρομολόγησης. Μια διεργασία σε ένα χώρο ονομάτων έχει εντελώς διαφορετική διαμόρφωση δικτύου από μια άλλη διεργασία που ζει σε διαφορετικό χώρο ονομάτων. Επί του παρόντος υπάρχουν 7 τύποι χώρων ονομάτων Cgroup, IPC, Network, Mount, PID, User, UTS. Οι χώροι ονομάτων δεν είναι κάποιο πρόσθετο χαρακτηριστικό ή βιβλιοθήκη που πρέπει να εγκαταστήσουμε αλλά παρέχονται από τον ίδιο τον πυρήνα του Linux και αποτελούν ήδη προαπαιτούμενο για την εκτέλεση οποιασδήποτε διεργασίας στο σύστημα. Σε κάθε δεδομένη στιγμή, κάθε διεργασία P ανήκει ακριβώς σε μία περίπτωση κάθε τύπου namespace - έτσι όταν χρειάζεται, ας πούμε, να ενημερώσει τον πίνακα δρομολόγησης στο σύστημα, το Linux της δείχνει το αντίγραφο του πίνακα δρομολόγησης του namespace στο οποίο ανήκει εκείνη τη στιγμή. Η διεργασία init πηγαίνει πάντα στον προεπιλεγμένο χώρο ονομάτων. Η προεπιλεγμένη συμπεριφορά του Kernel είναι να θέτει στο ίδιο namespace όλες τις κλωνοποιημένες διεργασίες που παράγονται από την init. Τότε, πώς μπορούμε πραγματικά να αλλάξουμε τον χώρο ονομάτων ενός συγκεκριμένου τύπου για μια νέα διεργασία; Για τον σκοπό αυτό τα namespaces παρέχουν ένα σύνολο διεπαφών χρήστη (API) και συγκεκριμένα τις λειτουργίες: clone(), unshare(), setns(). Οι παραπάνω λειτούργιες μας επιτρέπουν να αλλάξουμε χώρο ονομάτων καθώς δημιουργούμε μια καινούργια διεργασία ή να μεταβούμε σε διαφορετικό namespace στην ίδια διεργασία.

2.3.2 Ομάδες Ελέγχου

Ο μηχανισμός ομάδων ελέγχου, που συνήθως αναφέρεται ως ”cgroups”, είναι ένα χαρακτηριστικό του πυρήνα Linux που επιτρέπει την οργάνωση διεργασιών σε ιεραρχικές ομάδες, των οποίων η χρήση διαφόρωντύπων πόρων μπορεί στη συνέχεια να περιοριστεί

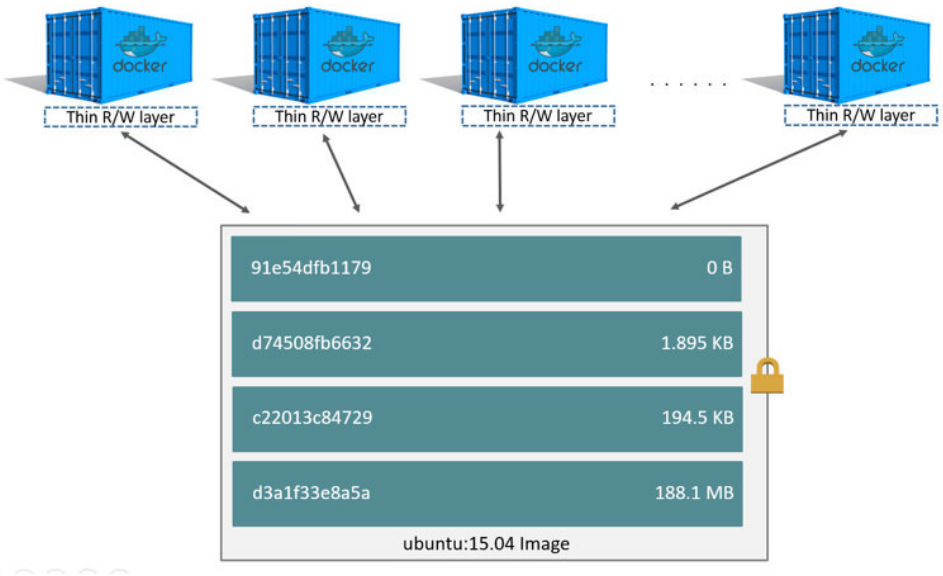
και να παρακολουθείται. Η διεπαφή cgroup του πυρήνα παρέχεται μέσω ενός ψευδοσυστήματος αρχείων που ονομάζεται cgroupfs. Η ομαδοποίηση υλοποιείται στον πυρήνα του κώδικα του πυρήνα cgroup, ενώ η παρακολούθηση των πόρων και τα όρια υλοποιούνται σε ένα σύνολο υποσυστημάτων ανά τύπο πόρου (μνήμη, CPU κ.ο.κ.). Με τον όρο cgroup αναφερόμαστε σε μια συλλογή διεργασιών που δεσμεύονται από ένα σύνολο ορίων ή παραμέτρων που ορίζονται μέσω του συστήματος αρχείων cgroup. Με λίγα λόγια, οι ομάδες cgroups περιορίζουν την ποσότητα των πόρων που μπορεί να καταναλώσει μια διεργασία.

2.4 Εικόνες Περιεκτών

Πολλά άρθρα αναφέρουν την τεχνολογία των containers ως άξιο υποκατάστατο των εικονικών μηχανών. Αντιθέτως, παρόλο που τα containers προσφέρουν κάποιου είδους απομόνωση και προστασία (όχι στον βαθμό των VMs), εμφανίστηκαν κυρίως ως τεχνολογία για να καλύψουν την ανάγκη επαναχρησιμοποίησης και αναπαραγωγιμότητας των εφαρμογών λογισμικού. Οι εικονικές μηχανές προσφέρουν μεγάλη απομόνωση και οι containers μεγάλη ευελιξία. Στο 2.3 το κέντρο της προσοχής μας ήταν ο τρόπος με τον οποίο οι containers προστατεύονται χρησιμοποιώντας namespaces και cgroups. Για να τα καταστήσουν ευέλικτα, μεταβιβάσιμα και συνδέσιμα σε κάθε πιθανό περιβάλλον, οι ομάδες μηχανικών λογισμικού χρειάζονταν έναν τρόπο να παγώσουν την κατάσταση των containers, ώστε να μπορούν αργότερα να στείλουν αυτή την παγωμένη κατάσταση στους χρήστες που θα τρέξουν τις εφαρμογές που περιέχονται σε αυτά.

Αυτή η παγωμένη έκδοση ενός container ονομάζεται εικόνα του container και είναι μια έννοια που εισήγαγε για πρώτη φορά το Docker. Χρησιμοποιώντας μια μεταφορά στον χώρο του προγραμματισμού, αν μια εικόνα είναι μια κλάση, τότε ένας container είναι ένα αντικείμενο χρόνου εκτέλεσης της κλάσης αυτής. Μια εικόνα ενός container είναι ουσιαστικά μια στατική αναπαράσταση που καθορίζει την εκτέλεσή του. Αυτό σημαίνει ότι περιέχει πληροφορίες τόσο για τη δομή του συστήματος αρχείων του container όσο και για τις ρυθμίσεις και διαμορφώσεις που σχετίζονται με την εκτέλεσή του. Με λίγα λόγια, ένα container image είναι ένα αμετάβλητο αρχείο το οποίο ουσιαστικά περιγράφει ένα στιγμιότυπο του container. Το σύστημα αρχείων της εικόνας δημιουργείται με

τη συσσώρευση μιας λίστας επιπέδων μόνο για ανάγνωση, τα οποία - στην περίπτωση του Docker - μπορεί να αντιπροσωπεύουν οδηγίες που δίνονται σε ένα ”Dockerfile”. Τα στρώματα είναι μόνο για ανάγνωση και κάθε στρώμα είναι απλώς ένα σύνολο διαφορών από το προηγούμενο στρώμα, καθώς στοιβάζονται το ένα πάνω στο άλλο.

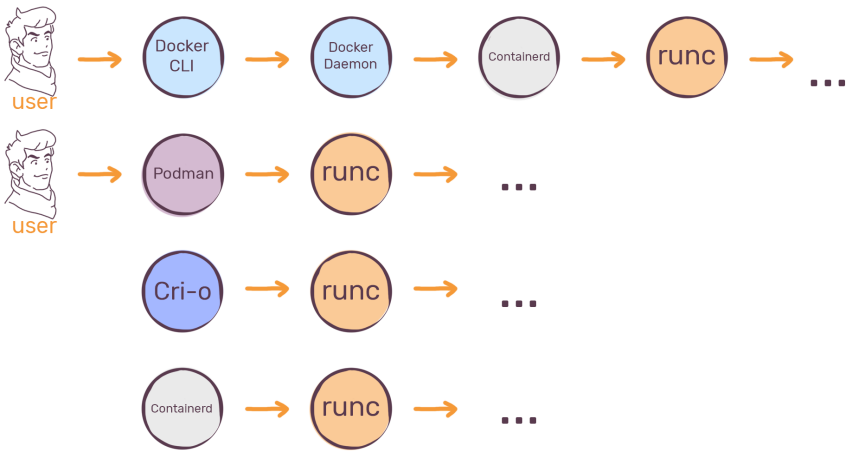


Σχήμα 2.4: *Docker Containers με το δικό τους εγγράψιμο επίπεδο και την ίδια υποκείμενη εικόνα [5]*

2.5 Χρόνος Εκτέλσης των Περιεκτών

Υπάρχουν πολλοί τρόποι εκτέλεσης των containers και ο πιο γνωστός είναι, χωρίς αμφιβολία, το Docker, το οποίο εκτόξευσε στα ύψη την υιοθέτηση αυτών. Υπάρχουν επίσης το Podman και το Buildah. Το CRI-O είναι επίσης μια επιλογή που λειτουργεί ως ένα υψηλότερο επίπεδο εκτέλεσης containers, το οποίο έχει γραφτεί επίτηδες για να χρησιμοποιηθεί με το Kubernetes CRI. Όλα τα παραπάνω ονομάζονται container runtimes με κοινό σκοπό που είναι η εκτέλεση των containers. Για να ξεκαθαρίσουμε λίγο τα πράγματα, ας μιλήσουμε για το εργαλείο που βρίσκεται στον πυρήνα του Docker, του Podman, του CRI-O και του Containerd: το runc. Το runc είναι το αρχικό container runtime πάνω στο οποίο αναπτύσσονται τα περισσότερα από τα εργαλεία.

runc Το runc είναι ένας πελάτης γραμμής εντολών για την εκτέλεση εφαρμογών συσκευασμένων σύμφωνα με τη μορφή Open Container Initiative (OCI)[7]



Σχήμα 2.5: *Οι περισσότεροι από τους χρόνους εκτέλεσης των containers καλούν τελικά το runc [6]*

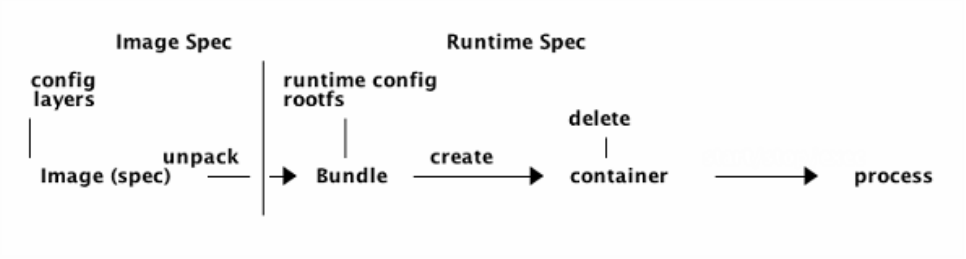
Το runc δεν έχει την έννοια των "εικόνων", όπως το Podman ή το Docker. Δεν μπορούμε απλά να εκτελέσουμε την εντολή *runc run ubuntu:latest*. Αντ' αυτού, το runc περιμένει από εμας να παρέχουμε ένα πακέτο χρόνου εκτέλεσης OCI, το οποίο είναι βασικά ένα root σύστημα αρχείων και ένα αρχείο ρυθμίσεων. Ο ορισμός ενός bundle ή αλλιώς δέσμης αφορά μόνο τον τρόπο με τον οποίο ένας container και τα δεδομένα διαμόρφωσής του αποθηκεύονται σε ένα τοπικό σύστημα αρχείων, ώστε να μπορούν να καταναλωθούν από ένα συμβατό runtime.

Ένα τυποποιημένο container bundle περιέχει όλες τις πληροφορίες που απαιτούνται για τη φόρτωση και την εκτέλεση ενός container. Αυτό περιλαμβάνει τα ακόλουθα:

- *config.json*: περιέχει δεδομένα διαμόρφωσης. Αυτό το απαραίτητο αρχείο πρέπει να βρίσκεται στη ρίζα του καταλόγου του πακέτου και πρέπει να ονομάζεται config.json.
- *container's root filesystem*: ο κατάλογος που αναφέρεται από το root.path, εάν η ιδιότητα αυτή έχει οριστεί στο config.json.

Τα Docker, Podman, CRI-O είναι υπεύθυνα για την αποσυμπίεση μιας εικόνας container σε ένα πακέτο χρόνου εκτέλεσης, το οποίο αποτελείται από τα παραπάνω δύο στοιχεία. Στη συνέχεια, το runc αναλαμβάνει και εκτελεί τον αντίστοιχο container. Για περισσότερες πληροφορίες σχετικά με τον τρόπο μετατροπής μιας OCI εικόνας container σε ένα OCI runtime bundle, δηλάδη, σε μία δέσμη που περιλαμβάνει όλα τα

απαραίτητα στοιχεία για την εκτέλεση ενός περιέκτη ανατρέξτε στην επίσημη τεκμηρίωση: [8]



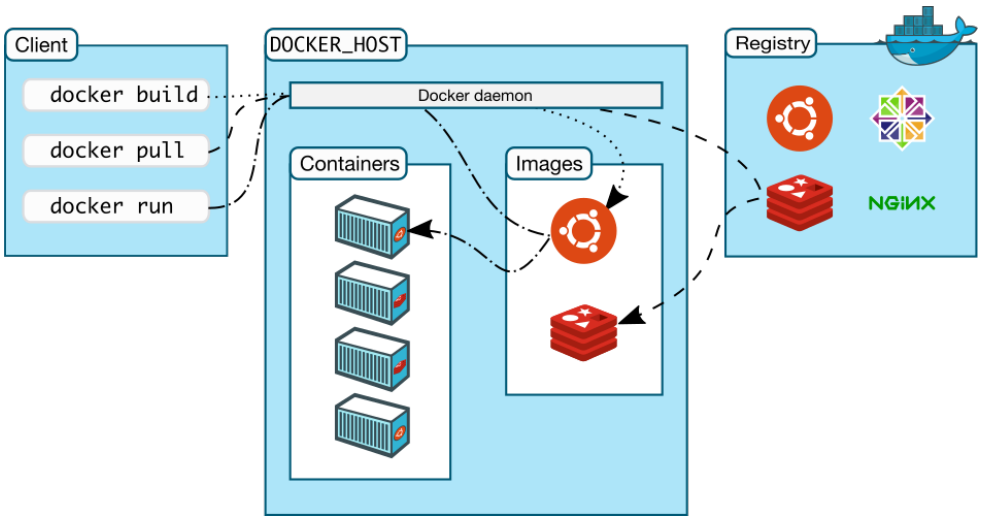
Σχήμα 2.6: *Από μια εικόνα container σε μια εκτελούμενη διεργασία [9].*

2.6 Αρχιτεκτονική του Docker

Σύμφωνα με τα λόγια της εταιρείας που την υποστηρίζει, της Docker Inc., το Docker είναι μια ανοικτή πλατφόρμα για την ανάπτυξη, την αποστολή και την εκτέλεση εφαρμογών οι οποίες διαχωρίζονται από την υποκείμενη υποδομή, επιτρέποντας έτσι την εύκολη και γρήγορη παράδοση λογισμικού και τη διαχείριση της υποδομής. Αυτό επιτυγχάνεται, βασικά, παρέχοντας τη δυνατότητα συσκευασίας και εκτέλεσης εφαρμογών σε containers μέσω της Docker Engine, της μηχανής εκτέλεσης containers. Επιπλέον, το Docker παρέχει εργαλεία και μια πλατφόρμα για τη διαχείριση του κύκλου ζωής των containers, με στόχο να βοηθήσει την ανάπτυξη των εφαρμογών και να καταστήσει τον container τη μονάδα διανομής και δοκιμής αυτών των εφαρμογών και, τέλος, να τις αναπτύξει σε περιβάλλοντα παραγωγής, ανεξάρτητα από το αν αυτό είναι ένα τοπικό κέντρο δεδομένων, ένας πάροχος cloud ή ένα υβρίδιο των δύο.

2.7 Kubernetes

Οι Containers παρέχουν έναν τρόπο για την εκτέλεση εφαρμογών μέσα σε απομονωμένα, αναλλοίωτα και αναπαραγώγιμα περιβάλλοντα. Τα τελευταία χρόνια η διαδικασία εκκίνησης ενός container έχει γίνει τετριμμένη και αμελητέα, επειδή κάθε προγραμματιστής το κάνει σε τακτική βάση. Η άνοδος των containers και της τεχνολογίας μικρουπηρεσιών είχε ως αποτέλεσμα εφαρμογές που πλέον περιλαμβάνουν εκατοντάδες ή μερικές φορές χιλιάδες containers. Η διαχείριση αυτών των φορτίων σε πολλαπλά



Σχήμα 2.7: *Αρχιτεκτονική Docker [10]*

περιβάλλοντα, η εκτέλεση ελέγχων υγείας ή η ανάκαμψη από μια αποτυχία με τη χρήση σεναρίων και αυτοσχέδιων εργαλείων μπορεί να είναι πραγματικά πολύπλοκη ή και αδύνατη. Αυτό το σενάριο αύξησε τη ζήτηση για έναν κατάλληλο τρόπο διαχείρισης ενός μεγάλου αριθμού containers. Ο Κυβερνήτης είναι ένα εργαλείο ενορχήστρωσης των container ανοικτού κώδικα που ικανοποιεί την ανάγκη αφαίρεσης του ελέγχου, της εποπτείας και της διαχείρισης πολυάριθμων container. Το όνομα Kubernetes προέρχεται από τα ελληνικά, που σημαίνει τιμονιέρης ή πιλότος. Το K8s ως συντομογραφία προκύπτει από την καταμέτρηση των οκτώ γραμμάτων μεταξύ του ”K” και του ”s”. Η Google έδωσε το έργο Kubernetes σε ανοικτή διάθεση το 2014. [11].

Ο Κυβερνήτης εγγυάται υψηλή διαθεσιμότητα ή μηδενικό χρόνο διακοπής λειτουργίας, το οποίο με απλά λόγια σημαίνει ότι μια εφαρμογή που αναπτύσσεται στον Κυβερνήτη θα είναι πάντα προσβάσιμη από τον χρήστη. Για παράδειγμα, εάν ένας container πέσει, ένα άλλος container πρέπει να ξεκινήσει. Επιπλέον, ο Κυβερνήτης διεκδικεί επεκτασιμότητα ανάλογα με τη ζήτηση, παρέχοντας τρόπους δυναμικής κλιμάκωσης των εφαρμογών. Έτσι, προσφέρει υψηλές επιδόσεις και υψηλούς ρυθμούς απόκρισης. Ο Κυβερνήτης είναι πανταχού παρόν στο σημερινό τοπίο του cloud computing κυρίως επειδή η πολυπλοκότητα κρύβεται από απλές αφαιρέσεις. Αυτό που κάνει τον Κυβερνήτη τόσο ξεχωριστό είναι ότι λειτουργεί με βάση τη δήλωση πρόθεσης, αντί να εκτελεί προστακτικές εντολές. Έμεις ως χρήστες ορίζουμε δηλωτικά την επιθυμητή κατάσταση για το σύστημα και το σύστημα θα παρακολουθεί συνεχώς τον εαυτό του και θα προσπαθεί να επιτύχει αυτή την κατάσταση. Αυτό καταργεί αυτόματα μια δύσκολη

απαίτηση για γνώση των εσωτερικών λειτουργιών του Κυβερνήτη.

2.8 Kubeflow

Οι περισσότερες βιομηχανίες που εργάζονται με μεγάλες ποσότητες δεδομένων έχουν αναγνωρίσει την αξία της τεχνολογίας Μηχανικής Μάθησης. Με την άντληση πληροφοριών από αυτά τα δεδομένα - συχνά σε πραγματικό χρόνο - οι οργανισμοί είναι σε θέση να εργάζονται πιο αποτελεσματικά ή να αποκτούν πλεονέκτημα έναντι των ανταγωνιστών τους. Μια κοινή πρακτική σήμερα είναι η ανάπτυξη εργασιών ΜΜ ως containers που διαχειρίζεται ο ενορχηστρωτής Kubernetes. Η δημιουργία ενός μοντέλου ΜΜ που μπορεί να προβλέψει αυτό που θέλουμε να προβλέψει από ένα σύνολο δεδομένων που κρατάμε εμείς οι ίδιοι, είναι εύκολη. Ωστόσο, η δημιουργία ενός μοντέλου ΜΜ που είναι αξιόπιστο, γρήγορο, ακριβές και μπορεί να χρησιμοποιηθεί από μεγάλο αριθμό χρηστών αποτελεί επίπονη διαδικασία. Η δημιουργία ενός συστήματος ΜΜ που θα λειτουργεί συνεχώς στην παραγωγή είναι μια σημαντικά βαριά δουλειά για έναν επιστήμονα δεδομένων. Η παρακολούθηση, η αποσφαλμάτωση και η αναπαραγωγιμότητα των μοντέλων Μηχανικής Μάθησης είναι επίσης επαχθείς διαδικασίες γι' αυτούς. Πολλές ομάδες διαθέτουν επιστήμονες δεδομένων και ερευνητές ΜΜ που μπορούν να κατασκευάσουν μοντέλα τελευταίας τεχνολογίας, αλλά η διαδικασία κατασκευής και ανάπτυξης μοντέλων ΜΜ είναι εξ' ολοκλήρου χειροκίνητη. Αυτό μπορεί να είναι επαρκές όταν τα μοντέλα σπάνια αλλάζουν. Στην πράξη, τα μοντέλα συχνά καταρρέουν όταν αναπτύσσονται στον πραγματικό κόσμο. Τα μοντέλα αποτυγχάνουν να προσαρμοστούν σε αλλαγές στη δυναμική του περιβάλλοντος.

Το Kubeflow είναι μια εργαλειοθήκη μηχανικής μάθησης για τον Κυβερνήτη που προσπαθεί να κάνει την ανάπτυξη ροών εργασίας μηχανικής μάθησης (ΜΜ) στον Κυβερντήτη απλή, φορητή και επεκτάσιμη. Στόχος του είναι να επιτρέψει τη χρήση μιας pipeline τεχνικής, ή αλλιώς τεχνικής σωλήνωσης στην επιστήμη της Μηχανικής Μάθησης για την ενορχήστρωση περίπλοκων ροών εργασίας που εκτελούνται στον Κυβερνήτη. [12]. Ένα Machine Learning Pipeline είναι μια ροή εργασίας, δηλαδή μια σειρά από διαδικασίες που συνδέονται αλυσιδωτά μεταξύ τους και που τελικά παράγουν ένα έτοιμο προς χρήση, μοντέλο μηχανικής μάθησης. Τα pipelines αποτελούνται από έναν αριθμό βημάτων, καθένα από τα οποία εκτελείται ως ανεξάρτητη διαδικασί-

α σε σχέση με τα υπόλοιπα. Κάθε βήμα εκτελεί μια συγκεκριμένη εργασία μιας ροής εργασίας μηχανικής μάθησης, π.χ.: προεπεξεργασία/καθαρισμός ενός συνόλου δεδομένων, εκπαίδευση ενός μοντέλου, παραγωγή προβλέψεων σε ένα υποσύνολο δοκιμής και εξυπηρέτηση ενός μοντέλου. Φυσικά, τέτοια pipelines μπορούν να εκτελούνται τοπικά, σε ένα μόνο μηχάνημα. Ωστόσο, αν κάποιος θέλει να αξιοποιήσει περισσότερα από όσα μπορεί να προσφέρει ένα μόνο μηχάνημα, πρέπει να τρέξει τέτοιες σωληνώσεις κατανεμημένα (π.χ.: σε ολόκληρες συστάδες, στον Κυβερνήτη).

Με το Kubeflow, οι επιστήμονες δεδομένων και οι μηχανικοί είναι πλέον σε θέση να αναπτύξουν έναν πλήρες pipeline που αποτελείται από τμηματικά βήματα. Αυτά τα τμηματοποιημένα βήματα στο Kubeflow είναι χαλαρά συνδεδεμένα στοιχεία ενός Machine Learning Pipeline κάθε ένα από τα οποία θα εκτελεστεί στον Κυβερνήτη. Το Kubeflow καταφέρνει να φέρει πολύ κόντα ή και να ενώσει δύο φάσεις που έως τώρα έιχαν τεράστιο κενό μεταξύ τους, την φάση της ανάπτυξης (development) και της παραγωγής (production). Χωρίς το Kubeflow οι χρήστες καλούνται να δουλέψουν στον τοπικό τους υπολογιστή καθώς αναπτύσσουν τα μοντέλα τους και έπειτα με επίπονες, κοστοβόρες και χρονοβόρες διαδικασίες αλλά και πολλές τροποποίησεις προσπαθούν να εναρμοσίσουν το μοντέλο τους στο περιβάλλον παραγωγής. Το Kubeflow καταφέρνει να μειώσει το χάσμα μεταξύ του επιπέδου ανάπτυξης και του επιπέδου παραγωγής, ενοποιώντας τα δύο περιβάλλοντα σε ένα, προσφέροντας όλα τα απαραίτητα εργαλεία για κάθε στάδιο.

2.9 Kale & Rok

Το KALE (Kubeflow Automated pipeLines Engine) είναι ένα έργο που στοχεύει στην απλοποίηση της εμπειρίας της Επιστήμης Δεδομένων κατά την ανάπτυξη ροών εργασίας στο Kubeflow.

Το Kubeflow είναι μια εξαιρετική πλατφόρμα για την ενορχήστρωση σύνθετων ροών εργασίας πάνω από τον Κυβερνήτη και το Kubeflow Pipelines παρέχει τα μέσα για τη δημιουργία επαναχρησιμοποιήσιμων στοιχείων που μπορούν να εκτελεστούν ως μέρος των ροών εργασίας. Ο αυτοεξυπηρετούμενος χαρακτήρας του Kubeflow το καθιστά εξαιρετικά ελκυστικό για χρήση στην Επιστήμη Δεδομένων, καθώς παρέχει εύκολη πρόσβαση σε προηγμένη ενορχήστρωση κατανεμημένων εργασιών, δυνατότητα επα-

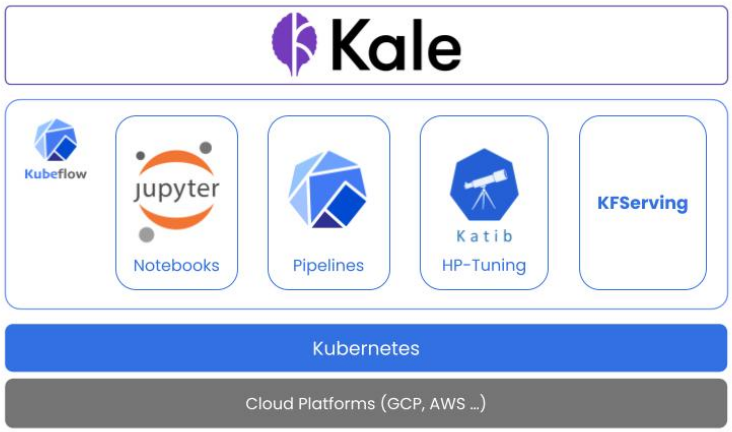
Σχήμα 2.8: *Το λογότυπο του Kale*

ναχρησιμοποίησης δομικών στοιχείων, Jupyter Notebooks, πλούσια UIs και πολλά άλλα. Παρόλα αυτά, η ανάπτυξη και η συντήρηση των ροών εργασίας Kubeflow μπορεί να είναι δύσκολη για τους επιστήμονες δεδομένων, οι οποίοι μπορεί να μην είναι ειδικοί στην εργασία με πλατφόρμες ενορχήστρωσης και σχετικά SDK. Επιπλέον, η επιστήμη δεδομένων συχνά περιλαμβάνει διαδικασίες εξερεύνησης δεδομένων, επαναληπτικής μοντελοποίησης και διαδραστικών περιβαλλόντων (κυρίως Jupyter notebook)[13].

Το Kale γεφυρώνει αυτό το κενό παρέχοντας ένα απλό UI για τον ορισμό ροών εργασίας Kubeflow Pipelines απευθείας από τη διεπαφή JupyterLab, χωρίς να χρειάζεται να αλλάξουμε ούτε μία γραμμή κώδικα. Το Kale SDK παρέχει τον απλούστερο δυνατό τρόπο μετατροπής οποιουδήποτε κώδικα Python σε πλήρως αναπαραγώγιμες εκτελέσεις Kubeflow Pipelines (KFP) χωρίς να αλλάζει ο πηγαίος κώδικας [14]. Ουσιαστικά, το Kale προσφέρει:

1. Μια επέκταση *Jupyter UI* που επιτρέπει τη μετατροπή των Jupyter Notebooks σε Kubeflow Pipelines. Το Kale καθιστά αυτό εφικτό επιτρέποντας στους χρήστες να σχολιάζουν τα κελιά κώδικα με συγκεκριμένες ετικέτες που υπαγορεύουν τα βήματα της ροής εργασίας και τις μεταξύ τους εξαρτήσεις. Στη συνέχεια, το Kale είναι υπεύθυνο για τη μετατροπή του μαρκαρισμένου notebook του χρήστη σε έναν λειτουργικό Kubeflow Pipeline, καθώς και για τη φροντίδα της διακίνησης δεδομένων μεταξύ των βημάτων.

2. Ένα *SDK*, για τη μετατροπή απλού κώδικα Python σε Kubeflow Pipelines. Το Kale Software Development Kit επιτρέπει στους χρήστες να γράφουν κώδικα Python, βασισμένο σε συναρτήσεις και να τον μετατρέπουν σε ένα Kubeflow Pipeline χωρίς να κάνουν σχεδόν καμία αλλαγή στον αρχικό πηγαίο κώδικα.



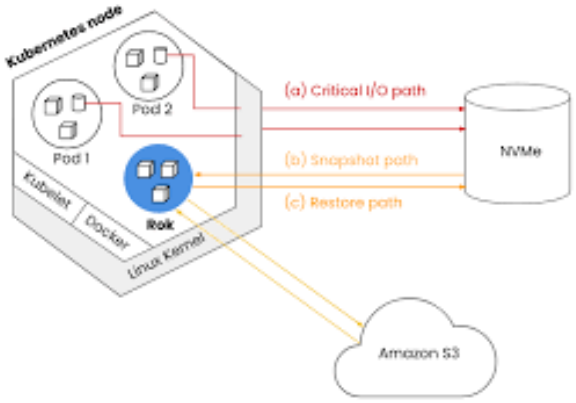
Σχήμα 2.9: *Το Kale λειτουργεί πάνω στο Kubeflow*

Πώς καταφέρνει όμως το Kale να κατασκευάσει τα απαιτούμενα Docker Images για κάθε δομικό στοιχείο του Pipeline; Στην πραγματικότητα, δεν το κάνει! Σε αυτό το σημείο μπαίνει στο παιχνίδι το Rok. Αντί να κατασκευάζει την εικόνα του δομικού στοιχείου, το Rok δημιουργεί στιγμιότυπα του περιβάλλοντος εργασίας του επιστήμονα δεδομένων (κώδικας + δεδομένα), το κλωνοποιεί και το επισυνάπτει στα Pods όπου θα εκτελεστεί κάθε δομικό στοιχείο. Η πλατφόρμα διαχείρισης και αποθήκευσης δεδομένων του Rok προσφέρει τα καλύτερα και από τους δύο κόσμους: την απόδοση της τοπικής αποθήκευσης και την ευελιξία της κοινής αποθήκευσης.

Το Rok μάς επιτρέπει να εκτελούμε stateful containers σε γρήγορο, τοπικό αποθηκευτικό χώρο NVMe on-prem ή στο cloud. Επιπλέον είμαστε σε θέση να δημιουργούμε στιγμιότυπα ολόκληρης της εφαρμογής, μαζί με τα δεδομένα της και να τα διανέμουμε αποτελεσματικά: σε μηχανήματα του ίδιου cluster Kubernetes ή σε διαφορετικές τοποθεσίες μέσω ενός αποκεντρωμένου δικτύου. Αυτή η υβριδική προσέγγιση στη δικτύωση container καθιστά απλή τη διαχείριση δεδομένων για εφαρμογές μηχανικής μάθησης και μεγάλων δεδομένων σε πραγματικό χρόνο, χωρίς να θυσιάζει την απόδοση [15].

Με αυτόν τον τρόπο, μπορούμε εύκολα να δημιουργήσουμε έτοιμες για παραγωγή ροές εργασίες ΜΜ με λειτουργίες ”point and click” και να επιστρέψουμε άμεσα σε οποιοδήποτε βήμα του pipeline για ταχεία αποσφαλμάτωση. Δυστυχώς, με αυτή την προσέγγιση, δεν μπορούμε να παράγουμε επαναχρησιμοποιήσιμα και διαμοιραζόμενα δομικά στοιχεία KFP, δηλαδή συγκεκριμένα βήματα του pipeline, καθώς ένα δομικό στοιχείο

θα πρέπει να συσκευάζεται εξ' ολοκλήρου σε μια εικόνα Docker. Σε αυτή η διπλωματική εργασία θα δούμε πως μπορούμε να κατασκευάσουμε διαφανώς από την πλεύρα του χρήστη εικόνες καθιστώντας τη διαδικασία κατασκευής επαναχρησιμοποιήσιμων και αναπαραγώγιμων δομικών στοιχείων KFP όσο το δυνατόν πιο απρόσκοπτη.



Σχήμα 2.10: *Rok Data Management*

3 Σχεδίαση

3.1 Επισκόπηση

Η ανάπτυξη και η συντήρηση ροών εργασίας Kubeflow είναι δύσκολη αποτελεί πρόκληση για τους επιστήμονες δεδομένων που δεν έχουν εμπειρία με πλατφόρμες ενορχήστρωσης και σχετικά SDKs. Το Kubeflow Pipelines Python SDK είναι ένα εξαιρετικό εργαλείο για την αυτοματοποίηση της δημιουργίας ροών εργασίας στον Κυβερνήτη, ειδικά όταν πρόκειται για πολύπλοκες ροές και περιβάλλοντα παραγωγής. Παρόλα αυτά, όταν παρουσιάζεται αυτή η τεχνολογία σε ερευνητές ML ή Data Scientists που δεν έχουν ισχυρή τεχνογνωσία σε θέματα τεχνολογίας λογισμικού, το KFP μπορεί να εκληφθεί ως πολύ περίπλοκο και δύσκολο στη χρήση. Τα Kubeflow Pipelines εξακολουθούν να έχουν ένα μεγάλο κενό στην αυτοματοποίηση και την αναπαραγωγιμότητα, καθώς δεν μπορούν ακόμη να αποσυνδέσουν τους χρήστες από τις έννοιες Container και Kubernetes. Συνεπώς, οι χρήστες καλούνται συχνά να κατασκευάσουν Docker Images για τα Pipeline τους. Η Επιστήμη Δεδομένων είναι συχνά θέμα πρωτοτυποποίησης νέων ιδεών, εξερεύνησης νέων δεδομένων και μοντέλων, γρήγορου και επαναληπτικού πειραματισμού. Σε αυτά τα σενάρια θα προτιμούσε κανείς να τρέξει απλώς κάποιο πρόχειρο κώδικα και να αναλύσει τα αποτελέσματα παρά να δημιουργήσει πολύπλοκες ροές εργασίας με ένα συγκεκριμένο SDK.

Το Kale μαζί με το Rok έκανε ένα πρώτο τεράστιο βήμα και γεφύρωσε αυτό το χάσμα παρέχοντας ένα απλό UI για τον ορισμό ροών εργασίας Kubeflow Pipelines απευθείας από τη διεπαφή του JupyterLab, χωρίς να χρειάζεται να αλλάξουμε ούτε μία γραμμή κώδικα. Επιπλέον, το Kale SDK παρέχει τον απλούστερο δυνατό τρόπο μετατροπής οποιουδή-

ποτε κώδικα Python σε πλήρως αναπαραγώγιμες εκτελέσεις Kubeflow Pipelines (KFP) χωρίς να αλλάξει ο πηγαίος κώδικας. Με τη βοήθεια των Rok Snapshots το Kale αναπαράγει και μεταφέρει το περιβάλλον εργασίας του χρήστη στα Kubernetes Pods όπου θα εκτελεστεί κάθε βήμα του Pipeline. Αυτή η προσέγγιση, ωστόσο, συνοδεύεται από ένα σημαντικό μειονέκτημα. Δεν επιτρέπει στο Kale να παράγει αυτόνομα, αναπαραγώγιμα και διαμοιραζόμενα KFP components, καθώς όλος ο απαιτούμενος κώδικας δεν είναι συσκευασμένος μεταξύ της βασικής εικόνας Docker του δομικού στοιχείου και του σημείου εισόδου του, όπως θα έπρεπε, αλλά είναι προσαρτημένος μέσω κλώνων των στιγμιοτύπων που δημιουργούνται από το Rok.

Προκειμένου να δημιουργηθούν κατάλληλα δομικά στοιχεία KFP, το πρώτο μέρος της παρούσας διπλωματικής εργασίας έρχεται σε επαφή με τα είδωλα περιεκτών, δηλαδή, τα container imageskαι τον τρόπο δημιουργίας τους όσο το δυνατόν πιο απρόσκοπτα, καθιστώντας τη διαδικασία δημιουργίας επαναχρησιμοποιούμενων δομικών στοιχείων KFP μια point and click λειτουργία. Συγκεκριμένα, αναλύουμε διεξοδικά και αποκαλύπτουμε σταδιακά έννοιες γύρω από τα container images. Αυτές οι πληροφορίες θα μας βοηθήσουν να σμιλέψουμε τον αλγόριθμό μας για την επέκταση τέτοιων εικόνων σε μη προνομιούχα περιβάλλοντα. Με αυτόν τον αλγόριθμο στα χέρια μας, θα σχεδιάσουμε και θα αναπτύξουμε έναν μηχανισμό δημιουργίας εικόνων που θα αποτελέσει τον ακρογωνιαίο λίθο για τη δημιουργία κατάλληλων KFP components.

Στο δεύτερο μέρος του κεφαλαίου, αναλύουμε σε βάθος τις έννοιες του Kubeflow γύρω από τη δημιουργία αναπαραγώγιμων pipeline και component. Εξετάζουμε τα τρωτά σημεία της προσέγγισης αυτής, πώς το Kale με το Rok κατάφερε να μας σώσει από πολλές δυσκολίες του Kubeflow και πώς η συμβολή μας απλοποίησε τη διαδικασία κατασκευής και ανάπτυξης διαμοιραζόμενων και αναπαραγώγιμων δομικών στοιχείων KFP.

Τελικά, αυτό το κεφάλαιο παρουσιάζει και ορίζει τον σκοπό και το σκεπτικό αυτής της διατριβής. Ξεκινώντας από τις ατέλειες των σημερινών προσεγγίσεων, δείχνουμε πως ο συλλογισμός μας μάς οδήγησε στην υλοποίηση του συστήματός μας.

3.2 Εικόνες Περιεκτών

Στις προηγούμενες ενότητες συζητήσαμε διεξοδικά τι είναι containers και πώς οι μηχανικοί αξιοποίησαν διάφορες τεχνολογίες του πυρήνα του Linux για να μιμηθούν τις

εικονικές μηχανές απομονώνοντας (namespaces) και περιορίζοντας (cgroups, capabilities, seccomp) τις διεργασίες συστήματος του Linux. Με άλλα λόγια, εξετάσαμε κυρίως τους containers από την οπτική γωνία της εικονικοποίησης. Ας θυμηθούμε τον ορισμό του container:

container Ένα εμπορευματοκιβώτιο είναι κάθε δοχείο ή περίβλημα για τη συγκράτηση ενός προϊόντος που χρησιμοποιείται στην αποθήκευση, τη συσκευασία και τη μεταφορά, συμπεριλαμβανομένης της ναυτιλίας. Τα πράγματα που φυλάσσονται μέσα σε ένα εμπορευματοκιβώτιο προστατεύονται από πολλές πλευρές, καθώς βρίσκονται μέσα στη δομή του.

Επικεντρωθήκαμε στους τρόπους προστασίας της συσκευασίας μας, στην απομόνωση και στην ασφάλεια της. Αυτό μπορεί να μας παραπλάνησε από τον αρχικό στόχο των container, ο οποίος υπερβαίνει την απλή προστασία του περιεχομένου. Ένα καλά σχεδιασμένο δοχείο θα παρουσιάζει επίσης ευκολία και χρηστικότητα, δηλαδή θα είναι εύκολο για τον εργαζόμενο να το ανοίξει ή να το κλείσει, να εισάγει ή να αφαιρέσει το περιεχόμενο και να χειριστεί το δοχείο κατά την αποστολή. Επιπλέον, ένας καλός περιέκτης θα έχει σχήμα και μέγεθος που ευνοεί την αποτελεσματική στοίβαξη και αποθήκευση. Σκοπός αυτής της ενότητας είναι να καταδείξει πώς τα Linux Containers καταφέρνουν να πληρούν και να ικανοποιούν τα περισσότερα (αν όχι όλα) από τα στοιχεία που το χαρακτηρίζουν.

Όπως ήδη αναφέραμε, οι containers έχουν μια παγωμένη κατάσταση στην οποία είναι αρχεία αποθηκευμένα στο δίσκο. Αυτό είναι αυτό που ονομάζουμε εικόνα container image. Με λίγα λόγια, ένα container image είναι ένα αμετάβλητο αρχείο το οποίο ουσιαστικά περιγράφει ένα στιγμιότυπο του container. Το σύστημα αρχείων της εικόνας δημιουργείται με τη στοίβαξη μιας λίστας από επίπεδα μόνο για ανάγνωση, χρησιμοποιώντας ένα σύστημα αρχείων union. Στη συνέχεια, όταν ο container ενσαρκώνεται από αυτή την εικόνα, ένα λεπτό στρώμα με δυνατότητα εγγραφής προστίθεται πάνω από τα στρώματα μόνο για ανάγνωση. Αυτό το στρώμα ονομάζεται επίσης ”container layer”.

Το Docker ήταν η πρώτη εταιρεία το 2013 που ήταν σε θέση να συσκευάσει έναν container. Αυτό επέτρεψε στους χρήστες να μετακινούν τις εικόνες αυτές μεταξύ μηχανών, γεγονός που σηματοδότησε τη γέννηση των εφαρμογών που βασίζονται σε containers.

Δηλαδή, το να μπορούμε να συσκευάσουμε την εφαρμογή μας με όλες τις απαιτούμενες βιβλιοθήκες, εξαρτήσεις και αρχεία σε μια αναπαραγώγιμη και κοινόχρηστη εικόνα με συγκεκριμένη έκδοση. Μέχρι σήμερα, υπάρχουν πολλαπλές εκδόσεις της μορφής ενός container image. Οι προγραμματιστές του Docker το 2013 δημιουργήσαν την πρώτη έκδοση την οποία αποφάσισαν να αποσύρουν το 2016 όταν και δημιούργησαν την έκδοση 2 (V2) και σχήματος 1. Μετά από πολλαπλές επαναλήψεις διαφόρων προσεγγίσεων βελτίωσης του μορφότυπου εικόνας, το σχήμα 2 κυκλοφόρησε για να αντικαταστήσει το υπάρχον σχήμα 1 το 2017. Η προδιαγραφή εικόνας V2 δόθηκε αργότερα στην Open Container Initiative (OCI) ως βάση για τη δημιουργία της προδιαγραφής εικόνας OCI.

3.2.1 Open Container Initiative

Η επιστροφή στον ορισμό του container βοηθάει πραγματικά να συνειδητοποιήσουμε γιατί οι μηχανικοί πήραν κάποιες αποφάσεις εξ αρχής. Τι θα συνέβαινε αν προσπαθούσαμε να μεταφέρουμε και να στοιβάζουμε ακανόνιστα σχήματα και μεγέθη κουτιών; Πιθανότατα όλεθρος. Η προδιαγραφή εικόνας του OCI έρχεται να λύσει αυτό το πρόβλημα ορίζοντας τον τρόπο με τον οποίο ένας container συσκευάζεται σε μια εικόνα.

Αυτή η προδιαγραφή ορίζει ένα OCI Image, το οποίο αποτελείται από ένα manifest αρχείο που περιγράφει βασικά χαρακτηριστικά της εικόνας, ένα δείκτη της εικόνας ή αλλιώς image index (προαιρετικό), ένα σύνολο από επίπεδα συστήματος αρχείων (filesystem layers) και ένα αρχείο με ρυθμίσεις διαμόρφωσης, δηλαδή ένα σύνολο από ρυθμίσεις σχετικά με την λειτουργία του container κατά τον χρόνο εκτέλεσης [16]. Ναι, αυτά είναι όλα όσα χρειαζόμαστε για να περιγράψουμε ένα container image. Όλα εκτός από τα filesystem layers είναι γραμμένα σε JavaScript Object Notation (JSON). Ο στόχος αυτής της προδιαγραφής είναι να επιτρέψει τη δημιουργία διαλειτουργικών εργαλείων για τη δημιουργία, τη μεταφορά και την προετοιμασία ενός container image για εκτέλεση.

1. Ένα *Image Index* είναι ένα ευρετήριο των manifest της εικόνας

2. Σε υψηλό επίπεδο, ένα *Image Manifest* είναι ένα έγγραφο που περιγράφει τα συστατικά που συνθέτουν μια εικόνα.

3. Ένα *Filesystem Layer* είναι ένα σύνολο αλλαγών που περιγράφει το τελικό σύστημα αρχείων ενός container

4. Ένα *Image Configuration* είναι ένα έγγραφο που καθορίζει τη διάταξη των επιπέδων του συστήματος αρχείων και το αρχείο διαμόρφωσης της εικόνας, δηλαδή τις κατάλληλες ρυθμίσεις και παραμέτρους για τη μετάφραση αυτής σε μία δέσμη χρόνου εκτέλεσης.

3.2.2 Ευρετήριο Εικόνων

Το ευρετήριο εικόνας (Image Index) είναι ένα manifest ανώτερου επιπέδου που παραπέμπει σε συγκεκριμένα manifest εικόνας, ιδανικά για μία ή περισσότερες πλατφόρμες. Ενώ η χρήση ενός ευρετηρίου εικόνων είναι προαιρετική για τους παρόχους εικόνων, οι καταναλωτές εικόνων, δηλαδή, τα λογισμικά που είναι σε θέση να διαβάσουν και να επεξεργαστούν τις εικόνες αυτές θα πρέπει να μπορούν να δεχτούν τέτοιου είδους αρχεία ως είσοδο [17]. Ενά παράδειγμα ενός index file παρατίθεται παρακάτω.

```
{
  "schemaVersion": 2,
  "manifests": [
    {
      "mediaType": "oci.image.manifest.v1+json",
      "digest": "45c",
      "size": 347,
      "annotations": {
        "org.opencontainers.image.ref.name": "latest"
      }
    }
  ]
}
```

Προφανώς, το ευρετήριο της εικόνας περιέχει μια λίστα από manifest αρχεία. Στην πραγματικότητα, χρησιμεύει ως ένας πίνακας δεικτών που παραπέμπει στα αρχεία αυτά. Φαίνεται ότι το ευρετήριο εικόνας είναι απλώς ένα manifest υψηλότερου επιπέδου, το οποίο περιέχει δείκτες σε πιο συγκεκριμένα manifest εικόνας που αφορούν διαφορετικές πλατφόρμες με συγκεκριμένο λειτουργικό σύστημα και αρχιτεκτονική. Όπως έχει

ήδη αναφερθεί, ο δείκτης εικόνας είναι πλήρως προαιρετικός. Το αναγνωριστικό που ταυτοποιεί μοναδικά τα manifest αυτά αρχεία είναι ένα hash digest που υπολογίζεται (εξάγεται από) από τα περιεχόμενα του αντίστοιχου manifest εικόνας.

Το *mediaType* είναι κάτι που θα συναντήσουμε πολλές φορές. Περιγράφει τον τύπο και τη μορφή του αντίστοιχου περιεχομένου και μπορεί να θεωρηθεί ως αναγνωριστικό του αντικειμένου, ενώ το digest είναι η μοναδική αναφορά σε αυτό. Για παράδειγμα, σε αυτή την περίπτωση ο τύπος πολυμέσων είναι:

```
application/vnd.oci.image.manifest.v1+json
```

Δηλαδή, το αρχείο στο οποίο δείχνει το ευρετήριο μας είναι ένα αρχείο manifest έκδοσης 1 μιας εικόνας OCI με μορφή JSON.

Επιπλέον το παρακάτω:

$$2a22f49c2fb14bae16fb907f6f36a9583a1f38f0044647077883221c7fa3645c$$

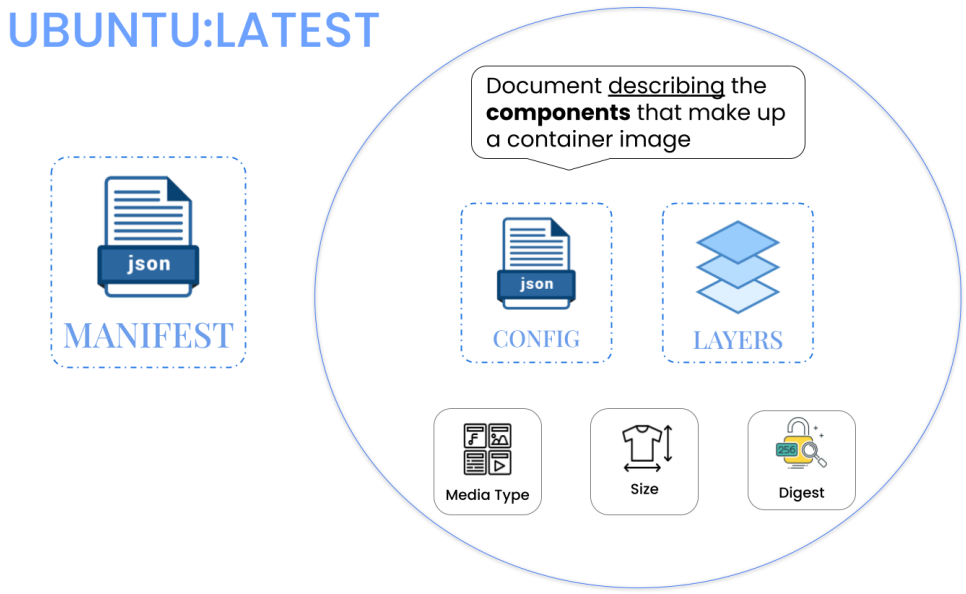
είναι το όνομα του αρχείου manifest και ταυτόχρονα το άθροισμα ελέγχου του περιεχομένου του, το οποίο εντοπίσαμε και στην αρχή κατά την εξερεύνεση της εικόνας busybox.

3.2.3 Manifest Εικόνων

Αναφερθήκαμε πολλές φορές στο Image Manifest. Είναι η κατάλληλη στιγμή για να λύσουμε το μυστήριο. Πρόκειται για ένα από τα πιο σημαντικά στοιχεία για ένα container image. Σε αντίθεση με το ευρετήριο εικόνας, το οποίο περιέχει πληροφορίες για ένα σύνολο εικόνων που μπορεί να καλύπτουν μια ποικιλία αρχιτεκτονικών και λειτουργικών συστημάτων, ένα image manifest παρέχει μια διαμόρφωση και ένα σύνολο από επίπεδα για μια ενιαία εικόνα για μια συγκεκριμένη αρχιτεκτονική και λειτουργικό σύστημα [18]. Στην πραγματικότητα, περιέχει μεταδεδομένα σχετικά με: 1) το αρχείο διαμόρφωσης και 2) τα διάφορα στρώματα της εικόνας. Αυτά τα μεταδεδομένα περιέχουν τα εξής:

- *mediaType*: Προσδιορίζει τη μορφή του αντίστοιχου πόρου. Για παράδειγμα, ένα στρώμα της εικόνας Docker έχει τον τύπο: *vnd.docker.image.rootfs.diff.tar.gzip*. Δηλαδή, είναι ένα συμπιεσμένο (gzip) tar αρχείο.

- *digest*: Άθροισμα ελέγχου του αντίστοιχου πόρου που υπολογίζεται με έναν αλγόριθμο κατακερματισμού όπως ο sha256
- *size*: Μέγεθος του πόρου



Σχήμα 3.1: *Η ανατομία μιας εικόνας περιεκτών: manifest αρχείο*

Συνεπώς, θα μπορούσαμε να πούμε ότι το manifest μιας εικόνας παρέχει μεταδεδομένα σχετικά με το αρχείο διαμόρφωσης και το σύνολο των layers που δημιουργούν το τελικό filesystem του container. Το *Image Index* μας αποκάλυψε τη θέση του *manifest*, η οποία θα μας αποκαλύψει τη θέση του αρχείου παραμετροποίησης/ρυθμίσεων και των στρωμάτων του αρχείου συστήματος. Αυτή τη φορά ερευνούμε τα περιεχόμενα του αρχείου 2a22f49c2fb14bae16fb907f6f36a9583a1f38f0044647077883221c7fa3645c (το αρχείο manifest σύμφωνα με το index.json).

```
{
  "schemaVersion": 2,
  "config": {
    "mediaType": "oci.image.config.v1+json",
    "digest": "417",
    "size": 575
  },
  "layers": [
```

```
    {
      "mediaType": "oci.image.layer.v1.tar+gzip",
      "digest": "2d1",
      "size": 772812
    }
  ]
}
```


Πράγματι, το Image Manifest περιέχει ένα πεδίο που ονομάζεται config και ένα άλλο που ονομάζεται layers. Το πρώτο μας λέει τη "διεύθυνση", δηλαδή το όνομα του αρχείου, το μέγεθος και τον τύπου του αρχείου παραμετροποιήσης. Το τελευταίο είναι ένας πίνακας με δείκτες σε όλα τα στρώματα της εικόνας του συστήματος αρχείων μαζί με το μέγεθος και τον τύπο τους.

3.2.4 Διαμόρφωση εικόνας

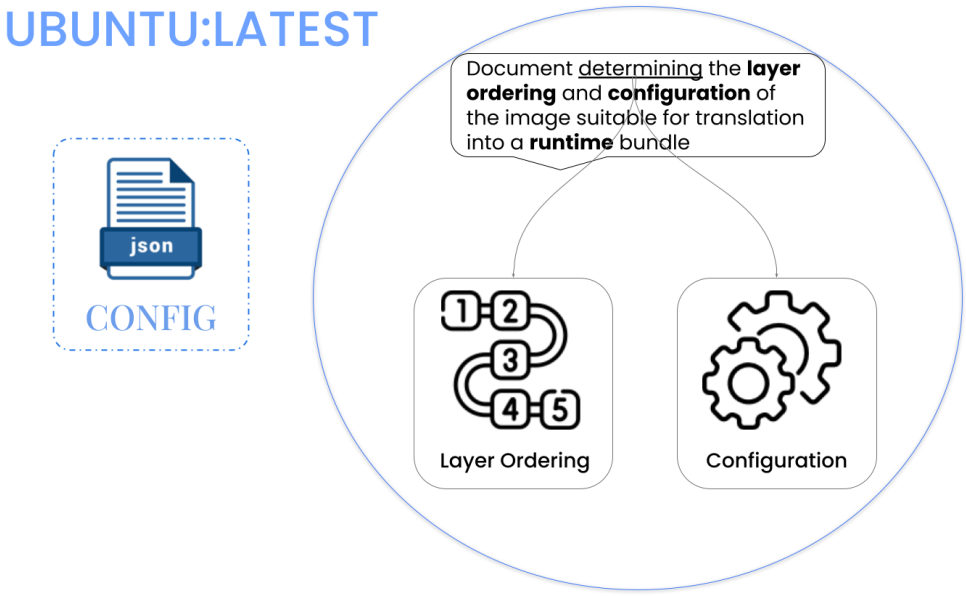
Τώρα είμαστε σε θέση να λάβουμε περισσότερες πληροφορίες σχετικά με το αρχείο διαμόρφωση της εικόνας, ή αλλιώς το configuration file.


```
{
  "created": "2022-04-14T02:29:36.517566461Z",
  "architecture": "amd64",
  "os": "linux",
  "config": {
    "Env": [
      "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
    ],
    "Cmd": [
      "sh"
    ]
  },
  "rootfs": {
    "type": "layers",
    "diff_ids": [
      "sha256:eb6b01329ebe73e209e44a616a0e16c2b8e91de6f719df9c35e6cdadadbe5965"
    ]
  },
  "history": [
```

```
    {
      "created": "2022-04-14T02:29:36.368193089Z",
      "created_by": "ce02 στο / "
    },
    {
      "created": "2022-04-14T02:29:36.517566461Z",
      "created_by": "/bin/sh -c #(nop) CMD [\"sh\"]",
      "empty_layer": true
    }
  ]
}
```


Στην κορυφή του JSON βρίσκουμε κάποια μεταδεδομένα, όπως η ημερομηνία *created* που δημιουργήθηκε, η αρχιτεκτονική (*architecture*) και το λειτουργικό *os* της εικόνας. Οι μεταβλητές περιβάλλοντος (Env) καθώς και οι εντολές που θα εκτελέσει ο container (cmd) βρίσκονται στην ενότητα *config* της διαμόρφωσης. Αυτή η ενότητα μπορεί να περιέχει ακόμη περισσότερες παραμέτρους, όπως τον κατάλογο εργασίας (WorkingDir), τον χρήστη που θα πρέπει να εκτελέσει τη διαδικασία του περιέκτη ή την εντολή με την οποία θα εκκινήσει ο container, δηλαδή, το entrypoint. Εν κατακλείδι, θα μπορούσαμε να σκεφτούμε ότι όλες αυτές οι παράμετροι που βρίσκονται στο configuration αρχείο μπορεί να έχουν προέλθει από ένα αντίστοιχο Dockerfile.

Τελικά, ένα OCI Image Configuration αποτελείται από μια διατεταγμένη συλλογή αλλαγών στο root σύστημα αρχείων (rootfs) και τις αντίστοιχες παραμέτρους εκτέλεσης για χρήση κατά τη διάρκεια του χρόνου εκτέλεσης του container [19]. Η διαμόρφωση υποδεικνύει ότι το rootfs χωρίζεται σε στρώματα. Αυτά τα στρώματα όμως σε τι διαφέρουν από τα αντίστοιχα που είδαμε στο αρχείο manifest; Τα diff_ids αναφέρονται στα αθροίσματα ελέγχου των **αποσυμπιεσμένων** αρχείων tar, δηλαδή, προκύπτουν από διαφορές μεταξύ του στρώματος γονέα και του στρώματος παιδιού. Γι' αυτό άλλωστε το αντίστοιχο πεδίο ονομάζεται diff_id. Αντίθετα, τα layers που βρίσκονται στο αρχείο manifest αφορούν στα αντίστοιχα **συμπιεσμένα** αρχεία. Γι' αυτό και το configuration file είναι άρρηκτα συνδεδεμένο με τον χρόνο εκτέλεσης ενός container.



Σχήμα 3.2: *Η ανατομία μίας εικόνας περιεκτών: configuration αρχείο*

3.2.5 Επίπεδα Εικόνας

Μέχρι τώρα έχουμε δει ότι μια εικόνα OCI αποτελείται από ένα αρχείο *Index*, ένα αρχείο *Manifest* και ένα αρχείο *Configuration*. Όλα αυτά είναι έγγραφα JSON τα οποία περιγράφουν κάποιες βασικές πληροφορίες σχετικά με την εικόνα όπως η ημερομηνία δημιουργίας, ο συγγραφέας, καθώς και ρυθμίσεις σχετικές με τον χρόνο εκτέλεσης όπως το σημείο εισόδου της, τα προεπιλεγμένα ορίσματα, η δικτύωση και οι τόμοι. Αυτή η δομή JSON παραπέμπει επίσης σε έναν κρυπτογραφικό κατακερματισμό κάθε στρώματος (layer) που χρησιμοποιείται από την εικόνα και παρέχει πληροφορίες ιστορικού για αυτά τα στρώματα.

Με άλλα λόγια, κάθε εικόνα περιλαμβάνει έγγραφα JSON που παρέχουν τις απαιτούμενες πληροφορίες που σχετίζονται με το χρόνο εκτέλεσης του container και έναν αριθμό στρωμάτων που συνθέτουν το τελικό σύστημα αρχείων αυτού.

Ειδικότερα, κάθε στρώμα αναπαριστά ένα σύνολο αλλαγών στο σύστημα αρχείων σε μορφή ενός tar archive, καταγράφοντας αρχεία που πρέπει να:

- Προστεθούν
- Αλλάξουν
- Διαγραφούν

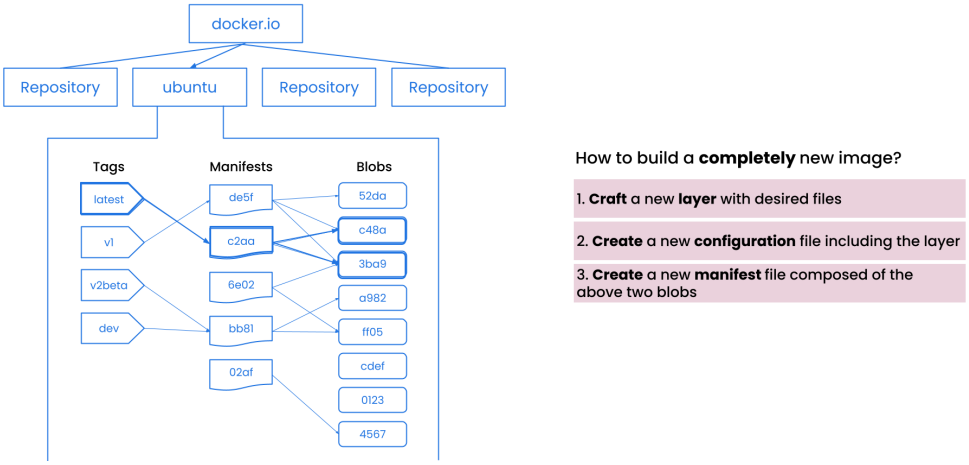
Σχήμα 3.3: *Η ανατομία μίας εικόνας περιεκτών: επίπεδο εικόνας*

σε σχέση με το **γονικό** του στρώμα. Τα στρώματα δεν έχουν μεταδεδομένα διαμόρφωσης, όπως μεταβλητές περιβάλλοντος ή προεπιλεγμένα ορίσματα - αυτά είναι ιδιότητες της εικόνας στο σύνολό της και όχι οποιουδήποτε συγκεκριμένου στρώματος.

Συνοψίζοντας, τα επίπεδα απεικονίζουν τις **αλλαγές** στο σύστημα αρχείων. Αυτός είναι ο λόγος για τον οποίο στο αρχείο ρυθμίσεων - κάτω από το πεδίο rootfs - το όνομα για την αναπαράσταση των κατακερματισμών περιεχομένου του στρώματος ονομάζεται diff_ids. Σημείωση: Τα diff_ids είναι το digest των **αποσυμπιεσμένων** tar αρχείων, πράγμα που σημαίνει ότι τα hashes αυτά αντιπροσωπεύουν στην πραγματικότητα τις διαφορές μεταξύ των στρωμάτων, ενώ στο αρχείο manifest η λίστα layers χρησιμεύει ως δείκτης στα συμπιεσμένα αρχεία που βρίσκονται στο σύστημά μας.

3.3 Αλγόριθμος Επέκτασης Εικόνων Περιεκτών

Σε αυτή την ενότητα εκμεταλλευόμαστε τις γνώσεις που αποκτήσαμε στα προηγούμενα κεφάλαια σχετικά με τα container images και παρουσιάζουμε έναν τρόπο επέκτασης μιας εικόνας OCI. Εξετάσαμε σε βάθος τι είναι μια εικόνα και τι κρύβεται πίσω από τα layers της εικόνας. Ανακαλύψαμε το σκοπό του manifest και του αρχείου ρυθμίσεων και τη σχέση μεταξύ τους. Συνειδητοποιήσαμε ότι τα έγγραφα JSON που συνθέτουν την τελική εικόνα θεωρούνται αμετάβλητα και η αλλαγή τους σημαίνει τη δημιουργία



Σχήμα 3.4: *Ένα μητρώο περιέχει πολλαπλά αποθετήρια, καθένα από τα οποία περιλαμβάνει διαφορετικές ετικέτες και παραστατικά που συνθέτουν πολλαπλές εικόνες.*

μιας νέας μοναδικής εικόνας.

Συνεπώς, ο χειρισμός αυτών των εγγράφων έχει ως αποτέλεσμα τη δημιουργία μιας εντελώς νέας εικόνας. Μάλιστα, ένα ελαφρώς διαφορετικό αρχείο manifest μπορεί να παραπέμπει σε διαφορετικά layers και διαφορετικό αρχείο διαμόρφωσης. Γνωρίζοντας, λοιπόν, τι ακριβώς είναι ένα στρώμα θα μπορούσαμε να ξεκινήσουμε με την κατασκευή ενός νέου - από μηδενική βάση - τέτοιου στρώματος, το οποίο υποθετικά περιλαμβάνει τις εξαρτήσεις μιας εφαρμογής. Στη συνέχεια, ξεκινώντας από μια εικόνα βάσης, θα επεξεργαστούμε κατάλληλα τα αρχεία manifest και configuration της, προκειμένου να ενσωματώσουμε το νέο μας layer στην εικόνα βάσης. Θα μπορούσαμε να πούμε ότι προσομοιώνουμε την προσθήκη μιας νέας εντολής σε ένα Dockerfile, αλλά χωρίς σε καμία περίπτωση να χρησιμοποιούμε το Docker και τον περιορισμό του να εκτελείται μόνο σε προνομιούχα περιβάλλοντα.

Αλγόριθμος

Πρώτα απ' όλα, πρέπει να δημιουργήσουμε το νέο μας layer που θα τοποθετηθεί πάνω σε μια βασική εικόνα. Είναι σημαντικό να θυμόμαστε πως ένα στρώμα που συμμορφώνεται με τις προδιαγραφές εικόνας OCI είναι ένα συμπιεσμένο αρχείο tar. Ωστόσο, πριν από τη συμπίεσή του θα πρέπει πρώτα να υπολογίσουμε το digest του αρχείου tar - που αντιπροσωπεύει τη διαφορά μεταξύ του ίδιου και του βασικού στρώματος - προκειμένου να προσαρτηθεί στο αρχείο ρυθμίσεων. Στη συνέχεια, τα μεταδεδομένα του συμπιεσμένου στρώματος θα ενσωματωθούν στο αρχείο manifest που παράγει τη νέα εικόνα. Μια περισσότερο τυποποιημένη περιγραφή του αλγόριθμου είναι η παρακάτω,

η οποία δείχνει τη λειτουργία του αλγόριθμου σε βήματα:

1. Κατεβάζουμε μια βασική εικόνα χρησιμοποιώντας το Docker, το Skopeo ή οποιοδήποτε άλλο εργαλείο που μεταφέρει εικόνες από ένα μητρώο πηγής σε ένα μητρώο προορισμού.

2. Αποφασίζουμε τα περιεχόμενα που θα συμπεριληφθούν στην εικόνα και δημιουργούμε το αντίστοιχο αρχείο tar

   (α′) Παράγουμε το αρχείο tar που αποτελείται από τα περιεχόμενα του συγκεκριμένου στρώματος

   (β′) Υπολογίζουμε τον κατακερματισμό του ασυμπίεστου περιεχομένου του στρώματος με χρήση του αλγορίθμου sha256.

3. Επεξεργασία του αρχείου ρυθμίσεων

   (α′) Βρίσκουμε το όνομα του αρχείου διαμόρφωσης στο αρχείο manifest

   (β′) Προσαρτούμε στο αρχείο διαμόρφωσης τα μεταδεδομενά του μη συμπιεσμένου στρώματος. Δηλαδή, προσθέστε το digest του μη συμπιεσμένου στρώματος στη λίστα των rootfs.diff_ids που βρίσκονται στο αρχείο που περιγράφει τις διαφορές μεταξύ των στρωμάτων.

   (γ′) Υπολογίζουμε τα μεταδεδομένα της ενημερωμένης διαμόρφωσης (μέγεθος & άθροισμα ελέγχου) και μετονομάζουμε το αρχείο διαμόρφωσης στο νέο άθροισμα ελέγχου.

4. Συμπιέζουμε το αρχείο tar του στρώματος και υπολογίζουμε το άθροισμα ελέγχου και το μέγεθος του συμπιεσμένου περιεχομένου του στρώματος.

5. Επεξεργασία του αρχείου manifest

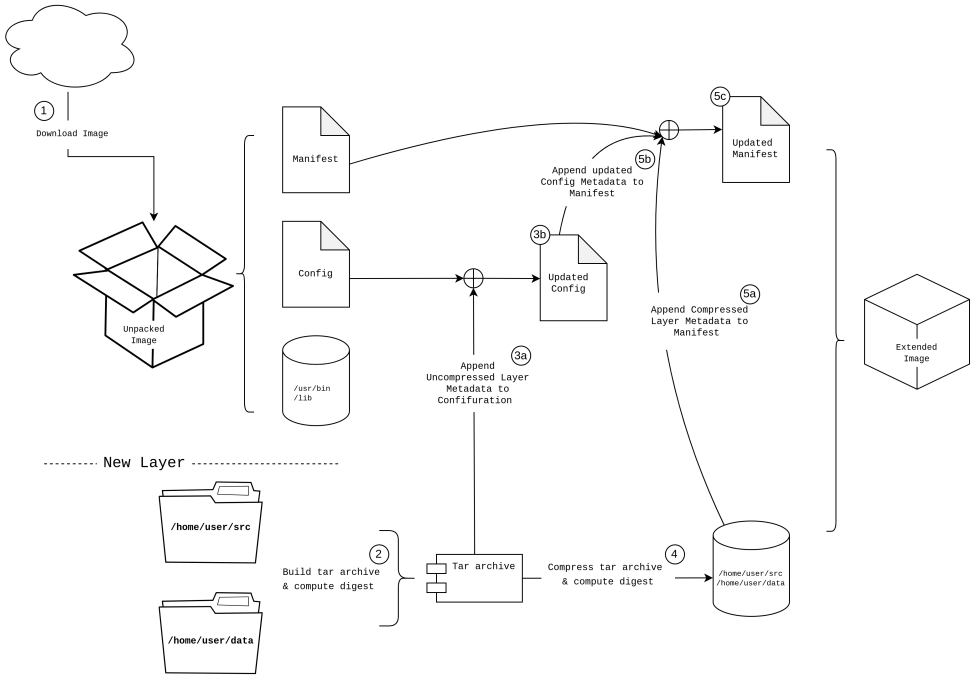
   (α′) Προσαρτούμε τα μεταδεδομένα του νέου στρώματος στη λίστα στρωμάτων του manifest αρχείου

   (β′) Τροποποιούμε το πεδίο config με τα νέα μεταδεδομένα που υπολογίστηκαν στο βήμα 3γ

   (γ′) Υπολογίζουμε τα μεταδεδομένα του νέου manifest (μέγεθος και άθροισμα ελέγχου) και μετονομάζουμε το αρχείο με το νέο άθροισμα ελέγχου.

6. Επεξεργασία του αρχείου ευρετηρίου

   (α′) Τροποποιούμε το πεδίο manifest ώστε να περιέχει τα μεταδεδομένα του ενημερωμένου manifest



Σχήμα 3.5: *Απεικόνιση του αλγορίθμου που περιγράφει τον τρόπο επέκτασης μιας βασικής εικόνας*

3.4 Μηχανισμός Κατασκευής Εικόνων Περιεκτών

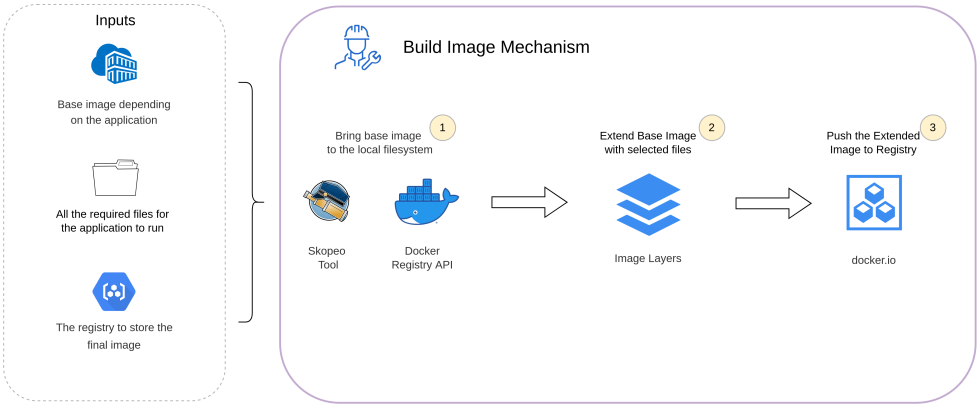
Στα προηγούμενα κεφάλαια εξερευνήσαμε τους περιέκτες του Linux. Εξετάσαμε τις εσωτερικές λειτουργίες ενός container όπως τα namespaces και τα cgroups. Συνειδητοποιήσαμε ότι ένας container δεν είναι τίποτα περισσότερο από μια διεργασία Linux με φανταχτερά κόλπα για απομόνωση και προστασία που αξιοποιούν διάφορα χαρακτηριστικά του πυρήνα του Linux. Στη συνέχεια, αναλύσαμε ποιος αποφασίζει για τα περιεχόμενα ενός container και πώς καθορίζεται το τελικό σύστημα αρχείων. Ένα container image που αποτελείται από έναν αριθμό επιπέδων όταν αποσυμπιέζεται συναρμολογεί ένα πακέτο χρόνου εκτέλεσης το οποίο στη συνέχεια χρησιμοποιείται για την εκκίνηση του container. Προς το τέλος του προηγούμενου κεφαλαίου συνειδητοποιήσαμε το μεγαλείο και τη μεγάλη αξία των στρωμάτων εικόνας. Η δυνατότητα διαμοιρασμού τους μεταξύ εικόνων ή η επέκταση και συρρίκνωση μιας εικόνας με την προσθήκη/αφαίρεση στρωμάτων αποδεικνύει την ευκολία χρήσης, την επεκτασιμότητα και την αποτελεσματικότητα του κόσμου των containers.

Σίγουρα η διαδικασία χειροκίνητου χειρισμού μιας εικόνας container με στόχο την επέκτασή της είναι μάλλον επαχθής. Αυτός είναι ο λόγος για τον οποίο υπάρχουν προγράμματα εκτέλεσης container όπως το Docker, σωστά; Και ναι και όχι! Το Docker, το Podman ή οποιοδήποτε άλλο πρόγραμμα εκτέλεσης container απαιτεί κατανόηση των εννοιών των container, των εικόνων και των μητρώων. Η συγγραφή ενός Dockerfile απαιτεί και την αντίστοιχη εξοικείωση με το Docker. Γιατί ένας επιστήμονας δεδομένων να μπει στον κόπο να φτιάξει εικόνες με όλες τις βιβλιοθήκες, τον κώδικα, τα δεδομένα και τις πολύπλοκες εξαρτήσεις του μοντέλου που αναπτύσσει; Τι συμβαίνει σε περίπτωση λάθους ή σφάλματος στον κώδικα; Πρέπει να διορθώσουμε το σφάλμα και να ξαναχτίσουμε την εικόνα με τις κατάλληλες αλλαγές και στη συνέχεια άλλο ένα σφάλμα και πάλι ξανα χτίσιμο και ο κύκλος δεν τελειώνει ποτέ.

Εκτός από αυτό, ένα σημαντικό μειονέκτημα του Docker είναι η σκληρή απαίτηση δικαιωμάτων root για την εκτέλεση ενός container και τη δημιουργία εικόνων. Στην πραγματικότητα, το Docker αναγκάζει τους χρήστες να επικοινωνούν με τον αντίστοιχο δαίμονα. Επομένως, σε περίπτωση που κάποιος θέλει να τρέξει έναν container μέσα σε έναν container, το Docker δεν αποτελεί άμεση επιλογή.

Σε αυτό το κεφάλαιο παρουσιάζουμε ένα Python SDK που επιτρέπει στους χρήστες

να επεκτείνουν μια βασική εικόνα με οποιοδήποτε είδος αρχείων ή φακέλων. Χρησιμοποιώντας αυτή τη δυνατότητα, οι χρήστες θα μπορούν να δημιουργούν απρόσκοπτα εικόνες Docker σε μη προνομιούχα περιβάλλοντα. Αυτή η βιβλιοθήκη αποτελεί το πρώτο βήμα για την παροχή πλήρους διαφάνειας και αυτοματοποίησης στη διαδικασία ανάπτυξης ροών εργασίας μηχανικής μάθησης. Ο μηχανισμός δεν έχει σχεδιαστεί



Σχήμα 3.6: *Οπτική απεικόνιση σε υψηλό επίπεδο του μηχανισμού κατασκεύης εικόνων περιεκτών*

απαραίτητα για να χρησιμοποιείται άμεσα από τον τελικό χρήστη. Στα επόμενα κεφάλαια θα δούμε πώς θα ενσωματωθεί στο γνωστό εργαλείο Kale (δείτε περισσότερα στην ενότητα 2.9) προκειμένου να δημιουργηθούν με το πάτημα ενός κουμπιού ροές εργασίας ML που θα αναπτυχθούν στον Κυβερνήτη. Παρ' όλα αυτά, η προσέγγιση αυτή επιτρέπει σε κάποιον να κατασκευάσει και να επεκτείνει εικόνες της επιλογής του, παρέχοντας τις ελάχιστες απαιτούμενες πληροφορίες που σχετίζονται με εικόνες και containers. Η ευθύνη του μηχανισμού είναι να επεκτείνει μια βασική εικόνα με ένα νέο στρώμα, που αποτελείται από μια δέσμη αρχείων/φακέλων. Τα βήματα που ακολουθεί είναι τα εξής:

1. Ανακτά την βασική εικόνα από ένα μητρώο πηγής και την αποθηκεύει στο τοπικό σύστημα αρχείων

2. Κατασκευάζει ένα tarball που περιλαμβάνει αρχεία και φακέλους όπως για παράδειγμα θα μπορούσε να είναι ο πηγαίος κώδικας, οι εγκατεστημένες βιβλιοθήκες, τα δεδομένα και οι σύνθετες εξαρτήσεις μιας εφαρμογής

3. Μετατρέπει το tarball σε ένα στρώμα εικόνας και το επισυνάπτει πάνω από τη βασική εικόνα.

4. Τέλος, μεταφέρει τη νέα εικόνα σε ένα μητρώο στο οποίο μπορεί να έχει πρόσβαση ένα περιβάλλον εκτέλεσης container.

Για τα βήματα (1) και (4) ο μηχανισμός αξιοποιεί τις δυνατότητες του Skopeo γύρω από τη μεταφορά εικόνων από μια πηγή σε έναν προορισμό.

Ο επιστήμονας δεδομένων ως το άτομο που αναπτύσσει, δοκιμάζει και τελικά κάνει deploy την εφαρμογή του θα κάνει τα εξής:

1. Αναπτύσσει την εφαρμογή του που αποτελείται από εγκατεστημένες βιβλιοθήκες, σύνθετες εξαρτήσεις όπως δεδομένα και κώδικα

2. Επιλέγει όλα τα απαιτούμενα αρχεία για την εκτέλεση της εφαρμογής του

3. Καλεί την συνάρτηση *build_image* παρέχοντας μόνο ένα όνομα εικόνας βάσης και τα αρχεία της επιθυμίας τους για τη μεταφορά της εφαρμογής τους σε container

4. Βρίσκει ένα σφάλμα στον κώδικά του, επιστρέφει στο (3)

Ας υποθέσουμε ότι αναπτύσσουμε μια υπέροχη εφαρμογή στην γλώσσα προγραμματισμού Python. Η βασική μας εικόνα θα είναι πιθανότατα μια εικόνα Python, δηλαδή μια εικόνα που περιέχει όλες τις απαραίτητες βιβλιοθήκες για να χρησιμοποιήσουμε την γλώσσα αυτή. Στη συνέχεια, η εφαρμογή μας αποτελείται από:

- Τον πηγαίο μας κώδικα που βρίσκεται στον κατάλογο $CWD/src
- Τα δεδομένα μας που βρίσκονται στον κατάλογο $CWD/data και
- Τις εγκατεστημένες βιβλιοθήκες μας python που βρίσκονται κάτω από τον κατάλογο $HOME/.local

Στόχος μας είναι να δημιουργήσουμε μια νέα εικόνα που έχει ως βάση της την Python που θα περιλαμβάνει όλα τα παραπάνω απαραίτητα αρχεία για να τρέξει η εφαρμογή μας. Για να το πετύχουμε αυτό καλούμε τη συνάρτηση *build_image* από τη βιβλιοθήκη μας:

```
>>> build_image(base_image="docker.io/python:latest",
. . .           include_paths=["/home/user/src", "/home/user/data",
. . .                          "/home/user/.local"],
. . .           dst_image="docker.io/pangiann/fabulous-app:v1")
'docker.io/pangiann/fabulous-app:v1@<digest>'
```

Αυτό είναι το μόνο που χρειάζεται να κάνουμε για να δημιουργήσουμε μια νέα έτοιμη προς χρήση εικόνα για την εφαρμογή μας και να την προωθήσουμε σε ένα μητρώο για να την καταστήσουμε δημόσια προσβάσιμη. Τι γίνεται αν ξεχάσαμε να προσθέσουμε ένα κρίσιμο αρχείο; Αυτό δεν αποτελεί πρόβλημα για τον μηχανισμό δημιουργίας εικόνας:

```
>>> build_image(base_image="docker.io/pangiann/fabulous:v1",
. . .           include_paths=["/home/user/icons"],
. . .           dst_image="docker.io/pangiann/fabulous-app:v2")
'docker.io/pangiann/fabulous-app:v2@<digest>'
```

Σημειώστε ότι το Skopeo ψάχνει σε συγκεκριμένα σημεία για να ανακτήσει τυχόν αρχεία εξουσιοδότησης και πιστοποίησης για να μας πιστοποιήσει στο επιλεγμένο μητρώο. Για το αρχείο πιστοποίησης η βασική διαδρομή είναι *$XDG_RUNTIME_DIR/containers/auth.json*, η οποία ορίζεται με τη χρήση του skopeo login. Εάν η κατάσταση εξουσιοδότησης δεν βρεθεί εκεί, ελέγχεται το *$HOME/.docker/config.json*, το οποίο έχει οριστεί χρησιμοποιώντας το docker login. Όπως μπορείτε να φανταστείτε, αυτή η προσέγγιση δεν καλύπτει τις ανάγκες όλων. Για το λόγο αυτό ο μηχανισμός προσφέρει την επιλογή να παρέχονται τα αρχεία ελέγχου ταυτότητας και πιστοποίησης τόσο για το μητρώο προέλευσης όσο και για το μητρώο προορισμού.

```
>>> build_image(base_image="docker.io/pangiann/fabulous:v1",
...             include_paths=["/home/user/icons"],
...             dst_image="docker.io/pangiann/fabulous-app:v2",
...             dst_auth_file="/user/kale/auth.json")
'docker.io/pangiann/fabulous-app:v2@<digest>'
```

Σε περίπτωση που έχουμε φέρει μια βασική εικόνα στο τοπικό μας σύστημα αρχείων και θέλουμε να την επεκτείνουμε όπως κάναμε στην ενότητα 3.3, η βιβλιοθήκη προσφέρει και αυτή την επιλογή με τη συνάρτηση *extend_image* η οποία επιστρέφει ένα αντικείμενο *DockerImage* το οποίο αντικατοπτρίζει μια εικόνα Docker σε αποσυμπιεσμένη μορφή περιλαμβάνοντας ταυτόχρονα διάφορες λειτουργίες σχετικές με container images.

```
>>> new_image = extend_image(base_image_path="/home/user/python_image",
...                          include_paths=["/home/user/src",
...                                         "/home/user/.local"])
>>> print(new_image.image_digest)
'sha256:<digest>'
```

3.5 Kubeflow Δομικά Στοιχεία

Το Kubeflow είναι ένα δωρεάν έργο ανοικτού κώδικα που έχει σχεδιαστεί για να κάνει την εκτέλεση ροών εργασίας μηχανικής μάθησης σε συστάδες του Κυβερνήτη απλούστερη και πιο συντονισμένη. Πρόκειται για ένα cloud-native framework για τη χρησιμοποίηση μηχανικής μάθησης σε containerized περιβάλλοντα στον Κυβερνήτη. Το Kubeflow έχει σχεδιαστεί για να εκτελείται όπου εκτελείται ο Κυβερνήτης. Μάλιστα, η ενσωμάτωση του πρώτου στο δεύτερο είναι πια μια απρόσκοπτη διαδικασία. [12]. Στον πυρήνα του, το Kubeflow προσφέρει μια εργαλειοθήκη ενορχήστρωσης από άκρη σε άκρη της στοίβας ML για να βασιστεί στο K8s ως ένας τρόπος ανάπτυξης, κλιμάκωσης και διαχείρισης πολύπλοκων συστημάτων. Αντιλαμβανόμαστε, λοιπόν, ότι παρέχει μια πληθώρα εργαλείων απευθείας στο περιβάλλον παραγωγής, δηλαδή, στον Κυβερνήτη, ενώνοντας έτσι το στάδιο ανάπτηξυς με το στάδιο παραγωγής. Οι επιστήμονες δεδομένων και οι μηχανικοί είναι πλέον σε θέση να αναπτύξουν ένα πλήρες pipeline το οποίο θα αποτελείται από συγκεκριμένα συστατικά, δηλαδή, βήματα του workflow χαλαρά συνδεδεμένα μεταξύ τους.

3.5.1 Τι είναι ένα Δομικό Στοιχείο Kubeflow;

Ένα Kubeflow pipeline είναι επεκτάσιμος ορισμός μιας ροής εργασίας μηχανικής μάθησης (ML), βασισμένος σε container. Αποτελείται από ένα σύνολο παραμέτρων εισόδου και έναν κατάλογο των βημάτων αυτής της ροής εργασίας. Κάθε βήμα στο pipeline είναι μια περίπτωση ενός δομικού στοιχείου. Όταν σχεδιάζουμε ένα pipeline, θα πρέπει να εξετάσουμε πώς θα χωρίσουμε τη ροή εργασίας ML στα δομικά αυτά στοιχεία. Κάθε τέτοιο δομικό στοιχείο θα πρέπει να έχει μία μόνο ευθύνη. Η ύπαρξη μίας μόνο ευθύνης διευκολύνει τη δοκιμή και την επαναχρησιμοποίησή του. Για παράδειγμα, ένα δομικό στοιχείο που φορτώνει δεδομένα μπορεί να επαναχρησιμοποιηθεί για παρόμοιες εργα-

σίες που φορτώνουν δεδομένα [20].

Τελικά, ένα **Kubeflow Pipelines component** είναι μια containerized εφαρμογή - αυτοτελές σύνολο κώδικα - που εκτελεί ένα βήμα σε μια ροή εργασίας ML [21]. Αποτελείται από:

- Τον κώδικα του δομικού στοιχείου, ο οποίος υλοποιεί τη λογική που απαιτείται για την εκτέλεση ενός βήματος στη ροή εργασίας MM
- Μια προδιαγραφή του δομικού στοιχείου, η οποία ορίζει τα εξής:
    - *Τα μεταδεδομένα του δομικού στοιχείου*: το όνομα και η περιγραφή του
    - *Η διεπαφή του δομικού στοιχείου*: τα **inputs** και **outputs**
    - *Η υλοποίηση του δομικού στοιχείου*: η εικόνα **Docker container image** που πρέπει να εκτελεστεί, ο τρόπος με τον οποίο θα περάσουμε τις εισόδους στον κώδικα του στοιχείου μας και ο τρόπος με τον οποίο θα λάβουμε τις εξόδους αυτού.

Κάθε ένα από αυτά τα δομικά στοιχεία μπορούν να ορίσουν τις εισόδους τους ως εξαρτώμενες από την έξοδο ενός άλλου βήματος. Οι εξαρτήσεις μεταξύ των βημάτων ορίζουν το γράφημα ροής εργασιών του pipeline. Η κατασκευή ενός pipeline, σημαίνει την ξεχωριστή κατασκευή κάθε δομικού στοιχείου που θα κατασκευάσει το τελικό pipeline.

3.5.2 Κατασκευή Δομικών Στοιχείων με το Kubeflow

Ένα pipeline αποτελεί μια περιγραφή μιας ροής εργασίας ML, η οποία περιλαμβάνει όλα τα δομικά στοιχεία της ροής εργασίας και τον τρόπο με τον οποίο αυτά συσχετίζονται, δημιουργώντας τελικά έναν γράφο. Το pipeline περιέχει τον ορισμό των εισόδων - παραμέτρων του pipeline - που απαιτούνται για την εκτέλεσή του και των εισόδων και εξόδων καθενός από τα δομικά στοιχεία από τα οποία αποτελείται.

Ένα δομικό στοιχείο είναι τελικά ενός κόμβος του γράφου που προκύπτει από το pipeline που έχουμε ορίσει. Μπορεί να θεωρήθει ανάλογο μιας συνάρτησης, δεδομένου ότι έχει ένα όνομα, παραμέτρους, τιμές επιστροφής και ένα σώμα. Όταν εκτελούμε ένα pipeline, ένα ή περισσότερα Kubernetes Pods για κάθε δομικό στοιχείο (για κάθε βήμα δηλαδή

του pipeline) εκκινούνται. Τα ίδια τα Pods εκκινούν τα απαραίτητα Docker containers, τα οποία με τη σειρά τους εκκινούν το πρόγραμμά μας. Για να συμβεί αυτό, το KFP χρησιμοποιεί τον *Argo Workflow Executor* κάτω από αυτό. Εν συντομία, καταφέρνει να μεταφράσει ένα Kubeflow Pipeline σε ένα Argo Workflow το οποίο πρακτικά είναι ένα resource του Κυβερνήτη όπως τα Pods, Deployments κ.α.

Το KFP παρέχει έναν τρόπο μετατροπής μιας συνάρτησης Python σε ένα επαναχρησιμοποιούμενο και διαμοιραζόμενο δομικό στοιχείο. Υπάρχουν δύο επιλογές που προσφέρει το KFP για τη δημιουργία ενός δομικού στοιχείου από μια συνάρτηση βασισμένη στην Python. Το παρεχόμενο API περιλαμβάνει την εντολή *create_component_from_func()*. Από προεπιλογή, το KFP χρησιμοποιεί μια βασική εικόνα Python. Ταυτόχρονα, μεταφέρει τον αντίστοιχο κώδικα στο Argo workflow το οποίο θα χρησιμοποιηθεί για την εκκίνηση του αντίστοιχου Pod στον Κυβερνήτη. Όμως, είπαμε πως κάθε βήμα ενός pipeline έχει εισόδους και εξόδους όπως και μια συνάρτηση. Οι containers κάνουν χρήση ενός σημείου εισόδου για να ξεκινήσουν την εκτέλεσή τους. Ως εκ τούτου, το KFP παράγει αυτόματα τον argument parser για να περάσει:

(Α′) τις εισόδους μέσω του σημείου εισόδου και

(Β′) τη διαδρομή προς το αρχείο όπου θα αποθηκευτούν οι έξοδοι προκειμένου να έχουν πρόσβαση σε αυτές τα επόμενα βήματα.

Αυτή η προσέγγιση πάσχει από πολλές επιπτώσεις και περιορισμούς. Η συνάρτηση της Python πρέπει να είναι αυτόνομη, δηλαδή:

1. Δεν πρέπει να χρησιμοποιεί κώδικα που δηλώνεται εκτός του ορισμού της συνάρτησης
2. Οι δηλώσεις εισαγωγής πρέπει να προστίθενται μέσα στη συνάρτηση.
3. Οι βοηθητικές συναρτήσεις πρέπει να ορίζονται μέσα στη συνάρτηση

Όλα τα παραπάνω αναγκάζουν τους χρήστες του Kubeflow να ακολουθήσουν τη δεύτερη επιλογή, η οποία είναι να κατασκευάσουν οι ίδιοι την εικόνα Docker του component, συμπεριλαμβανομένων όλων των πολύπλοκων εξαρτήσεων που απαιτούνται για την εκτέλεσή του. Αμέσως, όμως, αυτή η προσέγγιση παραβιάζει τον ”κανόνα” ότι οι Data Scientists δεν πρέπει να ασχολούνται με λειτουργίες σχετικές με Container/Kubernetes.

Αυτή η διαδικασία καθιστά επίσης πολύ πιο δύσκολη την αποσφαλμάτωση, την τροποποίηση του κώδικά μας και τη γρήγορη ανάπτυξη του μοντέλου μας, καθώς πρέπει να κατασκευάσουμε επαναληπτικά Docker Images με τα ενημερωμένα περιεχόμενα. Ένα απλό παράδειγμα του KFP dsl παρουσιάζεται παρακάτω:

```
# This is a simple function that will be executed in a KF Pipeline step.
# Notice that the inputs and outputs are simple Python values, so we
# annotate them with their type (float).
def add(a: float, b: float) -> float:
    """Calculate sum of two arguments."""
    return a + b

# This is the reproducible component that is created out of the
# above function. We'll use for both of the KF Pipeline steps.
add_op = create_component_from_func(add)


@dsl.pipeline(
    name='Calculation pipeline',
    description='An example pipeline that performs arithmetic calculations.'
)
def calc_pipeline(a='1'):
    """The Pipeline function."""
    first_task = add_op(a, 4)

    # Since the 'add()' function returns just one single value as output,
    # we can access the output of the 1st task through `first_task.output`.
    second_task = add_op(first_task.output, 4)

Compiler().compile(calc_pipeline, 'workflow.yaml')
```

Όπως μπορούμε να παρατηρήσουμε στο παραπάνω παράδειγμα γίνεται χρήση της παρεχόμενης από το KFP συνάρτησης create_component_from_func() η οποία όπως είπαμε νωρίτερα έχει πολλούς περιορισμούς καθιστώντας την ουσιαστικά άχρηστη για τους περισσότερους χρήστες. Στην περίπτωση που θέλουμε να ακολουθήσουμε τη δεύτερη επιλογή πρέπει να κάνουμε τα εξής:

1. Επέκταση του κώδικά μας με τον κατάλληλο argument parser για να επιτρέψουμε

να δεχθεί τις αντίστοιχες εισόδους μέσω του entrypoint, δηλαδή της εναρκτήριας εντολής ενός container.

2. Δημιουργία ενός Docker Image που θα περιέχει τον κώδικα και όλες τις εξαρτήσεις του

3. Δημιουργία του component definition, δηλαδή ενός αρχείου YAML που ορίζει τις απραίτητες πληροφορίες για το component

Παρακάτω παραθέτουμε έναν κοινό αναλυτή ορισμάτων που χρησιμοποιείται συχνά για το πέρασμα εισόδων και εξόδων σε κομμάτια Python κώδικα που καλούνται να τρέξουν σε έναν container. Όμοιοι αναλυτές υπάρχουν σχεδόν σε όλες τις περιπτώσεις εφαρμογών που παρέχουν πέρα από ένα API και το αντίστοιχο CLI (command line interface) για την επικοινωνία με αυτό.

```
# Defining and parsing the command-line arguments
parser = argparse.ArgumentParser(description='My program description')
# Paths must be passed in, not hardcoded
parser.add_argument('--input1', type=float,
  help='Input data')
parser.add_argument('--input2', type=float,
  help='Input data')
parser.add_argument('--output-path', type=str,
  help='Path of the local file where the Output data should be written.')
args = parser.parse_args()
```

Τέλος, παρακάτω βλέπουμε ένα απλό παράδειγμα ενός component definition το οποίο μπορεί να φορτωθεί στον αρχικό μας κώδικα και να χρησιμοποιηθεί ως βήμα σε ένα pipeline κάνοντας χρήση της συνάρτησης *load_component_from_file()*.

```
inputs:
- {name: Input 1, type: String, description: 'Data for input 1'}
- {name: Parameter 1, type: Integer, default: '100', description: 'Number of lines to copy'}

outputs:
- {name: Output 1, type: String, description: 'Output 1 data.'}

implementation:
  container:
```

```
    image: gcr.io/my-org/my-image@sha256:a172..752f
    # command is a list of strings (command-line arguments).
    # The YAML language has two syntaxes for lists and you can use either of them.
    # Here we use the "flow syntax" - comma-separated strings inside square brackets.
    command: [
      python3,
      # Path of the program inside the container
      /pipelines/component/src/program.py,
      --input1-path,
      {inputPath: Input 1},
      --param1,
      {inputValue: Parameter 1},
      --output1-path,
      {outputPath: Output 1},
    ]
```

3.5.3 Argo Workflow Executor

Ο παραπάνω ορισμός του δομικού στοιχείου μετατρέπεται σε μια ροή εργασίας Argo. Μια ροή εργασίας Argo είναι ουσιαστικά ένα CustomResource (CR) του Κυβερνήτη που περιγράφει ολόκληρο το γράφημα DAG και την εκτέλεση ενός KubeFlow Pipeline. Ορίζει τη ροή εργασίας όπου κάθε βήμα σε αυτήν είναι ένας container. Περιέχει τον κώδικα που θα εκτελεί κάθε βήμα του Pipeline και καθορίζει επίσης τις εξαρτήσεις και τη διακίνηση δεδομένων μεταξύ των βημάτων. Το KFP SDK είναι σε θέση να παράγει την αντίστοιχη ροή εργασίας Argo ξεκινώντας από ένα Pipeline. Το pipeline ουσιαστικά μεταγλωττίζεται σε ένα Argo Workflow το οποίο στη συνέχεια εφαρμόζεται σε μια συστάδα του Κυβερντήτη και το pipeline αρχίζει να εκτελείται.

```
- name: add
container:
  args: [--input1, '{{inputs.parameters.a}}', --input2, '4', '--output-path',
↪  /tmp/outputs/Output/data]
  command:
  - python3
  - script.py
  inputs:
    parameters:
    - {name: a}
```

```
    outputs:
      parameters:
      - name: add-Output
      valueFrom: {path: /tmp/outputs/Output/data}
...
```

Μπορούμε πλέον να αντιληφθούμε γιατί είναι υποχρεωτική η επέκταση του αρχικού μας κώδικα με έναν αναλυτή ορισμάτων. Δεν αρκεί φυσικά να τρέξει ο κώδικας μας με χρήση της κλασσικής εντολής 'python3 script.py', χρειάζονται και τα αντίστοιχα ορίσματα ώστε να περαστούν κατάλληλα οι είσοδοι που θα αποτελέσουν και παραμέτρους της σχετικής συνάρτησης του βήματος που εκτελείται.

Η εμπειρία του χρήστη για την παραγωγή Kubeflow Pipelines είναι μάλλον δύστροπη και στριφνή. Παραδόξως, η κατάσταση επιδεινώνεται. Τι πρέπει να κάνουμε σε περίπτωση σφάλματος; Να ενημερώσουμε κατάλληλα τον κώδικά μας, να αλλάξουμε τον αναλυτή επιχειρημάτων, να δημιουργήσουμε μια νέα εικόνα Docker και να αλλάξουμε την υλοποίηση του component μας. Στη μηχανική μάθηση, που είναι η περίπτωση που μελετάμε, αυτό μπορεί να είναι αρκετό όταν τα μοντέλα σπάνια αλλάζουν ή εκπαιδεύονται. Στην πράξη, τα μοντέλα συχνά σπάνε όταν αναπτύσσονται στον πραγματικό κόσμο. Αποτυγχάνουν να προσαρμοστούν στις αλλαγές στη δυναμική του περιβάλλοντος ή στις αλλαγές στα δεδομένα που περιγράφουν το περιβάλλον, επομένως πρέπει να αλλάζουμε συνεχώς τον κώδικα και τα δεδομένα μας.

3.6 Εκτέλεση Kubeflow Pipelines με Kale & Rok

Είναι περιττό να πούμε ότι χρειαζόμαστε έναν τρόπο να δημιουργούμε εύκολα έτοιμα για την παραγωγή pipeline μηχανικής μάθησης με λειτουργίες "point and click" και να επιστρέφουμε άμεσα σε οποιοδήποτε βήμα του pipeline για γρήγορη αποσφαλμάτωση. Το Kale με το Rok έκανε το όνειρο πραγματικότητα με τη λήψη στιγμιότυπων του περιβάλλοντος εργασίας του χρήστη και την επισύναψή του σε Pods όπου θα εκτελούνται τα βήματα. Δυστυχώς, προκειμένου να δημιουργηθούν κατάλληλα στοιχεία του KFP, η στήριξη στα στιγμιότυπα του Rok δεν αποτελεί άμεση επιλογή, καθώς όλος ο απαιτούμενος κώδικας πρέπει να συσκευαστεί μεταξύ της βασικής εικόνας docker και του σημείου εισόδου του container.

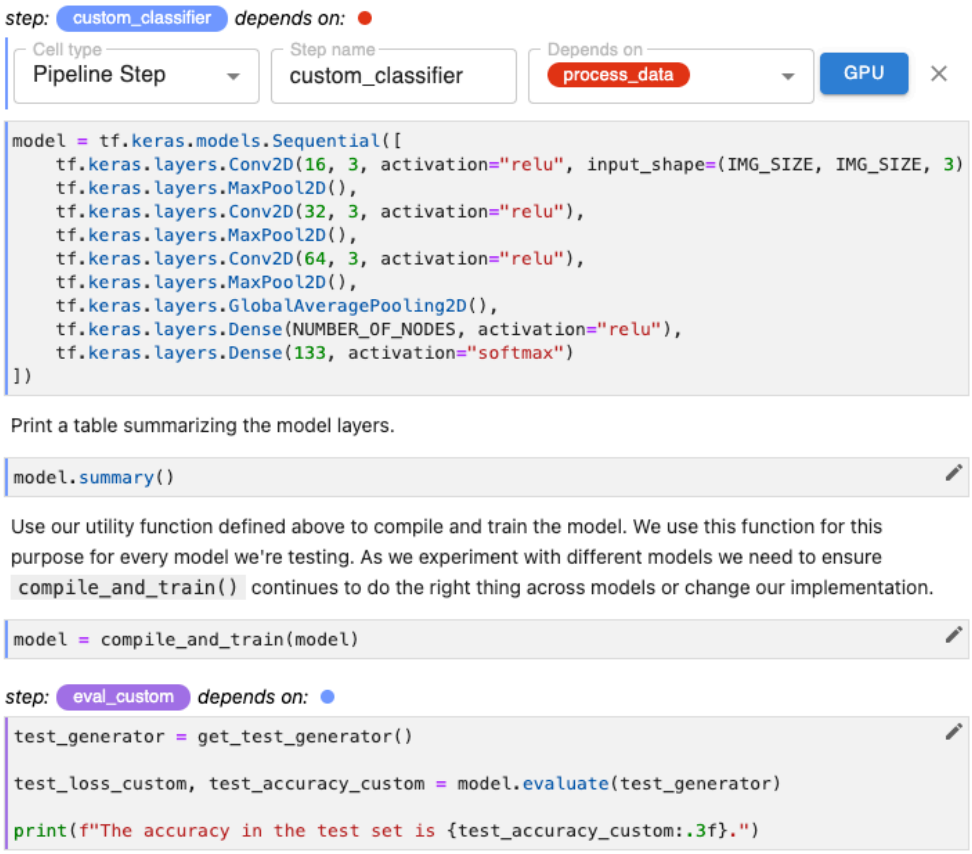
3.6.1 Kale Jupyter & SDK

Η βασική ιδέα πίσω από το Kale είναι να αξιοποιήσει τη δομή JSON των Jupyter Notebooks και να τα επεκτείνει κατάλληλα με μεταδεδομένα τόσο σε επίπεδο συνολικά του notebook αλλά και κάθε κελιού ξεχωριστά. [14]. Αυτές οι επισημειώσεις μας επιτρέπουν να:

1. Να αντιστοιχίσουμε κελιά κώδικα σε συγκεκριμένα components του pipeline
2. Συγχώνευση πολλαπλών κελιών σε ένα ενιαίο component του pipeline
3. Να καθορίσουμε τις εξαρτήσεις εκτέλεσης μεταξύ τους

Συνολικά, το Kale καθιστά αρκετά εύκολη την εννοιολόγηση των κελιών του Jupyter Notebook ως βήματα μιας ροής εργασίας ML. Σημειώστε ότι τα βήματα/Jupyter cells χωρίς εξαρτήσεις θα εκτελούνται παράλληλα σε Kubernetes Pods, μεγιστοποιώντας την απόδοση του Pipeline.

Για παράδειγμα, η παρακάτω εικόνα δείχνει πολλαπλά κελιά που αποτελούν μέρος του ίδιου βήματος ενός pipeline. Έχουν το ίδιο μπλε χρώμα και εξαρτώνται από ένα προηγούμενο βήμα του pipeline. Το μόνο που χρειάζεται να κάνει κανείς για να ορίσει βήματα είναι να επεξεργαστεί το αντίστοιχο κελί, να ονομάσει το βήμα αυτό όπως επιθυμεί και να ορίσει τις εξαρτήσεις του. Μια εξάρτηση μπορεί να είναι οποιοδήποτε από τα βήματα του pipeline. Το Kale αναλαμβάνει και μετασχηματίζει το notebook μας, μετατρέποντάς το σε ένα KFP pipeline. Επίσης, επειδή το Kale ενσωματώνεται με το Rok, οπότε λαμβάνει ένα στιγμιότυπο των volumes του χρήστη. Το Rok φροντίζει για την δημιουργία πολλαπλών εκδόσεων των δεδομένων και έτσι μας επιτρέπει να αναπαράγουμε ολόκληρο το περιβάλλον όπως ήταν όταν πατήσαμε το κουμπί COMPILE AND RUN. Με αυτόν τον τρόπο, έχουμε μια μηχανή του χρόνου για τα δεδομένα και τον κώδικά μας, ένα ακριβές versioned και αναπαραγώγιμο σημείο από το οποίο ξεκινάει το pipeline μας και το οποίο εκτελείται σε ένα πανομοιότυπο περιβάλλον με αυτό στο οποίο αναπτύξαμε τον κώδικά μας, χωρίς να χρειάζεται να δημιουργήσουμε νέες εικόνες docker. Λάβετε υπόψη ότι αυτή η προσέγγιση έχει και κάποιες επιπτώσεις και θα δούμε σε αυτή τη διπλωματική εργασία τι προσφέρει η διαφορετική μας προσέγγιση στον τελικό χρήστη και πώς μπορούμε να συνδυάσουμε και τους δύο μηχανισμούς (τον δικό



Σχήμα 3.7: *Η επέκταση JupyterLab του Kale μας επιτρέπει να σημειώσουμε τα κελιά ως βήματα του pipeline και να ορίσουμε τις εξαρτήσεις τους.*

μας και του Rok) για να τους αξιοποιήσουμε στο μέγιστο και να έχουμε τα καλύτερα αποτελέσματα.

Μπορούμε αναμφισβήτητα και με έμφαση να παραδεχτούμε την αξία του Kale στα Kubeflow Pipelines. Στην προηγούμενη ενότητα αντιληφθήκαμε ότι η χρήση απευθείας των Kubeflow Pipelines τις περισσότερες φορές απαιτεί μαζί με τη βαθιά κατανόηση του SDK του KFP, πολλές αλλαγές στον αρχικό πηγαίο κώδικα και γνώσεις σχετικά με containers, τις εικόνες και τον Κυβερνήτη. Εκτός από αυτό, οι επιστήμονες δεδομένων δεν είναι σε θέση να μετατρέψουν ένα Jupyter Notebook σε ένα Kubeflow Pipeline μόνο με το SDK του KFP. Το Kale έρχεται με μεγάλες και σημαντικές λύσεις, κάνοντας τη ζωή του μηχανικού Μηχανικής Μάθησης αρκετά πιο εύκολη, απαιτώντας μόνο να περιγράψει τη ροή εργασίας στο Kale και να πατήσει ένα κουμπί.

Εκτός από την επέκταση του JupyterLab το Kale προσφέρει ένα SDK που παρέχει τον απλούστερο τρόπο μετατροπής οποιουδήποτε κώδικα Python σε πλήρως αναπαραγώγιμες εκτελέσεις Kubeflow Pipelines χωρίς να αλλάξει ο πηγαίος κώδικας [22].

3.6.2 Pipeline & Steps

Ομοίως με την επέκταση JupyterLab, οι μηχανικοί ΜΜ πρέπει να περιγράψουν στο Kale ποια θα είναι τα βήματα (κόμβοι) του Pipeline και οι εξαρτήσεις τους (ακμές), ή με άλλα λόγια το τελικό γράφημα. Από το JupyterLab είδαμε ότι το Kale μας παρέχει σχόλια κελιών. Από το SDK, το Kale προσφέρει δύο διακοσμητές συναρτήσεων: *step* και *pipeline*.

Step Class

Οι συναρτήσεις που θέλουμε να είναι βήματα της ροής εργασίας ML πρέπει να τις διακοσμήσουμε με τον διακοσμητή step.

```
@step(name="data_loading")
def load(random_state):
    """Create a random dataset for binary classification."""
    rs = int(random_state)
    x, y = make_classification(random_state=rs)
    return x, y
```

Στο παρασκήνιο, το Kale ενσαρκώνει ένα αντικείμενο Step, συμπεριλαμβανομένων των ζωτικών πληροφοριών που απαιτούνται για τη μετατροπή του σε ένα δομικό στοιχείο KFP που θα αποτελέσει μέρος του τελικού Kubeflow Pipeline. Η κλάση *Step* περιλαμβάνει:

1. Η λειτουργία του βήματος που θα εκτελεστεί.
2. Τα μεταδεδομένα του βήματος, το όνομα και η περιγραφή του.
3. Η διεπαφή του βήματος, δηλαδή οι είσοδοι και οι έξοδοι του βήματος
4. Η υλοποίηση του στοιχείου, η εικόνα Docker που θα εκτελεστεί, το σημείο εισόδου και τα ορίσματα του CLI.
5. Η λογική marshalling που περιβάλλει τη συνάρτηση Python και υλοποιεί το μηχανισμό μεταφοράς δεδομένων του Kale μεταξύ των βημάτων, δηλαδή τροφοδοτεί τις εξόδους ενός βήματος στις εισόδους ενός επόμενου βήματος.

Pipeline Class

Στην επέκταση JupyterLab η επιλογή σχολιασμού depends on cell δείχνει τη σειρά με

την οποία θα εκτελεστούν τα βήματα. Στο Kale SDK, μια συνάρτηση του pipeline διακοσμημένη με το διακοσμητή *pipeline* περιέχει τις κλήσεις συναρτήσεων διακοσμημένων βημάτων που απεικονίζουν το τελικό γράφημα *pipeline graph*.

```
@pipeline(name="binary-classification", experiment="kale-tutorial")
def ml_pipeline(rs=42, iters=100):
    """Execution of the pipeline"""
    x, y = load(rs)
    x, x_test, y, y_test = split(x, y)
    train(x, x_test, y, iters)
```

Οι συναρτήσεις *load*, *split* και *train* είναι όλες step decorated. Το Kale θα ανακαλύψει αυτόματα τη σειρά και τις εξαρτήσεις μεταξύ των βημάτων και θα παράγει το σχετικό γράφημα. Τελικά, η συνάρτηση pipeline έχει ως τελικό στόχο την αναπαράσταση της τελικής ροής εργασίας ML.

Ο παραπάνω decorator δημιουργεί ένα νέο αντικείμενο Pipeline που αποτελεί μέρος του Kale. Έπειτα, το Kale καλείται να μεταγλωττίσει το object αυτό σε ένα Kubeflow Pipeline Component και στη συνέχεια να παράξει με χρήση της κατάλληλης διεπαφής του KFP το τελικό Argo Workflow.

3.7 Kale Δομικά Στοιχεία

3.7.1 Εκτέλεση Kubeflow Pipelines με Kale & Build Image SDK

Ως πρώτο βήμα, πρέπει να ενσωματώσουμε τον μηχανισμό κατασκευής εικόνων που αναπτύξαμε στο κεφάλαιο 3.4 και να αποσυνδέσουμε την εκτέλεση του τελευταίου από το Rok. Μέχρι τώρα, το Kale προσέφερε έναν τρόπο για την αυτόματη δημιουργία pipelines με το Rok λαμβάνοντας στιγμιότυπτα του χώρου εργασίας του χρήστη και συνδέοντάς τα με τα Pods όπου θα εκτελούνται τα βήματα. Με αυτόν τον τρόπο, το Kale κατάφερε να μεταφέρει τον κώδικα και τα δεδομένα του χρήστη στο περιβάλλον εκτέλεσης του αντίστοιχου βήματος. Για την αποσύνδεση του Kale από το Rok, τα KFP components απαιτούν μια εικόνα Docker που θα περιέχει τις απαραίτητες εξαρτήσεις για την εκτέλεση (δηλαδή τον πηγαίο κώδικα). Όπως αναφέραμε προηγουμένως, ο ορισμός ενός Kubeflow component περιλαμβάνει το Docker Image που είναι υπεύθυνο για την εκτέλεση του σε έναν container.

Συγκεκριμένα, η εικόνα αντικατοπτρίζει τα περιεχόμενα ενός container, καθώς το σύστημα αρχείων του container καθορίζεται από αυτήν. Επομένως, το σύστημα αρχείων του component και εν συνεχεία η νέα εικόνα, πρέπει να περιέχει το αρχείο πηγής και όλες τις απαιτούμενες βιβλιοθήκες, τον κώδικα και τα δεδομένα που εξαρτώνται από αυτό το αρχείο. Για να το πετύχουμε αυτό, επεκτείνουμε το SDK του Kale ώστε να μεταγλωττίζει και να εκτελεί τον κώδικα του χρήστη ως Kubeflow Pipeline χωρίς τον μηχανισμό στιγμιότυπων, αλλά αξιοποιώντας τη βιβλιοθήκη build image που αναπτύξαμε στο 3.4. Η ευθύνη του Kale είναι να κατασκευάσει μια νέα εικόνα για τα βήματα της ροής εργασίας κάνοντας ό,τι καλύτερο μπορεί για να πληροί τις ελάχιστες απαιτήσεις για την εκτέλεση του pipeline. Για τον σκοπό αυτό, συγχωνεύουμε τον μηχανισμό build image που αναπτύξαμε με το Kale.

Από τη σκοπιά του Επιστήμονα Δεδομένων, το μόνο που χρειάζεται είναι να αναπτύξουν τον κώδικα ML με τους κατάλληλους step & pipeline decorators. Στη συνέχεια, εισάγουμε ένα καινούργιο όρισμα στο Command Line Interface του Kale, το οποίο ονομάζεται --build-image, όπου μαζί με το Kale θα αναλάβει όλες τις απαραίτητες διαδικασίες για τη μετατροπή του κώδικά μας σε ένα KFP pipeline, κατασκευάζοντας το απαραίτητο Docker Image για αυτόν.

```
@step(name="step1")
def add(x: int, y: int) -> int:
    return x + y


@pipeline(name="my-pipeline")
def my_pipeline(x = 42: int):
    res = add(x, 17)


if __name__ == "__main__":
    my_pipeline()
```

Συγκεκριμένα, από τώρα και στο εξής το Kale προσφέρει μια νέα επιλογή για την παραγωγή KFP components και την εκτέλεση ενός pipeline ξεκινώντας από μια συνάρτηση Python:

```
$ python3 -m kale script.py --compile --build-image
```

1. Το Kale ανακτά τη βασική εικόνα του Notebook Server όπου αναπτύσσουμε τον κώδικά μας.

2. Από προεπιλογή, το Kale δημιουργεί μια νέα εικόνα, ξεκινώντας από το περιβάλλον του χρήστη κάνοντας ό,τι μπορεί για να ικανοποιήσει τις ελάχιστες απαιτήσεις για τη δημιουργία και την εκτέλεση του pipeline. Έτσι, εκτός από το πηγαίο αρχείο το Kale περιλαμβάνει επίσης στο σύστημα αρχείων του component:

    - Τον κώδικάς μας → Το Kale υποθέτει ότι ζει κάτω από το CWD/src
    - Τα δεδομένα μας → Το Kale υποθέτει ότι βρίσκονται στο CWD/data
    - Εγκατεστημένες βιβλιοθήκες Python → Ζουν στο $HOME/.local

Μπορεί να υπάρχουν περιπτώσεις όπου η προεπιλεγμένη συμπεριφορά του Kale δεν ικανοποιεί τις ανάγκες μας, και θέλουμε να προσθέσουμε συγκεκριμένα αρχεία στην τ μπορούμε να χρησιμοποιήσουμε τη σημαία *--build-path* για να συμπεριλάβουμε στην τελική εικόνα τα αρχεία/φακέλους της επιλογής μας και να αντικαταστήσουμε τα προεπιλεγμένα του Kale. Για παράδειγμα, ο πηγαίος κώδικας και τα δεδομένα μας βρίσκονται όλα στο $HOME/fabulous-app. Η νέα εντολή θα είναι η εξής:

```
$ python3 -m kale script.py --compile --build-image --build-path /home/user/
fabulous-app
```

Το Kale δίνει ένα όνομα στην εικόνα από προεπιλογή, αλλά μπορούμε πάντα να το αλλάξουμε αυτό με τη σημαία *--build-image-name*. Η γνωστή κατάλληλη μορφή είναι *<name>:<tag>*. Σε αυτό το σημείο, το Kale χρησιμοποιεί ένα ιδιωτικό μητρώο Docker για να αποθηκεύσει το πρόσφατα κατασκευασμένο image και να αφήσει το kubelet να τρέξει ένα container (π.χ. ένα βήμα) με αυτό. Μπορούμε να χρησιμοποιήσουμε το δικό μας μητρώο θέτοντας τη σημαία *--registry-host*. Όταν απαιτείται από το μητρώο μας πρέπει επίσης να ορίσουμε τα μονοπάτια *--registry-auth-file* και *--registry-cert-file* για να συνδεθούμε και να επικοινωνήσουμε με ασφάλεια με αυτό.

```
$ python3 -m kale /home/jovyan/kale_sdk.py --kfp --build-image \
--build-image-name my-pipeline:v1 --registry-host gcr.io/my-registry/ \
--registry-auth-file /home/user/auth.json
```

Όλες οι παραπάνω ρυθμίσεις μπορούν να συνοψιστούν στα παρακάτω ορίσματα:

1. *--build-image*: Κατασκευή της εικόνας που θα χρησιμοποιήσει το μεταγλωττισμένο pipeline για την εκτέλεση των βημάτων του.

2. *--build-path*: Διαδρομή σε ένα αρχείο/φάκελο που θα αποτελεί μέρος του συστήματος αρχείων όπου θα εκτελείται το pipeline

3. *--build-image-name*: Το όνομα της εικόνας που θα κατασκευάσει το Kale. Η αναμενόμενη μορφή είναι η εξής: <name>:<tag>

4. *--registry-host*: Το μητρώο στο οποίο θα βρίσκεται η εικόνα

5. *--registry-auth-file*: Διαδρομή προς το αρχείο ελέγχου ταυτότητας των διαπιστευτηρίων για τη σύνδεση στο μητρώο.

6. *--registry-cert-file*: Διαδρομή προς το αρχείο πιστοποίησης (.crt, *.cert,*.key) για τη σύνδεση στο μητρώο.

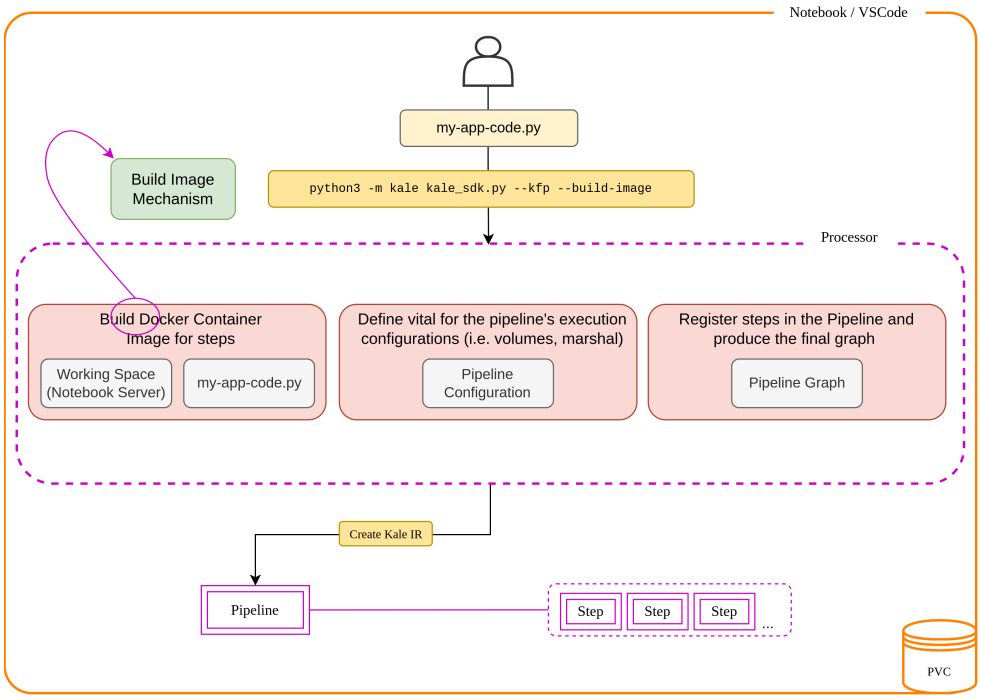
Κάθε ένα από αυτά έχει μια αντιπροσωπευτική μεταβλητή περιβάλλοντος η οποία μπορεί να διαμορφωθεί και να αποφασίσει για τις προηγμένες ρυθμίσεις της εικόνας κατασκευής σε περίπτωση που τα αντίστοιχα ορίσματα CLI είναι κενά:

- --build-image-name → KALE_BUILD_IMAGE_NAME
- --registry-host → KALE_REGISTRY_HOST
- --registry-auth-file → KALE_REGISTRY_AUTH
- --registry-cert-file → KALE_REGISTRY_CERT

Σε σύγκριση με αυτό που κάνει το Kubeflow Pipelines αντιλαμβανόμαστε αμέσως την ευκολία που παρέχει η συνεισφορά μας. Οι επιστήμονες δεδομένων δεν χρειάζεται πια να παρεμβαίνουν σε λειτουργίες σχετικές με Docker images, containers και Kubernetes.

3.7.2 Κατασκευή KFP Δομικών Στοιχείων

Η εκτέλεση των Kubeflow Pipelines με τον μηχανισμό μας για την κατασκευή εικόνων ήταν μόνο η αρχή. Όπως αναφέρθηκε στην εισαγωγή της διπλωματικής εργασίας, ο απώτερος στόχος μας είναι να δημιουργήσουμε επαναχρησιμοποιούμενα και αναπαραγώγιμα δομικά στοιχεία KFP. Με αυτό εννοούμε όχι μόνο να κατασκευάζουμε την



Σχήμα 3.8: *Διαδρομή εκτέλεσης της ροής εργασίας Kale με χρήση του μηχανισμού Build Image. Τώρα, το μέρος της επεξεργασίας περιέχει την κατασκευή μιας εικόνας για τα βήματα του pipeline*

εικόνα για το component αλλά και να εξάγουμε τον ορισμό του στο τοπικό σύστημα αρχείων για να το μπορεί να διαμοιραστεί σε άλλους επιστήμονες δεδομένω. Έτσι, απλώς η μεταγλώττιση και η εκτέλεση ενός pipeline με τον μηχανισμό δημιουργίας εικόνων δεν είναι αρκετή.

Θέλουμε να δημιουργήσουμε μια κοινότητα μηχανικών ML που θα μπορούν εύκολα να μοιράζονται, να ανακαλύπτουν και να επαναχρησιμοποιούν στοιχεία ροής εργασιών. Το Kubeflow, ενισχυμένο με το Kale, θα είναι η de facto πλατφόρμα ML που θα φέρνει κοντά τους επιστήμονες δεδομένων για να αντιμετωπίσουν τις πιο δύσκολες εργασίες ML με τον καλύτερο δυνατό τρόπο και να τις μοιραστούν με την κοινότητα. Οι μηχανικοί ML θα είναι σε θέση να αποθηκεύουν και να επαναχρησιμοποιούν τα στοιχεία KFP σε διάφορους διαφορετικούς συνδυασμούς εξοικονομώντας πολύτιμο χρόνο προς την κατεύθυνση της εύρεσης της καλύτερης ροής εργασίας ML για τις ανάγκες τους.

Ο τελικός μας στόχος είναι να παράγουμε ένα KFP component ξεκινώντας από μια συνάρτηση Python που θα είναι αυτόνομο, αναπαραγώγιμο, επαναχρησιμοποιούμενο και μεταβιβάσιμο σε άλλα περιβάλλοντα. Σε αυτή τη διατριβή περιγράφουμε έναν τρόπο για να:

1. Μεταγλωττίσουμε μια συνάρτηση Python σε ένα επαναχρησιμοποιούμενο KFP δομικό στοιχείο που περιέχει:

   (α′) Εισόδους και εξόδους

   (β′) Εικόνα Docker που θα χρησιμοποιηθεί για την εκτέλεσή του

   (γ′) Εντολή εκκίνησης του container που θα εκτελεστεί το component

2. Εκτέλεση μεμονωμένα ενός KFP component σε οποιοδήποτε pipeline ανεξάρτητα από το πού αναπτύχθηκε ή πού μπορεί να καταλήξει να βρίσκεται.

Ο Επιστήμονας Δεδομένων ως το άτομο που αναπτύσσει, δοκιμάζει και τελικά κάνει deploy έναν ML pipeline θα κάνει τα εξής:

1. Αναπτύσσει μια ροή εργασίας ML που αποτελείται από ένα pipeline με έναν αριθμό βημάτων.

   (α′) Έχει την επιλογή να εισάγει components από άλλους μηχανικούς που θα ταιριάζουν στις ανάγκες του

   (β′) Δοκιμάζει διαφορετικές ροές εργασίας συνδυάζοντας έτοιμα προς χρήση components στο pipeline του

2. Όταν η ροή εργασίας του είναι έτοιμη για παραγωγή δίνει εντολή στο Kale να:

   (α′) Παράγει ένα επαναχρησιμοποιούμενο component KFP ξεκινώντας από μια συνάρτηση python.

   (β′) Μεταγλωττίσει, μεταφορτώσει και εκτελέσει την αντίστοιχη ροή εργασίας Μηχανικής Μάθησης.

Στην παρούσα διπλωματική εργασία υλοποιούμε έναν τρόπο για την απρόσκοπτη μετατροπή μιας συνάρτησης Python σε ένα Kubeflow Pipeline Component και την ανάπτυξή του στο Kubeflow χρησιμοποιώντας το Kale SDK. Συγκεκριμένα, εισάγουμε μια νέα συνάρτηση για το σκοπό αυτό:

- save_component()

Η συνάρτηση *save_component()* μετατρέπει μια step decorated συνάρτηση σε ένα αυτόνομο δομικό στοιχείο KFP. Οι τύποι των ορισμάτων της εισόδου και της εξόδου της συνάρτησης χρησιμοποιούνται ως τύποι εισόδου/εξόδου του δομικού στοιχείου. Έπειτα, μια νέα εικόνα Docker δημιουργείται για το component. Προαιρετικά, η μέθοδος δέχεται προηγμένες ρυθμίσεις που σχετίζονται με την εικόνα του δομικού στοιχείου, όπως το περιεχόμενό της, το όνομα και το μητρώο που πρέπει να προωθηθεί μαζί με τα απαιτούμενα αρχεία ελέγχου ταυτότητας ή/και πιστοποίησης. Εάν δεν παρέχεται, το Kale υποθέτει ότι οι σχετικές τιμές μπορούν να βρεθούν είτε από τις αντίστοιχες μεταβλητές περιβάλλοντος είτε από το CLI.

```
def save_component(save_path: str = None,
                   **build_kwargs):
```

Τα ορίσματα της συνάρτησης αναλυτικά είναι τα εξής:

- *save_path*: Διαδρομή προς ένα αρχείο όπου θα αποθηκευτεί ο ορισμός του δομικού στοιχείου. Εάν είναι κενό, το αρχείο αποθηκεύεται στο CWD/<όνομα στοιχείου>.kale.yaml
- ***build_kwargs*: Ορίσματα λέξης-κλειδιά που σχετίζονται με τις προηγμένες ρυθμίσεις της εικόνας που θα κατασκευαστεί (π.χ. περιεχόμενο, όνομα, μητρώο)

Η συνάρτηση *save_component()* επιστρέφει το ενημερωμένο instance του step που περιλαμβάνει τη νέα εικόνα Docker. Αυτή η περίπτωση μπορεί να χρησιμοποιηθεί απευθείας σε ένα pipeline function και να γίνει τελικά μέρος του γράφου μιας ροής εργασίας. Ένα απλό παράδειγμα χρήσης της *create_component()* παρουσιάζεται παρακάτω:

```
>>> @step(name="add")
>>> def add(a: float, b: float) -> float:
>>> return a + b
>>>
>>> add.save_component(
>>>     save_path="add.component.yaml",
>>>     build_paths=["./component_dep"],
>>>     build_image_name="my-component:v1",
>>>     registry_host="docker.io/my-registry"
>>> )
```

```
>>> add.docker_image
docker.io/my-registry/my-component:v1@<digest>
```

Μετά την επιτυχή εκτέλεση αυτής της εντολής, μπορούμε να αναζητήσουμε το αρχείο YAML του component. Η προδιαγραφή YAML θα πρέπει να μοιάζει πολύ με την παρακάτω:

```
name: add
inputs:
- {name=”a”, type=”Integer”}
- {name=”b”, type=”Integer”}
outputs:
- {name=”out”, type=”Integer”}
υλοποίηση:
  δοχείο:
    image: gcr.io/my-org/my-component@sha256:a172..752f
    command: [
      python3,
      -u,
      -m,
      kale,
      /home/jovyan/script.py::add
   ]
   args:
     --in
     - a
     - {inputValue: a}
     --in
     - b
     - {inputValue: b}
     --out
     - out
     - {outputPath: out}
```

Με μία μόνο εντολή και ελάχιστες αλλαγές στον αρχικό κώδικα καταλήγουμε σε ένα αυτόνομο, αναπαραγώγιμο και διαμοιραζόμενο KFP δομικό στοιχείο. Αναμφισβήτητα, η συνεισφορά μας αφαιρεί την ταλαιπωρία του KFP για την παραγωγή KFP δομικών στοιχείων, αφού όλη η διαδικασία αντικαθίσταται με μία μόνο εντολή.

3.7.3 Ενσωμάτωση Εξωτερικών Δομικών Στοιχείων σε Σωληνώσεις

Τώρα είμαστε σε θέση να μοιραστούμε το component μας με άλλους επιστήμονες ή να ανακτήσουμε ένα τέτοιο component από έναν άλλο επιστήμονα δεδομένων και να το χρησιμοποιήσουμε στο pipeline μας. Για να το κάνουμε αυτό, επεκτείνουμε το Kale με την ακόλουθη μέθοδο:

- kale.common.componentutils.load_component()

Αυτή η συνάρτηση λαμβάνει μια διαδρομή προς ένα αρχείο yaml που αναπαριστά τις προδιαγραφές του component και επιστρέφει ένα step instance Kale που μπορούμε να χρησιμοποιήσουμε στο pipeline μας. Το ακόλουθο παράδειγμα δείχνει πώς μπορεί να ενσωματώσει κανείς ένα εξωτερικό component KFP μέσω της προδιαγραφής του εβρισκόμενη σε ένα αρχείο YAML και να το εκτελέσει σε ένα pipeline δύο βημάτων.

1. Στο σενάριο python που ορίζει τη ροή εργασίας μας ML μπορούμε να χρησιμοποιήσουμε το εξής:

```
@step(name=”sum”)
def print_num(num: int):
   """Print a number"""
    print(num)

add = load_component(”/path/to/component/yaml”)

@pipeline(name=”my_pipeline”)
def pipeline_func(x1: int = 42):
    res = add(x1, 13)
    print_num(res)
```

2. Αναπτύσσουμε και εκτελούμε τον κώδικά μας ως σωλήνωση KFP:

```
$ python3 -m kale kale_sdk.py --kfp
```

Ας δούμε ξανά πώς μοιάζει το εξωτερικό KFP component:

```
name: add
inputs:
- {name=”a”, type=”Integer”}
```

```
- {name=”b”, type=”Integer”}
outputs:
- {name=”out”, type=”Integer”}
υλοποίηση:
  δοχείο:
    image: gcr.io/my-org/my-component@sha256:a172..752f
    command: [
      python3,
      -u,
      -m,
      kale,
      /home/jovyan/script.py::add
   ]
   args:
     --in
     - a
     - {inputValue: a}
     --in
     - b
     - {inputValue: b}
     --out
     - out
     - {outputPath: out}
```

Παρατηρήστε τα KFP placeholders *inputValue* και *outputPath*. Δεν έχουμε καλέσει ακόμα τη συνάρτηση του component, πράγμα που σημαίνει ότι οι τιμές των ορισμάτων της είναι ακόμα άγνωστες.

Ο χρήστης μεταγλωττίζει και εκτελεί τη ροή εργασίας του ML με το *’python3 -m kale kale_sdk.py --kfp’*. Πριν από την επεξεργασία και τη μεταγλώττιση του pipeline, **φορτώνουμε τα εξωτερικά δομικά στοιχεία** και τα μετατρέπουμε σε step objects, ανακτώντας από αυτά: τις εισόδους/εξόδους με τους τύπους τους, το όνομα, την εικόνα docker και τη διαδρομή πηγής. Το step instance που παράγεται από το παραπάνω δομικό στοιχείο KFP είναι:

```
name: add
ins: (
    a, {
        name: a,
```

```
            param_type: Integer,
            param_value: None,
            source: None
        },
        b, {
            name: b,
            param_type: Integer,
            param_value: None,
            source: None
        }
    )
    outs: (
        out, {
            name: out
            param_type: Integer,
            param_value: None,
            source: None
        }
    )
    source_path: /home/jovyan/script.py
    docker_image: gcr.io/my-org/my-component@sha256:a172..752f
    marshal_path: None
```

Προφανώς, ένα step instance δεν διαφέρει πολύ από ένα component. Θα μπορούσαμε να πούμε ότι ένα Kale step είναι ένα υπερσύνολο ενός δομικού στοιχείου KFP που περιέχει πρόσθετες πληροφορίες που απαιτούνται για το Kale ώστε να κάνει τη ζωή του μηχανικού ML ευκολότερη. Στη συνέχεια, το Kale ακολουθεί την ίδια διαδρομή για την επεξεργασία, τη μεταγλώττιση και την εκτέλεση του pipeline.

4 Υλοποίηση

4.1 Επισκόπηση

Σε αυτό το κεφάλαιο αποκαλύπτουμε τα μυστικά του συστήματός μας δίνοντας μια λεπτομερή περιγραφή της βιβλιοθήκης Build Image μαζί με τη διεπαφή δημιουργίας επαναχρησιμοποιούμενων και αναπαραγώγιμων στοιχείων KFP. Στο προηγούμενο κεφάλαιο παρουσιάσαμε κάποιο από το θεωρητικό υπόβαθρο, καθώς και τη βασική σχεδιαστική λογική και τη συνολική αρχιτεκτονική και των δύο συστημάτων. Ωστόσο, υπάρχουν αρκετές πτυχές που δεν συζητήθηκαν. Η διαδικασία υλοποίησης κάθε συστήματος πραγματοποιήθηκε σε πολλαπλές επαναλήψεις, όπως και ο σχεδιασμός τους. Μέρος αυτού του κεφαλαίου είναι τα εμπόδια που συναντήσαμε κατά την προσπάθεια να εφαρμόσουμε τις σχεδιαστικές μας αποφάσεις στην πράξη και τα βήματα που ακολουθήσαμε για να τα ξεπεράσουμε. Επιπλέον, περιγράφουμε μερικά από τα patches του Kale που γράψαμε για να υποστηρίξουμε το σύστημά μας.

Το μεγαλύτερο μέρος του κώδικα είναι γραμμένο στη γλώσσα προγραμματισμού Python. Παρακάτω έχουμε συμπεριλάβει μόνο μερικά επιλεγμένα τμήματα κώδικα και συγκεκριμένα σε μορφή ψευδοκώδικα, προκειμένου να βοηθήσουμε τον αναγνώστη να κατανοήσει την υλοποίησή μας. Για να αποφύγουμε την πολυπλοκότητα ορισμένων λειτουργιών, καθώς είναι άσχετη με το σκοπό. Η όλη υλοποίηση αποτελείται από δύο σημαντικές ενότητες Python που επεκτείνουν το Kale SDK και εισάγονται ως βοηθητικά προγράμματα:

- *imageutils*: Μια σουίτα βοηθητικών προγραμμάτων που σχετίζονται με την επεξεργασία container images. Περιέχει κυρίως λειτουργικότητα για το χειρισμό

αποσυμπιεσμένων εικόνων Docker και OCI και την επέκτασή τους με νέα layers.

- *componentutils*: Μια σουίτα βοηθητικών προγραμμάτων που σχετίζονται με τα KFP components. Περιέχει κυρίως λειτουργικότητα γύρω από τα δομικά στοιχεία KFP (π.χ. δημιουργία KFP components, φόρτωση εξωτερικών component και άλλα)

4.2 Τα Μυστικά του Μηχανισμού Δημιουργίας Εικόνων

Το εξωτερικό επίπεδο αυτού του μηχανισμού είναι η συνάρτηση *build_image*. Πρώτα απ' όλα, ο ορισμός της συνάρτησης είναι ο εξής:

```
def build_image(base_image: str,
                include_paths: List[Union[str, Tuple[str, str]]],
                dst_image: str,
                src_auth_file: str = None,
                src_cert_dir: str = None,
                dst_auth_file: str = None,
                dst_cert_dir: str = None) -> str:
```

Σχήμα 4.1: *Ο ορισμός της συνάρτησης build image*

Αυτή η συνάρτηση δημιουργεί μια νέα εικόνα από μια βασική εικόνα και ένα σύνολο αρχείων/φακέλων. Δέχεται ένα έγκυρο όνομα εικόνας βάσης που βρίσκεται είτε σε κάποιο ιδιωτικό είτε σε κάποιο δημόσιο μητρώο και ένα σύνολο αρχείων και καταλόγων που θα προστεθούν στο σύστημα αρχείων της νέας εικόνας. Δέχεται επίσης τα κατάλληλα διαπιστευτήρια και πιστοποιητικά για έλεγχο ταυτότητας, όταν απαιτείται από το μητρώο.

Συγκεκριμένα τα βήματα που ακολουθεί είναι τα εξής:

1. Φέρνει στο σύστημα αρχείων τη βασική εικόνα (που καθορίζεται από το όνομα, την ετικέτα και το άθροισμα ελέγχου της)
2. Επεκτείνει την εικόνα προσθέτοντας τα παρεχόμενα αρχεία και καταλόγους στο σύστημα αρχείων της εικόνας
3. Μεταφέρει τη νέα εικόνα σε ένα μητρώο προορισμού

Τα ορίσματα της συνάρτηση είναι:

- *base_image(str)*: Μια έγκυρη βασική εικόνα με τη μορφή: *<registry>/<name>:<tag>@digest*. Το digest είναι προαιρετικό και η ετικέτα έχει ως προεπιλογή την τιμή latest. Είναι ιδιαίτερα εύκολο να ξεκινήσετε χρησιμοποιώντας μια βασικήκ εικόνα από ένα δημόσιο μητρώο όπως το docker hub.
- *include_paths(List[Union[str, Tuple[str, str]]])*: μια λίστα μονοπατιών ή/και πλειάδων ενός μονοπατιού πηγής και ενός μονοπατιού προορισμού
- *dst_image(str)*: Ένα έγκυρο όνομα για τη νέα εικόνα με τη μορφή: *<registry>/<name>:<tag>*
- *src_auth_file(str)*: Μια διαδρομή προς το αρχείο ελέγχου ταυτότητας για τη σύνδεση με το μητρώο πηγής (προαιρετικό).
- *src_cert_dir(str)*: Μια διαδρομή προς το αρχείο πιστοποίησης για μια ασφαλή και αξιόπιστη σύνδεση με το μητρώο προορισμού (προαιρετικό).
- dst_auth_file(str): Μια διαδρομή προς το αρχείο πιστοποίησης για τη σύνδεση με το μητρώο πηγής (προαιρετικό).
- dst_cert_dir(str): Μια διαδρομή προς το αρχείο πιστοποίησης για μια ασφαλή και αξιόπιστη σύνδεση με το μητρώο προορισμού (προαιρετικό).

Η συνάρτηση επιστρέφει το όνομα της νέας εικόνας με τη μορφή
<registry>/<name>:<tag>@digest.
Ακολουθεί η υλοποίηση σε μορφή ψευδοκώδικα.

```
def build_image(...) -> str:
    # Pulling image
    copy_image(from_registry, to_local_fs)

    # Επέκταση της εικόνας
    extended_image = extend_image(base_image_path, include_paths)

    # Pushing image
    copy_image(from_local_fs, to_registry)

    # Η νέα εικόνα κατασκευάστηκε επιτυχώς
    return dst_image + "@" + extended_image.image_digest
```

Σχήμα 4.2: *Η υλοποίηση της συνάρτησης build_image() σε μορφή ψευδοκώδικα*

Όπως είναι φανερό, καλούμε την *copy_image()* για την μεταφορά της εικόνας μεταξύ του τοπικού συστήματος αρχείων αλλά και απομακρυσμένων μητρώων (λειτουργίες pull και push). Αυτή η συνάρτηση αντιγράφει μια εικόνα από μια πηγή σε μια θέση προορισμού χρησιμοποιώντας το εργαλείο Skopeo. Το Skopeo λειτουργεί με μητρώα των εικόνων περιεκτών μορφής API V2, όπως τα μητρώα docker.io και quay.io, ιδιωτικά μητρώα, τοπικούς καταλόγους και τοπικούς καταλόγους με μορφή OCI. Όταν απαιτείται από το μητρώο, το Skopeo μπορεί να περάσει τα κατάλληλα διαπιστευτήρια και πιστοποιητικά για έλεγχο ταυτότητας.

Το 'skopeo copy' αντικαθιστά αυτό:

```
docker pull internal.registry/myimage:latest
docker tag internal.registry/myimage:latest prod.registry/myimage:v1
docker push production.registry/myimage:v1.0
```

με αυτό:

```
skopeo copy docker://internal.registry/myimage:latest \
            docker://production.registry/myimage:v1
```

με ένα σημαντικό πλεονέκτημα: Δεν χρειάζεται να έχετε δικαιώματα root ή να έχουμε εγκαταστήσει δαίμονα docker. Δυστυχώς, το Skopeo δεν παρέχει ένα Python SDK, επομένως πρέπει να χρησιμοποιήσουμε το εκτελέσιμο αρχείο απευθείας. Αργότερα σε αυτό το κεφάλαιο θα δούμε έναν εναλλακτικό και πιο αποτελεσματικό τρόπο για να φέρουμε τοπικά αλλά και να ανεβάσουμε Docker Images σε απομακρυσμένα μητρώα για τον σκοπό μας, χρησιμοποιώντας το Docker Registry API.

4.2.1 Επέκταση Εικόνων Docker

Σε αυτό το σημείο εξετάζουμε την συνάρτηση *extend_image*. Αυτή η συνάρτηση αυτοματοποιεί όλη τη διαδικασία που εξηγήσαμε διεξοδικά στην Ενότητα 3.3. Ο ορισμός της συνάρτησης είναι ο εξής:

```
def extend_image(base_image_path: str,
                 include_paths: List[Union[str, Tuple[str, str]]]
                 ) -> DockerImage:
```

Δέχεται μια βασική εικόνα που είναι μια διαδρομή προς τα αρχεία μιας έγκυρης Docker εικόνας και την επεκτείνει προσθέτοντας ένα νέο επίπεδο που αποτελείται από μια δέ-

σμη αρχείων ή/και φακέλων. Η διαδικασία που ακολουθείται από τη συνάρτηση είναι η εξής:

1. Κατασκευάζει το tarball που αποτελείται από τα παρεχόμενα αρχεία/φακέλους
2. Προσθέτει ένα νέο επίπεδο πάνω στη βασική εικόνα χρησιμοποιώντας το παραπάνω tarball ακολουθώντας τον αλγόριθμο που δείξαμε στην Ενότητα 3.3
3. Επιστρέφει τη νέα εικόνα

Τα ορίσματα είναι τα εξής:

- *base_image_path(str)*: Μια συμβολοσειρά που αντιπροσωπεύει τη διαδρομή προς μια έγκυρη εικόνα Docker 2ης έκδοσης και σχήματος 2.
    - *path* μπορεί να είναι είτε: (i) μια απόλυτη διαδρομή είτε (ii) μια σχετική διαδρομή.
- *include_paths(list[str | tuple(str, str)])*: Μια λίστα με διαδρομές προέλευσης και προορισμού για να δηλώσουμε ποια αρχεία/φακέλους από το τοπικό σύστημα αρχείων πρέπει να αντιγραφούν στο νέο στρώματ της εικόνας και σε ποια θέση.
    - Η *πηγή* μπορεί να είναι είτε απόλυτη είτε σχετική διαδρομή
    - Ο *προορισμός* μπορεί να είναι είτε:
        * *Κενός*: η απόλυτη διαδρομή της πηγής θα προστεθεί ως έχει στο νέο επίπεδο
        * *Μια απόλυτη διαδρομή προορισμού*: ο φάκελος/το αρχείο που υποδεικνύεται από τη διαδρομή πηγής θα προστεθεί κάτω από τη διαδρομή προορισμού.

Στη συνάρτηση *extend_image()* κάνουμε τα εξής:

1. Αρχικά επικυρώνουμε τη διαδρομή της βασικής εικόνας, δηλαδή βεβαιωνόμαστε ότι η διαδρομή υπάρχει. Η συνάρτηση *os.path.exists()* [23] είναι το δεξί μας χέρι για αυτό.
2. Με αυτή τη βασική εικόνα, δημιουργούμε ένα καινούργιο στιγμιότυπο της κλάσης *DockerImage*.

3. Στη συνέχεια, επεξεργαζόμαστε και επικυρώνουμε τα μονοπάτια που θα αποτελέσουν το νέο επίπεδο της εικόνας.

4. Τέλος, δημιουργούμε ένα νέο αρχείο tar το οποίο τελικά προσαρτάται στη βασική εικόνα.

Παρακάτω παρουσιάζεται η υλοποίηση της συνάρτησης με μορφή ψευδοκώδικα:

```
def extend_image(...) -> DockerImage:
    # Επικύρωση της βασικής εικόνας
    validate_base_image()
    image = DockerImage(base_image_abspath)

    # Επεξεργασία των μονοπατιών που θα αποτελούν το αρχείο tar
    tar_members = process_include_paths(include_paths)

    # Δημιουργία αρχείου tar
    tar_file = build_tar(tar_members)

    # Προσθήκη νέου επιπέδου
    image.append_layer(tar_file)
    return image
```

Η συνάρτηση *process_include_paths()* επικυρώνει και επεξεργάζεται μια λίστα μονοπατιών. Συγκεκριμένα, παράγει έγκυρες πλειάδες μονοπατιών πηγής και προορισμού που προορίζονται για τα ορίσματα της διαδικασίας δημιουργίας ενός tar archive. Η πηγή αντιπροσωπεύει τη διαδρομή προς ένα αρχείο στο τοπικό σύστημα αρχείων και ο προορισμός τη διαδρομή εντός του tar. Ακολουθεί ένα απλό παράδειγμα χρήσης της συνάρτησης:

```
>>> include_paths = [”/home/test”]
>>> processed_paths = _process_include_paths(include_paths)
>>> print(processed_paths)
[(”/home/test”, ”/home/test”)]

>>> include_paths = [(”/home/test”, ”/hello/world”)]
>>> processed_paths = _process_include_paths(include_paths)
>>> print(processed_paths)
[(”/home/test”, ”/hello/world”)]
```

Στο τέλος, οι συναρτήσεις *tarfile.open* [24] μαζί με *tar.add* [25] θα λάβουν μια απόλυτη/σχετική διαδρομή προς το αρχείο προέλευσης και μια απόλυτη/σχετική διαδρομή προς τον προορισμό μέσα στο αρχείο tar. Ωστόσο, οι χρήστες μπορεί να μην θέλουν να δώσουν διαφορετική διαδρομή προορισμού μέσα στο tar, οπότε καταφέρνουμε να προσαρμόσουμε ανάλογα την είσοδό τους σε αυτό που περιμένει η συνάρτηση tar.add. Κατά τη διάρκεια της υλοποίησης αυτού του μηχανισμού, προσπαθήσαμε να τον κάνουμε εύχρηστο και φιλικό προς το χρήστη, επιτρέποντας διαφορετικές παραλλαγές των εισόδων και αναφέροντας ένα ενημερωτικό σφάλμα όταν η μορφή εισόδου δεν είναι σωστή.

4.2.2 Η Κλάση DockerImage

Πριν, αναφερθήκαμε σε ένα αντικείμενο με όνομα *DockerImage*. Η κλάση *DockerImage* αναπαριστά ένα Docker Image σε αποσυμπιεσμένη μορφή. Μια τυπική δομή μιας εικόνας Docker είναι η εξής:

```
image/
├── <configuration_digest> <- Αρχείο JSON διαμόρφωσης
├── <layer1_digest>
├── <layer2_digest>
    .
    .
├── <layerN_digest> <- αρχείο tar.gz
├── manifest.json
└── έκδοση
```

Όπως ήδη γνωρίζουμε, το αρχείο manifest είναι η υπογραφή της εικόνας που περιέχει μεταδεδομένα (μέγεθος και άθροισμα ελέγχου) σχετικά με το αρχείο ρυθμίσεων και τα layers της εικόνας. Το αρχείο ρυθμίσεων είναι μια ταξινομημένη συλλογή αλλαγών στο ριζικό σύστημα αρχείων και τις αντίστοιχες παραμέτρους εκτέλεσης για χρήση μέσα σε ένα χρόνο εκτέλεσης του container.

Αυτή η κλάση λειτουργεί σαν καθρέφτης για μια εικόνα Docker σε αποσυμπιεσμένη μορφή. Συγκεκριμένα:

1. Φορτώνει από το τοπικό σύστημα βασικά συστατικά της εικόνας

   (α′) Manifest αρχείο

(β′) Configuration αρχείο

2. Ανακτά διάφορα χαρακτηριστικά της εικόνας:

   (α′) Το μέγεθος και άθροισμα ελέγχου του n-οστού layer της εικόνας

   (β′) Το άθροισμα ελέγχου της εικόνας

3. Προσθέτει ένα νέο επίπεδο στην κορυφή της εικόνας

Ορισμένες από τις πιο σημαντικές μεθόδους της κλάσης είναι οι εξής:

- *read_manifest_file() και read_config_file()* οι οποίες επιστρέφουν τα ακατέργαστα δεδομένα κάθε αρχείου αντίστοιχα. Και οι δύο κάνουν χρήση της συνάρτησης *open()* [26] Python.
- *get_layer_digest() και get_layer_len()* οι οποίες επιστρέφουν το μήκος και το μέγεθος του n-οστού επιπέδου. Αυτά τα μεταδεδομένα γράφονται στο αρχείο manifest και στο αρχείο ρυθμίσεων. Η συνάρτηση δεν κάνει τίποτα περισσότερο από το άνοιγμα των αρχείων και την ανάκτηση των σχετικών τιμών από τα αντίστοιχα πεδία.
- *append_layer()* η οποία προσθέτει ένα νέο επίπεδο στην κορυφή της εικόνας.

4.2.3 Προσθήκη Νέων Επιπέδων σε Εικόνες

Ο τρόπος για να προσαρτήσουμε ένα επίπεδο πάνω σε μια εικόνα είναι να επεξεργαστούμε το manifest και το configuration αρχείο. Αυτό είναι κάτι που συνειδητοποιήσαμε στα προηγούμενα κεφάλαια. Συγκεκριμένα, το αρχείο manifest παρακολουθεί μια λίστα με τα μεγέθη και τα digests των συμπιεσμένων layers και το μέγεθος και το digest του config. Το αρχείο ρυθμίσεων, από την άλλη πλευρά, καταγράφει μια λίστα με τις αλλαγές στο ριζικό σύστημα αρχείων, δηλαδή τα digests των μη συμπιεσμένων στρωμάτων που έχουν αρχειοθετηθεί με tar. Σημειώστε ότι τα ονόματα των αρχείων configuration και layer πρέπει να είναι το digest των δικών τους περιεχομένων.

Κατά συνέπεια, η συνάρτηση αυτή λαμβάνει μια διαδρομή προς ένα αρχείο αρχειοθέτησης tar που αντιπροσωπεύει το νέο στρώμα. Έπειτα, η *append_layer(tar_file)* κάνει τα εξής:

1. Φορτώνει το αρχείο διαμόρφωσης χρησιμοποιώντας τη συνάρτηση read_config_file(). Στη συνέχεια, υπολογίζει το digest του αρχειοθετημένου στρώματος tar με χρήση της βιβλιοθήκης *hashlib* στην Python [27] και το προσθέτει στο αρχείο διαμόρφωσης.

2. Υπολογίζει το μήκος και το digest της ενημερωμένης διαμόρφωσης. Η συνάρτηση os.path.getsize μας βοηθά να υπολογίσουμε το μέγεθος του αρχείου.

3. Συμπιέζει το επίπεδο χρησιμοποιώντας το *gzip* [28] και λαμβάνει το digest και το μήκος του.

4. Τέλος, ενημερώνει το αρχείο manifest και κάνει την απαιτούμενη μετονομασία των αρχείων

```
def append_layer(self, tar_file):
    configuration = self.read_config_file()
    # Append new diff_id under diff_ids field in configuration object
    configuration["rootfs"]["diff_ids"].append(
        compute_file_digest(tar_file))
    config_str = json.dumps(configuration).encode("utf-8")
    config_digest = compute_sha256_digest(config_str)

    zip_layer_digest = self.zip_layer(tar_file)
    zip_layer_len = os.path.getsize(os.path.join(self.path,
                                                 self.LAYER_FILENAME))

    self.update_manifest(len(config_str), config_digest,
                         zip_layer_len, zip_layer_digest)

    save_config_file()

    # Μετονομασία του συμπιεσμένου αρχείου tar στο δικό του digest
    # για να γίνει μέρος της εικόνας
    rename_layer()
```

4.2.4 Docker Registry API vs Skopeo

Το HTTP API του Docker Registry είναι το πρωτόκολλο που διευκολύνει τη διανομή εικόνων στη μηχανή docker. Αλληλεπιδρά με τις περιπτώσεις του docker registry, το

οποίο είναι μια υπηρεσία για τη διαχείριση πληροφοριών σχετικά με τις εικόνες docker και την ενεργοποίηση της διανομής τους.

Το Docker χρησιμοποιεί τη μορφή v2_2, την οποία γνωρίζουμε σε μεγάλη κλίμακα, για την αποθήκευση εικόνων στα μητρώα Docker. Αυτή είναι επίσης η μορφή (μαζί με την εικόνα OCI) που χρησιμοποιούμε για να χειριστούμε, να επεξεργαστούμε και να επεκτείνουμε μια εικόνα. Στην πραγματικότητα, φέρνουμε την εικόνα στο τοπικό σύστημα αρχείων από ένα μητρώο σε μορφή v2_2 ή OCI και στη συνέχεια επεκτείνουμε την εικόνα με την επεξεργασία του manifest και του αρχείου ρυθμίσεων. Αυτή η μέθοδος έχει κάποιες επιπτώσεις, επειδή τα περισσότερα προγράμματα εκτέλεσης container δεν μπορούν να εκτελέσουν έναν container που χρησιμοποιεί αυτή τη μορφή εικόνας. Το Docker χρησιμοποιεί έναν storage driver για την αποθήκευση των layer της εικόνας. Ο πιο κοινός και προτιμώμενος οδηγός αποθήκευσης είναι ο overlay2. Ο οδηγός αποθήκευσης ελέγχει τον τρόπο αποθήκευσης και διαχείρισης των εικόνων και των container στο σύστημα υποδοχής.

Τελικά, για να ολοκληρώσουμε τη διαδικασία δημιουργίας μιας εικόνας, αφού την φέρουμε στο τοπικό fs χρησιμοποιώντας τη μορφή v2_2 / OCI και την επεκτείνουμε τροποποιώντας τα αντίστοιχα αρχεία, πρέπει να προωθήσουμε τη νέα επεκταμένη εικόνα σε ένα μητρώο, είτε τοπικό είτε απομακρυσμένο. Για το σκοπό αυτό, μέχρι τώρα στο μηχανισμό μας χρησιμοποιούσαμε το Skopeo το οποίο αντιγράφει μια εικόνα από μια πηγή σε έναν προορισμό.

Παρόλο που το Skopeo φαίνεται να είναι το ιδανικό εργαλείο για λειτουργίες σχετικές με απομακρυσμένα μητρώα εικόνων, δεν έχει δημιουργηθεί για τη δική μας περίπτωση χρήσης, η οποία είναι η επέκταση εικόνων, αποκαλύπτοντας έτσι ορισμένα μειονεκτήματά του. Συγκεκριμένα, το Skopeo μας αναγκάζει να κατεβάσουμε το πλήρες περιεχόμενο μιας εικόνας, δηλαδή όλα τα συμπιεσμένα επίπεδα, το αρχείο manifest και το αρχείο ρυθμίσεων.

Το απλούστερο endpoint που παρέχει πληροφορίες υποστήριξης του API του Docker Registry είναι τοποθετημένο στη διεύθυνση '/v2/' [29]. Η μορφή του αιτήματος είναι η ακόλουθη:

```
GET /v2/
```

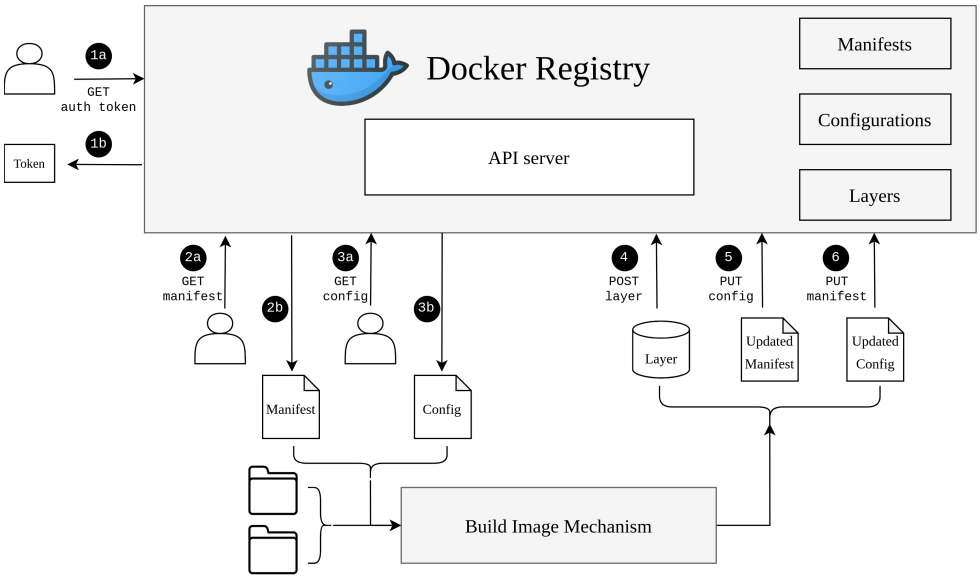
- *Αν επιστραφεί απάντηση 200 OK*, το μητρώο υλοποιεί το API του μητρώου V2(.2)

και ο πελάτης μπορεί να προχωρήσει με ασφάλεια σε άλλες λειτουργίες V2.

- *Αν επιστραφεί η κατάσταση απόκρισης 404 Not Found* ή άλλη απροσδόκητη κατάσταση, ο πελάτης πρέπει να προχωρήσει με την υπόθεση ότι το μητρώο δεν υλοποιεί το V2 του API.

Αυτά είναι τα βήματα του αλγορίθμου που φέρνουν μια εικόνα στο τοπικό σύστημα αρχείων, την επεκτείνουν και την προωθούν σε ένα μητρώο χρησιμοποιώντας το Docker Registry API.

1. GET το διακριτικό ελέγχου ταυτότητας για την επικοινωνία με το μητρώο
2. GET το αρχείο manifest της βασικής εικόνας
3. Επέκταση της εικόνα χρησιμοποιώντας το Python SDK μας
4. POST του νέου συμπιεσμένου tar σε ένα μητρώο της επιλογής μας
5. PUT το ενημερωμένο αρχείο ρυθμίσεων
6. PUT το ενημερωμένο αρχείο manifest
7. Επιβεβαίωση ότι έχουμε δημιουργήσει επιτυχώς μια νέα εικόνα



Σχήμα 4.3: *Επικοινωνία με το διακομιστή Docker Registry API για την επέκταση μιας εικόνας*

4.3 Τα Μυστικά της Δημιουργίας Δομικών Στοιχείων

Ο απώτερος στόχος της διατριβής είναι η αφαίρεση της μετάβασης από μια συνάρτηση Python σε έναν ορισμό KFP δομικού στοιχείου. Το όνειρό μας είναι να επιτρέψουμε στους χρήστες να ταυτίσουν μια συνάρτηση Python με ένα KFP component. Αυτός ακριβώς είναι ο σκοπός της create_component().

4.3.1 Save Component

```
def save_component(save_path: str = None,
                   **build_kwargs):
```

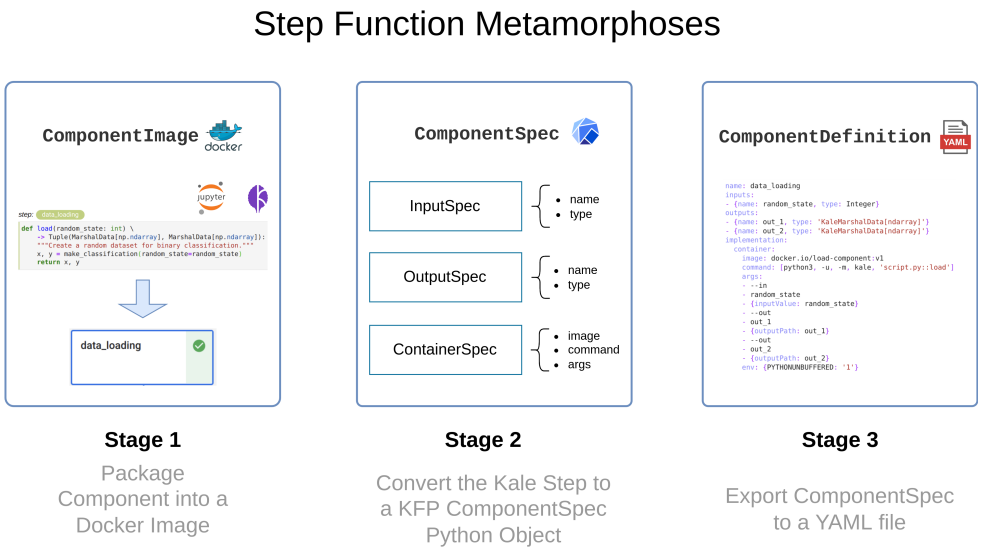
Τα ορίσματα της συνάρτησης αναλυτικά είναι τα εξής:

- *save_path*: Διαδρομή προς ένα αρχείο όπου θα αποθηκευτεί ο ορισμός του στοιχείου. Εάν είναι κενό, το αρχείο αποθηκεύεται στο CWD/<όνομα στοιχείου>.kale.yaml
- ***build_kwargs*: Ορίσματα που σχετίζονται με προηγμένες ρυθμίσεις της εικόνας κατασκευής (π.χ. περιεχόμενα, όνομα, μητρώο)

Υπάρχουν 3 στάδια για τη μετάβαση από μια συνάρτηση σε έναν ορισμό δομικού στοιχείου που βρίσκεται στο τοπικό σύστημα αρχείων.

1. Το πρώτο στάδιο με ύψιστη σημασία είναι η συσκευασία του κώδικα του component σε μια εικόνα Docker. Η βιβλιοθήκη που υλοποιήσαμε για την κατασκευή εικόνων θα είναι υπεύθυνη για αυτό.
2. Το δεύτερο στάδιο περιλαμβάνει τις μεταμορφώσεις της διακοσμημένης συνάρτησης του βήματος σε ένα αντικείμενο KFP *ComponentSpec*.
3. Τέλος, στο τρίτο στάδιο, το αντικείμενο *ComponentSpec* πρέπει να μετατραπεί σε ένα αρχείο προδιαγραφών yaml που θα βρίσκεται στο τοπικό σύστημα αρχείων καθιστώντας το κοινόχρηστο και μεταβιβάσιμο.

Συνολικά, παρουσιάζουμε την εφαρμογή του τρόπου μας για την παραγωγή επαναχρησιμοποιούμενων και διαμοιραζόμενων στοιχείων KFP:



Σχήμα 4.4: *Από μια step decorated συνάρτηση σε έναν ορισμό του component*

```
def save_component(...):

    # Δημιουργία της εικόνας του component με βάση την εικόνα του περιβάλλοντος εργασίας
    base_docker_image = get_docker_base_image()
    step.docker_image = build_image_component(base_docker_image,
                                              **build_kwargs)

    # Μετατροπή του Step σε ComponentSpec
    component = step.component

    # Αποθήκευση του ComponentSpec σε ένα αρχείο YAML
    with open(output_path) as f:
        f.write(dump_yaml(component.to_dict()))
```

Σχήμα 4.5: *Από μια συνάρτηση της Python σε έναν ορισμό yaml του component*

4.3.2 Load Component

Είναι λογικό ότι η φόρτωση ενός component στον κώδικα μας να ακολουθεί σχεδόν την αντίστροφη διαδικασία με εκείνη της δημιουργίας ενός component. Φυσικά, δεν μπορούμε να αντιστρέψουμε διαδικασία κατασκευής μιας εικόνας Docker για το συστατικό, επομένως είναι αυτονόητο ότι θα παραλείψουμε αυτό το στάδιο. Τα δύο εναπομείναντα στάδια είναι τα εξής:

- Φόρτωση ορισμού του δομικού στοιχείου από ένα αρχείο YAML σε ένα αντικείμενο Python ComponentSpec
- Μετατροπή του ComponentSpec σε ένα Step object που το Kale μπορεί να επε-

ξεργαστεί, να ελέγξει, να καταχωρήσει σε ένα Pipeline και να το εκτελέσει

Σχήμα 4.6: *Η μετατροπή ενός YAML αρχείου σε ένα Kale step*

Για να συμβεί αυτό, εισάγουμε την παρακάτω συνάρτηση:

```
def load_component(component_path: str) -> Step:
```

Η συνάρτηση λαμβάνει μια διαδρομή προς ένα αρχείο προδιαγραφών YAML που περιέχει τον ορισμό ενός έγκυρου KFP component και δημιουργεί το αντίστοιχο στιγμιότυπο της κλάσης Step που μπορεί να ενσωματωθεί και να χρησιμοποιηθεί απευθείας σε μια συνάρτηση pipeline.

Συγκεκριμένα, ξεκινώντας από ένα αρχείο YAML κάνει τα εξής:

1. Φορτώνει το yaml σε ένα αντικείμενο dictionary της Python
2. Μετατρέπει το dictionary σε ComponentSpec αντικείμενο του KFP
3. Μετατρέπει το αντικείμενο ComponentSpec σε ένα καινούργιο στιγμιότυπο της κλάσης ExternalStep.

ΣΗΜΕΙΩΣΗ: Διακρίνουμε ένα Kale component, δηλαδή ένα δομικό στοιχείο που εκτελείται υπό την αιγίδα του Kale, από ένα απλό δομικό στοιχείο KFP. Συγκεκριμένα, στην πρώτη περίπτωση επεκτείνουμε το entrypoint του component με ένα marshal path, δηλαδή, μια διαδρομή που χρησιμοποιείται για την μεταφορά δεδομένων μεταξύ των βημάτων του pipeline. Αυτο συμβαίνει προκειμένου να είναι επιτυχής ο μηχανισμός marshalling, που περιβάλλει το βήμα κατά τη διάρκεια του χρόνου εκτέλεσης. Από την άλλη πλευρά, στην περίπτωση ενός απλού component KFP δεν παρεμβαίνουμε στο σημείο εισόδου του component, δεδομένου ότι δεν θα εκτελεστεί από το Kale.

Παράδειγμα:

```
>>> from kale.common import componentutils
>>> @step(name="print_num")
>>> def print_num(a: int):
>>>     print(a)
>>>
>>> add = componentutils.load_component(
>>>     component_path="add.component.yaml"
>>> )
>>> add.docker_image
'docker.io/my-registry/my-component:v1@<digest>'
```

Τα ορίσματα της συνάρτησης είναι:

- *component_path*: Διαδρομή προς το αρχείο YAML ορισμού του δομικού στοιχείου.

Η συνάρτηση τελικά επιστρέφει ένα σιγμιότυπο του αντικειμένου ExternalStep που μπορεί να χρησιμοποιηθεί απευθείας σε μια συνάρτηση pipeline.

5 Επίλογος

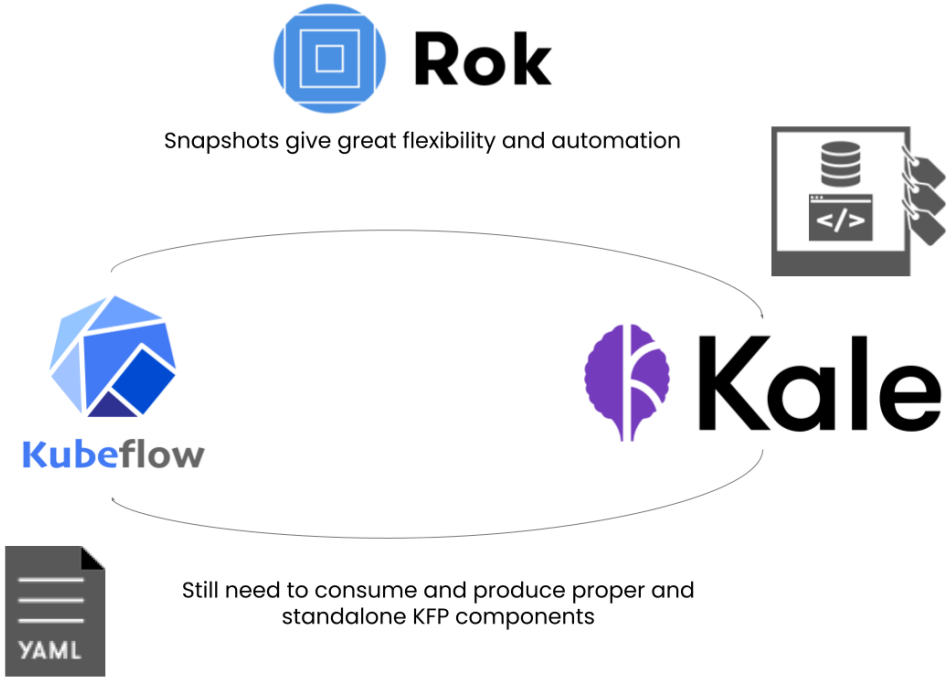
Σε αυτό το τελευταίο κεφάλαιο, παρουσιάζουμε μια σύντομη σύνοψη της εργασίας μας που αξιολογεί ορισμένα κύρια σημεία της διαδικασίας σχεδιασμού. Στη συνέχεια, καταλήγουμε αναφέροντας μερικές πιθανές επεκτάσεις και βελτιώσεις που θα μπορούσαν να αναπτυχθούν στο μέλλον.

5.1 Συμπερασματικά Σχόλια

Ο στόχος μας ήταν να μελετήσουμε και να πειραματιστούμε με τις εικόνες περιεκτών και τις ιδιότητές τους και να διερευνήσουμε τον τρόπο αξιοποίησής τους για τη διαφανή αναπαραγωγή ενός περιβάλλοντος στο Kubeflow. Αποτελεί σταθερή πεποίθησή μας ότι στον σύγχρονο κόσμο οι επιχειρήσεις μηχανικής μάθησης θα ευδοκιμήσουν. Η ανάπτυξη, η συσκευασία και, φυσικά, το deployment μπορούν αναμφισβήτητα να επωφεληθούν από τα πλεονεκτήματα των containers και την ευκολία από τις δυνατότητες που παρέχουν τα εργαλεία ενορχήστρωσης, όπως ο Κυβερνήτης. Παρόλα αυτά, υπάρχει ένα μεγάλο κενό στην αυτοματοποίηση και την αναπαραγωγιμότητα που δεν επιτρέπει να αναδειχθούν οι δυνατότητες των MLOps.

Γι' αυτό το λόγο, αναπτύξαμε μια βιβλιοθήκη που είναι υπεύθυνη για τη δημιουργία εικόνων. Στη συνέχεια, με τον μηχανισμό αυτό καταφέρνουμε με απρόσκοπτο τρόπο να πακετάρουμε ολόκληρο το περιβάλλον του χρήστη σε ένα docker image, εξαιρετικά γρήγορα μέσα σε ένα μη προνομιούχο περιβάλλον. Τέλος, συνδυάζουμε τα παραπάνω με την παραγωγή στοιχείων του Kubeflow Pipeline. Στο τέλος, παρέχουμε μια απλή

εντολή για την πλήρη μεταμόρφωση ενός βήματος της ροής εργασίας μηχανικής μάθησης σε ένα αυτόνομο, επαναχρησιμοποιούμενο και αναπαραγώγιμο συστατικό του KFP. Επομένως, μπορούμε τώρα να συμπεράνουμε ότι ο στόχος μας επιτεύχθηκε. Παρά ταύτα, υπήρχε ένας ακόμη υποκείμενος στόχος: να συνεργαστούμε αποτελεσματικά με προγραμματιστές της Arrikto και του Kubeflow από όλο τον κόσμο, καθώς και να συμβάλουμε σε ένα έργο ανοικτού κώδικα τέτοιας κλίμακας. Όσον αφορά τον δεύτερο στόχο, τη συνεργασία, ήταν μια εμπειρία που περιελάμβανε εποικοδομητικό άγχος. Παρ' όλα αυτά, ήταν υπέροχη.



Σχήμα 5.1: *Time to close the loop*

5.2 Μελλοντικό Έργο

Σχεδόν όπως αναμενόταν, ο μηχανισμός μας για τη δημιουργία εικόνων μαζί με τον τρόπο παραγωγής αυτόνομων δομικών στοιχείων του KFP έχει χώρο για περισσότερη ανάπτυξη.

- **Μηχανισμός κατασκευής εικόνας**

  Προς το παρόν είμαστε αναγκασμένοι να φέρουμε τα περιεχόμενα της βασικής

εικόνας στο τοπικό σύστημα αρχείων του χρήστη. Αυτό σημαίνει ότι σπαταλάμε το χώρο του χρήστη έστω και προσωρινά. Στην ίδια κατεύθυνση, η κατασκευή του νέου επιπέδου απαιτεί δύο λειτουργίες: 1) Δημιουργία αρχείου tar και 2) Συμπίεση του αρχείου tar. Η πρώτη αντιγράφει τα περιεχόμενα των αρχείων που θέλουμε να προσθέσουμε στο αρχείο μας και τα επικολλά σε ένα ξεχωριστό αρχείο που θα αποτελεί το αρχείο tar μας. Προφανώς, αυτό σημαίνει ότι έχουμε ένα αντίγραφο των ίδιων αρχείων στο σύστημα αρχείων μας. Το ίδιο συμβαίνει και με τη συμπίεση του αρχείου. Κατά τη διάρκεια της λειτουργίας συμπίεσης και για λόγους ασφαλείας (τουλάχιστον μέχρι να τελειώσει) υπάρχουν αντίγραφα των ίδιων αρχείων στο σύστημα αρχείων. Ως εκ τούτου, στο μέλλον και με βάση τη συμβολή των πελατών θα εξετάσουμε το ενδεχόμενο να διαθέσουμε έναν εξωτερικό τόμο με τον απαιτούμενο χώρο για να φέρουμε τη βασική εικόνα και να εκτελέσουμε όλη τη διαδικασία του μηχανισμού δημιουργίας εικόνας. Επιπλέον, θέλουμε να απεμπολίσουμε πλήρως το εργαλείο *Skopeo* και να αναπτύξουμε το δικό μας μηχανισμό για την επικοινωνία με τοπίκα ή απομακρυσμένα μητρώα. Τέλος, σκοπός μας είναι να μπορούμε να βλέπουμε αυτόματα τις αλλαγές που γίνονται στο σύστημα αρχείων και να τις συμπεριλαμβάνουμε σε μια βασική εικόνα ως ένα νέο επίπεδο.

- **User Interface & Notebooks**

  Προς το παρόν οι χρήστες μπορούν να παράγουν διαμοιραζόμενα και αναπαραγώγιμα δομικά στοιχεία KFP και να εκτελούν pipelines με τον μηχανισμό build image μόνο με χρήση του Kale SDK. Ένα από τα επόμενα βήματά μας είναι η ενσωμάτωση ολόκληρου του μηχανισμού στα Jupyter Notebooks. Αυτό θα περιλαμβάνει αλλαγές στην επέκταση Kale's Lab, ώστε να μπορούν οι χρήστες να διαμορφώνουν τις προηγμένες ρυθμίσεις του μηχανισμού κατασκευής εικόνων και τα περιεχόμενα της τελικής τους εικόνας.

- **Component Marketplace**
  Όπως αναφέραμε πολλές φορές σε αυτή τη διατριβή, προσπαθούμε να δημιουργήσουμε μια κοινότητα ML όπου οι επιστήμονες δεδομένων και οι ομάδες ML θα μπορούν να αλληλεπιδρούν μεταξύ τους και να μοιράζονται τις ιδέες τους με το πάτημα ενός κουμπιού. Αναμφισβήτητα, δεν είναι τόσο εύκολο να μοιραστείς

ένα αρχείο yaml (ορισμός συστατικού), επομένως η πρόθεσή μας είναι να δημιουργήσουμε ένα αποθετήριο που θα περιλαμβάνει δομικά στοιχεία από όλους τους Επιστήμονες Δεδομένων και θα μπορούν να χρησιμοποιηθούν γρήγορα και εύκολα.

1 Introduction

1.1 Motivation

Containers + Machine Learning = MLOps

In the last couple of years containerization, that is, packaging, protection and shipment of an application, has emerged extremely as a technology. The concept ofcontainerization and process isolation is actually decades old, but the emergence in 2013 of the open source Docker Engine-an industry standard for containers with simple developer tools and a universal packaging approach—accelerated the adoption of this technology. Today organizations are using containerization increasingly to create new applications, and to modernize existing applications for the cloud. Perhaps most important, containerization allows applications to be "written once and run anywhere." The rise of containers and microservices technology resulted in applications that now comprise hundreds or sometimes thousands of containers. Managing those loads of containers across multiple environments can be really challenging. Kubernetes is an open source container orchestration tool that satisfies the need of abstracting the control, supervision and administration of numerous containers.

Containers along with Kubernetes helped tremendously the DevOps culture to evolve and remove the barriers between two traditionally siloed teams, development and operations. DevOps is a set of practices in the traditional software development world that enables faster, more reliable software development into production. DevOps relies on automation, tools, and workflows to abstract accidental complexity away and allow developers to focus on more critical issues. DevOps achieves benefits such as increasing

deployment speed, shortening development cycles, and increasing the dependability of releases by introducing two concepts during development of the software system:

- Continuous Integration (CI)
- Continuous Deliver (CD)

What else has emerged lately? Enter Machine Learning. Nowadays, machine learning (ML) is a core piece of businesses' daily life, as it proves to be valuable for solving various real-world problems. Most industries working with large amounts of data have recognized the value of machine learning technology. By gleaning insights from this data – often in real time – organizations are able to work more efficiently or gain an advantage over competitors. Machine learning has far too many applications and exists in our everyday life more than we might believe it does. Email categorization, Speech Recognition, Self-driving cars, Google translate as well as all kinds of online advertisements are just some of them. Machine learning can be a very resource-intensive activity, whether it be the process of building a machine learning model or deploying and executing it in a production environment.

Can we leverage containers to facilitate the Machine Learning Workflow and improve models performance? Absolutely! Kubernetes can meet many of the computational challenges by orchestrating this workflow through containers. A common practice today, as it happens with most of the applications, is to package and deploy ML jobs as containers managed by the Kubernetes orchestrator. But it is definitely an onerous task for the data scientist to move from their local computer where they develop their model to a Kubernetes Cluster for training and serving.

Our motive and what we strive for in this thesis is to make the Data Scientists everyday life easier by bridging the gap between development and production. We want to enable data scientists to focus on building and deploying their models. How cool would it be to serve models with a right mouse click?
To build an ML system that will continuously operate in production is a significantly heavy job for a data scientist. Many teams have data scientists and ML researchers who can build state-of-the-art models, but their process for building and deploying ML models on Kubernetes is entirely manual. Productionizing machine learning is onerous and abstruse. The machine learning lifecycle consists of many complex components

such as data ingest, data prep, model training, model tuning, model deployment, model monitoring, explainability, and much more. It also requires collaboration and hand-offs across teams, from Data Engineering to Data Science to ML Engineering. What if we applied the DevOps art - appropriately adjusted - in Machine Learning applications?

Machine Learning Operations

Indeed! Similarly to DevOps, MLOps came to address these issues by encompassing the experimentation, iteration, and continuous improvement of the machine learning lifecycle. To effectively achieve machine learning model lifecycle management, MLOps fosters communication and collaboration between operations professionals and data scientists. MLOps stands for Machine Learning Operations. MLOps is a core function of Machine Learning engineering, focused on streamlining the process of taking machine learning models to production, and then maintaining and monitoring them.

1.2 Problem Statement

MLOps is the future! Yet, it is not in a state where data scientists will not interfere with unrelated to their model development work. In fact, many projects and products attempt to tackle the problem of deploying machine learning workflows on Kubernetes. Despite their goal to make their usage simple and easy, they are still not appealing to the data scientists. As a consequence, even simple operations require a lot of irrelevant studying by the user.

Data science is inherently a pipeline workflow, from data preparation, to training and deployment, thus every ML project is organized in these logical steps. Without enforcing a strict pipeline structure to Data Science projects, it is often too easy to create messy code, with complicated data and code dependencies and hard to reproduce results.

A popular machine learning toolkit dedicated to making deployment of ML workflows on Kubernetes simple, portable and scalable is Kubeflow. Kubeflow Pipelines (KFP) is the Kubeflow component responsible for the end-to-end orchestration of ML pipelines. A pipeline is a description of a machine learning workflow, including all of the components in the workflow and how these components relate to each other in the form of a graph.

Kubeflow along with their Pipelines mechanism is an excellent tool to drive data scien-

tists to adopt a disciplined ("pipelined") mind set when developing ML code and scaling it up in the Cloud. The Kubeflow Pipelines' Python SDK is a great tool to automate the creation of these pipelines, especially when dealing with complex workflows and production environments. Still, when presenting this technology to ML researchers or Data Scientists that don't have strong software engineering expertise, KFP can be perceived as too complex and hard to use.

Data Science is often a matter of prototyping new ideas, exploring new data and models, experimenting fast and iteratively. In these scenarios one would prefer to just run some rough code and analyze the results rather than setting up complex workflows with a specific SDK. Kubeflow has become the preferred MLOps automation platform because it offers a production workflow that is automated, portable, reproducible, and secure. However, there are currently two major gaps between the potential for MLOps and the reality of deploying Kubeflow at scale in production:

1. *Automation Gap*: disjointed workflows. Data scientists are not DevOps or infrastructure experts, but today they are expected to create Docker containers, publish to repos, write DSL and DAG definitions for pipelines, write YAML files, etc.

2. *Reproducibility Gap*: data versioning hell. ML is unique: both data and code constantly change, making debugging, compliance, and collaboration nearly impossible.

During this diploma thesis these will be the two key points that we will aim to solve. Kubeflow Pipelines consist of a number of steps, each one of which runs as an independent process with regards to the others. Each step performs a specific task of a machine learning workflow, e.g: preprocessing/cleaning a dataset, training a model, producing predictions on a test subset etc.. A pipeline component is a self-contained set of user code, packaged as a Docker image, that performs one step in the pipeline.

Kubeflow hasn't managed yet to abstract concepts like containers, Kubernetes, yaml files and Argo workflows from data scientists. Consequently, users are frequently asked to build Docker images for their Kubeflow Pipeline. The only provided solution by Kubeflow is to use by default a Python base image and extend the user's code with the bare minimum argument parser in order to be able to read the inputs and outputs. KFP suggests users to build their own Docker image if they have complex dependencies. This

is reasonable, because the default KFP approach and behavior comes with serious implications and restrictions. For example:

1. It should not use any code declared outside of the function definition.

2. Import statements must be added inside the function and the same thing holds for helper functions.

1.3 Proposed Solution

Kale alongside Rok made a first tremendous step on addressing these difficulties by providing a tool to simplify the deployment process of a Jupyter Notebook into Kubeflow Pipelines workflows. Translating Jupyter Notebook directly into a KFP pipeline ensures that all the processing building blocks are well organized and independent from each other, while also leveraging on the experiment tracking and workflows organization provided out-of-the-box by Kubeflow. How does Kale build the necessary Docker images for the KFP components? It doesn't. Rok takes snapshots of the user's Kubeflow environment (code + data) and attaches it to the Kubernetes Pods where the Pipeline will run. Unfortunately, this approach comes with some drawbacks:

1. Kale can't produce shareable and standalone KFP components, since they have to be packaged along with their dependencies into Docker images.

2. Pipelines compile and run only under the aegis of Rok. But Rok is an enterprise and heavy software serving needs for very demanding users.

3. Rok can't be easily attached in very restricted environments with limited resources, which means pipeline steps can't run anywhere.

Eventually, Kale with Rok solves some of the major Kubeflow's problems, still there are some gaps that need to be filled. In this diploma thesis our ultimate goal will be to address these shortcomings. Our sight will always point towards automation and reproducibility. In other words, we will aim to make it easy for the Data and Machine Learning Scientists to get from their development environment to production with a

few clicks and without requiring deep knowledge of neither Kubernetes nor containers. In addition, we will strive for enabling them to effortlessly experiment with various ML workflow combinations by reusing already packed pipeline components created either by themselves or other engineers. Our hope is to save them valuable time towards finding the best model for their needs, increasing collaboration and compliance.

For all this to happen, we will look for ways to package Kubeflow PIpeline components and their complex dependencies into a Docker image in a seamless way. Towards the road of success, we'll have to explore container images technology at its core. Gaining a deep understanding of how Docker and OCI produce a fully enclosed space that can be used to contain, store, and transport applications will allow us to develop our way of manipulating and in turn to extend Docker and OCI images. After that, we will be able to introduce and establish a way to build images on the fly in unprivileged environments, that is, everything happens in user space. This is crucial for safety and security reasons, since many ask to run rootless containers. It adds one more safety layer that keeps customers happy when it comes to Kubernetes clusters. Eventually, we cannot run Docker inside an inherently unpriviliged environment like Kubeflow. This is why we will develop our own way to build images.

With a build image mechanism in our hands, we will expand Kale and its capabilities to support creating reusable, shareable and standalone KFP pipeline components. In particular, at first we will integrate the new mechanism into Kale with an API composed of two high-level functionalities: save and load components.

All in all, we will manage to package the user's working environment (Kubeflow's Notebook Server) along with their code and data using our build image mechanism. Then, we will extend Kale to produce proper standalone components that can be run anywhere from anyone in a Kubeflow environment, enhancing collaboration.

1.4 Thesis Structure

The content of the thesis is structured as follows:

- Chapter 2: a brief overview of some of the core concepts and systems that our work is founded upon.

- Chapter 3: an analysis of the architecture of our solution and the design decisions from a higher-level perspective.
- Chapter 4: a brief demonstration of some of the focal points of our development process, including mainly details about the implementation of certain design choices that could have been implemented in multiple different ways.
- Chapter 5: concluding remarks and future improvements and extensions to our proposed solution

2 Background

In this chapter we provide some elementary information regarding various aspects of the background knowledge required to understand the design and the implementation of the proposed way to build container images and produce standalone and reusable components in this work. We begin by deconstructing containers and their implementation in the Linux kernel into their principal constituents: Linux namespaces and control groups. After briefly describing the functionality of these components, we discuss about container images and the OCI image specification. Next, we demonstrate how all these concepts and mechanisms can be used to cooperatively formulate the foundation for a container runtime engine: we discuss about some aspects of Docker and its take on the design and implementation of such a system. We move on to concisely analyze some facets of Kubernetes and its rationale: its perspective on containerization, some software architecture and engineering choices, several abstractions that it creates, and more. This chapter concludes with a quick presentation of Kubeflow, Kale and Rok and their value in Machine Learning Operations.

At this point, we have to remark that this chapter's content is by no means a complete analysis or explanation of all concepts and systems involved in the thesis. It rather is a high-level overview of the basic functionality and architecture of some of them. Therefore, this chapter is safe to skip for the readers who are already familiar with the topics mentioned in the above paragraph, whereas it only provides elementary, high-level information about them to the readers who are not already familiar with them, since the details are deliberately missing. Additional background concepts and knowledge that may be required for fully understanding the reasoning of the arguments in the thesis

are sometimes presented in the following chapters as well. Such concepts may constitute design or implementation decisions, thus fitting better in chapter 3 or chapter 4.

2.1 OS-level Virtualization and Containers

2.1.1 Welcome to Containers World

What is a container?

According to Wikipedia, a container is any receptacle or enclosure for holding a product used in storage, packaging, and transportation, including shipping. Things kept inside of a container are protected on several sides by being inside of its structure [1]. The term is most frequently applied to devices made from materials that are durable and are often partly or completely rigid. We can consider a container as a basic tool that creates a partially or fully enclosed space that can be used to contain, store, and transport objects or materials.

Why use a container?

The product characteristics that create utility for a container go beyond just providing shock and moisture protection for the contents. A well-designed container will also exhibit ease of use, that is, it is easy for the worker to open or close, to insert or extract the contents, and to handle the container in shipment. In addition, a good container will have convenient and legible labeling locations, a shape that is conducive to efficient stacking and storing, and easy recycling at the end of its useful life [1]. Imagine trying to ship and stack erratic shapes and sizes of boxes. That is the picture that should immediately come to your mind when standards and specifications are missing.

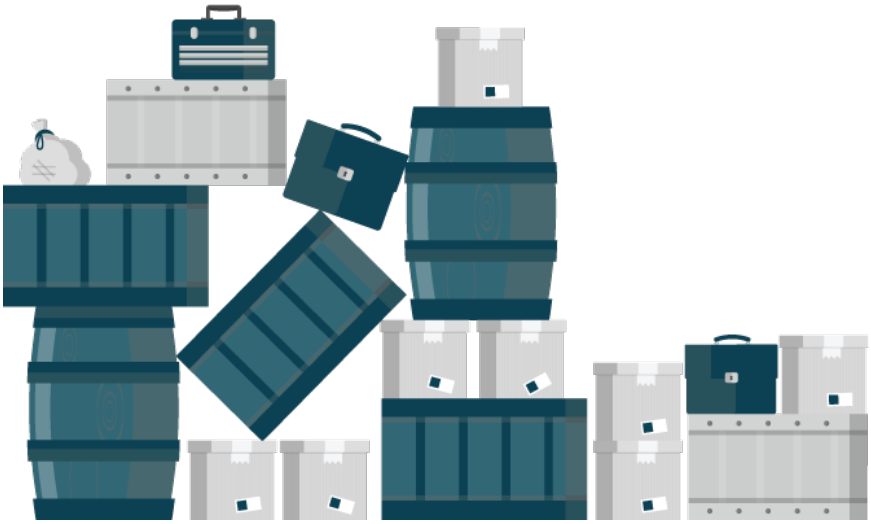


Figure 2.1: *Boxes manually loaded on trains and ships in 1921*

What is a Linux container?

In the computer engineer's world -from time to time- the terms of a new technology reflect the technology itself and describe its functionality. That is the case with Linux containers. The definition of a physical container applies precisely to the Linux Containers World. A Linux container is a set of 1 or more processes that are isolated from the rest of the system [2] (protected on several sides). All the files necessary to run them are provided from a distinct image (package), meaning Linux containers are portable and consistent as they move(transportation) from development, to testing, and finally to production.

Like a normal Linux program, containers really have two states - rest and running. When at rest, a container is a file (or set of files) that is saved on disk. This is referred to as a Container Image. When a container starts, the Container Engine unpacks the required files and metadata, then hands them off to the Linux kernel. Once running, Containers are simply, plainly a Linux process. The process for starting containers, as well as the image format on disk, are defined and governed by standards.

Why use Linux containers?

Linux container characteristics are almost identical to physical ones. The same holds for their benefits. Imagine you're developing an application. You do your work on a laptop and your environment has a specific configuration. Other developers may have slightly different configurations and definitely a not identical workspace. The application you're developing relies on that configuration and is dependent on specific libraries,

dependencies, and files. You want to emulate those environments as much as possible locally, but without all the overhead of recreating environments in order to make your app work across these environments. How do you pass quality assurance, and get your app deployed without massive headaches, rewriting, and break-fixing? Enter containers.

The container that holds your application has the necessary libraries, dependencies, and files so you can move it through production without nasty side effects. In fact, the contents of a container image can be thought of as an installation of a Linux distribution because it comes complete with RPM packages, configuration files, etc.

That's a common example that unveils the significance of being able to fit a box in the most grueling spaces, which also silently reveals the supreme space efficiency containers have. But Linux containers can be applied to many different problems where portability, configurability, and isolation is needed. Think about reproducibility, reusability and shipment. You open the box, add some content, and close the box. None of the most peculiar, odd and demanding environments can stop you from versioning, extending/ shrinking and shipping your app. The point of Linux containers is to develop faster and meet business needs as they arise. In some cases, containers are essential because they're the only way to provide the scalability an application needs. No matter the infrastructure—on-premise, in the cloud, or a hybrid of the two—containers meet the demand [2].

Containerisation, an Operating system (OS)-level virtualization technology, simplifies the deployment and management of applications by providing a flexible, low-overhead virtualization solution. In recent years, containers have been deployed as a replacement for VirtualMachine (VM)-based virtualization (are they?). Containers offer better performance and flexibility at the expense of isolation and security when compared to traditional hypervisor virtualization.

Figure 2.2: *Perfectly stacked containers showing their ease of use and benefits in shipment.*

2.1.2 Virtualization

The traditional solution to enable isolation and resource sharing uses hypervisor-based virtualization. The hypervisor abstracts complete computer systems; these abstractions are called Virtual Machines. Each VM has its operating system that executes completely isolated from other VMs. Consequently, different operating systems can be virtualized on a single host. To enable the creation of VMs, compute, network and storage components are abstracted by the hypervisor. Hypervisors offer better isolation between resources than other technologies like containers because applications in separate VMs run on a different Guest OS. The increased isolation offered by VMs affects performance significantly; First, booting and rebooting a VM can take tens of seconds as a complete OS needs to be started. Second, because a full machine is imitated, instructions are emulated or just in time (JIT) compiled, which can introduce a performance hit.

2.2 Containers ~~vs~~ and VMs

Are containers a substitute for Virtual Machines, or maybe a subset of VMs? None of them is actually true. Do they serve a similar purpose? Not exactly. What is the rela-

tionship between the two of them? Answer: Engineering. The real magic, captivation and fascination with engineering is that nothing is treated as binary. Engineers build technology in nature's image, well arguably this approach has its drawbacks, since nature and humans are not infallible. Technology adapts and evolves to each era's needs, circumstances, and environment to leverage the most out of each situation. Therefore, you can think of containers and VMs as being complementary to one another. Virtualization lets multiple operating systems run simultaneously on a single hardware system. Containers share the same operating system kernel and isolate the application processes from the rest of the system. They function in a different stack of the computer's pyramid and if used together provide a great deal of flexibility in deploying and managing apps.

As inexplicable and mysterious as it may seem they have completely different implementation techniques, towards a similar end goal. And now is the right time to dive into Linux containers and realize the complete disparity between them and virtualization.

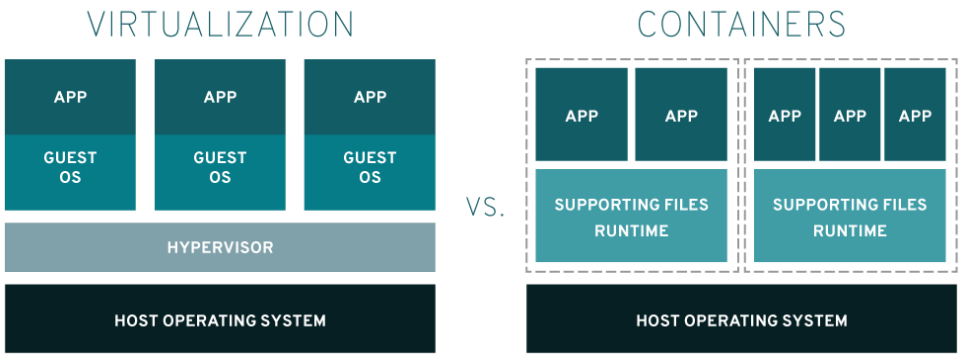


Figure 2.3: *Containers vs Virtual Machines*

2.3 Linux Kernel and Containers

Containers are just normal Linux Processes with additional configuration applied. Containers aren't really a thing. They're a mishmash of Linux kernel-isms like namespaces and cgroups. The current Linux kernel community philosophy is that the Linux kernel should provide a bunch of different technologies, ranging from experimental to very mature, enabling users to mix these technologies together in creative, new ways. And, that's exactly what a container engine (Docker, Podman, CRI-O, etc) does - it leverages

kernel technologies to create, what we humans call, containers.

Containers don't exist in the Linux world. No one ever defined what a container is, or created a structure called container in the Linux kernel. Container is just a combination of different technologies provided by the Linux kernel. The concept of a container is a user construct, not a kernel construct. We will dig deeper into the technical underpinnings, but for now, understand that containerized processes are regular Linux processes which are isolated using kernel technologies like namespaces, selinux and cgroups. This is sometimes described as ”sand boxing” or ”isolation” or an ”illusion” of virtualization. All processes live side by side, whether they are regular Linux processes, long lived daemons, batch jobs, interactive commands which you run manually, or containerized processes. All of these processes make requests to the Linux kernel for protected resources like memory, RAM, TCP sockets, etc.

“Containers are fancy Linux processes” is a phrase we can compromise with and describes adequately what a container is. And that fancy refers to the sandboxing and isolation that we try to achieve and get closer to the word/world of virtualization without though by any means virtualize anything. Before starting investigating the fancy part of containers, let's prove to ourselves that containers are really just a linux process.

2.3.1 Containers are Linux Processes

From here on, we enter the world of linux containers. Docker will help us run containers.

```
# docker run -d --name=redis redis:alpine
```

The flag *-d* stands for detached mode. This means that we're not getting inside the container. We let it run in the background. Our purpose is to search for the container (or else the process) in the tree of the linux processes of our host system. To do so we'll use the *pstree* command.

```
# pstree -c -p -A $(pgrep containerd-shim)
containerd-shim(49415)-+-redis-server(49435)-+-{redis-server}(49471)
…
```

We realize that redis-server exists in the processes' tree and not only that but we can view all its information under */proc/«pid»/* file. For example, here is the memory mapping of the process:

```
# cat /proc/49435/maps/
5557e0198000-5557e0202000 r--p  fd:01 20975945     /usr/local/bin/redis-server
5557e0202000-5557e0391000 r-xp  fd:01 20975945     /usr/local/bin/redis-server
5557e0391000-5557e0412000 r--p  fd:01 20975945     /usr/local/bin/redis-server
...
```

2.3.2 Containers are not VMs

What would happen if we run acommand inside a container? Will it appear in the process tree of the host system? If yes, then what will be the difference between the process ID inside the container and the host system, if any.

```
# docker run -it --name=ubuntu-test ubuntu
root@7c91c24b3001:/# ls
bin  boot  dev  etc  home  lib  lib32  lib64  libx32  media
mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
```

The flag *-it* is used to get into the container in an interactive mode. Notice that immediately we changed the root directory and moved to a different one pretending to be root ('/') with various popular Ubuntu directories (i.e. /bin, /tmp, /boot). We'll come back to this. watch<command> runs the same command every 2 seconds, staying active constantly. As a command 'ps aux' will also serve the purpose of discovering the processes running inside our container.

```
root@7c91c24b3001:/# watch ps aux
    PID TTY      STAT   TIME COMMAND
    1 pts/0     Ss     18:25 bash
    10 pts/0    S+     18:32 watch ps aux
    11 pts/0    S+     18:32 watch ps aux
    12 pts/0    R+     18:32 ps aux
```

The container can't see the processes running in the host system, what about the other way around?

```
$ ps aux | grep 'watch ps aux' | head -1
    USER         PID TTY      STAT START   TIME COMMAND
    root       53419 pts/0    S+   21:36   0:00 watch ps aux
```

We notice that *watch ps aux* is a process that even though on paper it does not run in the host system - instead it runs inside a container - the host system can clearly see it as its own process. In fact, the same process has a different PID inside the container(10) and

in the host system(53419). Or, in other words, this process has **two** PIDs. In the world of virtualization, this situation would never happen since machines are completely isolated from the host.

2.3.3 Chroot

As it happens in Virtual Machines, a new, independent and isolated environment needs a different root directory that comprises all the installed libraries and binaries that will be used instead of the ones living in the host system. While a user executes a command like 'ls', linux goes to the root directory of the host system and finds the *ls* binary file under */usr/bin* directory. How does a simple linux process (container) manage setting a new root directory? As we already mentioned, containers are fancy processes exploiting many of Linux Kernel's capabilities. One of them is *chroot*.

- chroot [30]
  chroot - run command or interactive shell with special root directory
  chroot() changes the root directory of the calling process to that specified in path. This directory will be used for pathnames beginning with /. The root directory is inherited by all children of the calling process [31].

It is sometimes referred to as "jail", meaning that we lose our initial root and we cannot come back to it using an absolute path. However, relative paths can still refer to files outside of the new root. At the end of the day a user can easily escape from this kind of jail. Moreover, there is neither process nor network isolation at all. What is needed to run a new chroot environment? The '/bin/bash' binary and the relevant dependent libraries. We can even *kill programs running outside of the jail*, what a *metaphor* [32]!

In the container that we ran previously using Docker, we saw that we cannot have access to processes running outside the container. What is the missing piece of the puzzle? This is where Linux namespaces join the party.

2.3.4 Namespaces

Namespaces are a Linux kernel feature which were introduced back in 2002 with Linux 2.4.19. A namespace wraps a global system resource in an abstraction that makes it

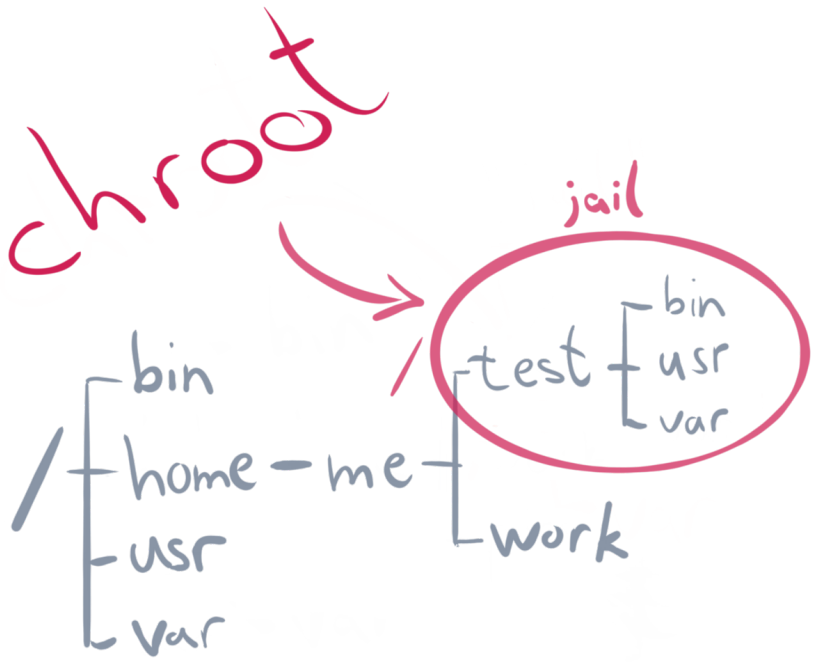


Figure 2.4: *chroot visualization [32]*

appear to the processes within the namespace that they have their own isolated instance of the global resource. Changes to the global resource are visible to other processes that are members of the namespace, but are invisible to other processes. One use of namespaces is to implement containers [4]. One of the fundamental parts of a container is namespaces. The concept of namespaces is to limit what processes can see and what parts of the system can access, such as other network interfaces or processes. The kernel's namespace abstraction allows different groups of processes to have different views of the system. Conceptualize a namespace as a box. Inside this box are these system resources, which ones exactly depend on the box's (namespace) type.

For instance, the Network namespace encapsulates system resources related to networking such as network interfaces (e.g wlan0, eth0), route tables. A process in a namespace has a completely different network configuration than another process living in a different namespace. Highlight this, network namespaces are vital for Kubernetes and clusters where more than one container lives in the same node. This allows multiple containerized web servers on the same host system, with each server bound to port 80 in its (per-container) network namespace. Imagine being forced to have completely different Virtual Machines for each network service. This would be catastrophic.

There are currently 7 types of namespaces Cgroup, IPC, Network, Mount, PID, User, UTS. Namespaces aren't some add-on feature or library that you need to apt install, they are provided by the Linux kernel itself and already are a prerequisite to run any process on the system. At any given moment, any process P belongs to exactly one instance of each namespace type - so when it needs to say, update the route table on the system, Linux shows it the copy of the route table of the namespace to which it belongs at that moment. Note though that there is a tree structure relationship between namespace instances, as it happens with processes. The boxes are not completely independent of one another. The init process always goes to the default namespace. The default Kernel's behavior is to set to the same namespace all the cloned processes produced from init. Then, how do we really change a namespace instance of a specific type for a new process? Enter clone, nsenter and unshare.

clone

The clone API function creates a new child process, in a manner similar to fork [33]. Unlike fork, the clone API allows the child process to share parts of its execution context with the calling process, such as the memory space, the table of file descriptors, and the table of signal handlers. It is actually a Linux specific syscall mostly used to implement threads. Have you ever questioned how threads share the heap space? Clone is the answer. And guess what, you can pass different namespace flags to clone to create new namespaces for the child process. For example, A new PID namespace is created by calling clone() with the CLONE_NEWPID flag.

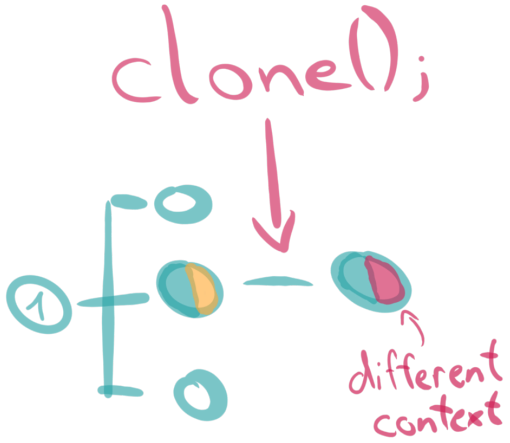


Figure 2.5: *A simple visualization of a process spawning a child process in a different context [32]*

unshare

Now, you should ask yourself how we could change namespace for a particular process without creating a new one. Unshare will knock on our door. *unshare()* allows a process (or thread) to disassociate parts of its execution context that are currently being shared with other processes (or threads). Part of the execution context, such as the mount namespace, is shared implicitly when a new process is created using fork(2) or vfork(2), while other parts, such as virtual memory, may be shared by explicit request when creating a process or thread using clone(2). The main use of unshare() is to allow a process to control its shared execution context without creating a new process [34]. All in all, the role of unshare is to not only allow a smooth divorce with our partner with whom we share some resources, but also finds for us a new one to engage with.

setns

What happens if we regret our decision and want to reconnect with the love of our life? Linux kernel has covered this case too for us. Another feature of Linux Kernel is the function setns() that associates the calling process with the provided namespace file descriptor. The setns() system call allows the calling thread to move into different namespaces [35]. But how does someone select a particular namespace instance? Since we are in a Linux land, the well-known mantra *everything is a file* dominates. And it is actually true, the more we dive deep into the Linux world the more we feel it in our veins. So, the answer is namespaces are nothing more than a file and as a matter of

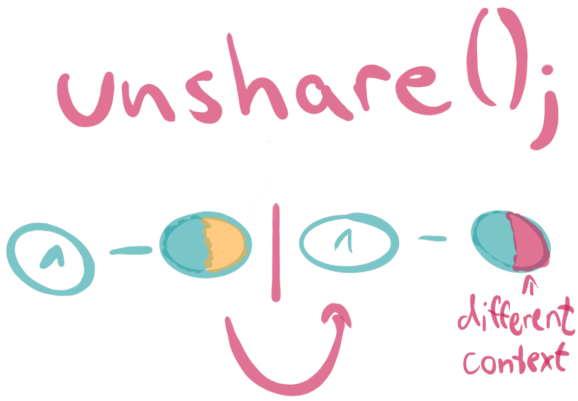


Figure 2.6: *unshare helps us change the context of a process without spawning a new one [32]*

fact they are inodes on the disk. This allows for processes to share the same namespace and makes possible the interaction between them. We could list all the namespaces of a

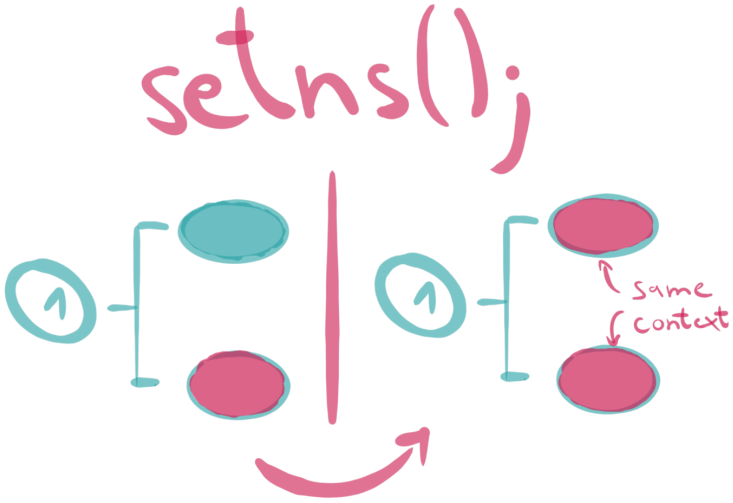


Figure 2.7: *A simple visualization of entering to a context [32]*

process with:

```
$ ls -Gg /proc/self/ns
total 0
lrwxrwxrwx 1 0 Mαï   5 15:25 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 0 Mαï   5 15:25 ipc -> 'ipc:[4026531839]'
lrwxrwxrwx 1 0 Mαï   5 15:25 mnt -> 'mnt:[4026531840]'
```

```
lrwxrwxrwx 1 0 Maï   5 15:25 net -> 'net:[4026532024]'
lrwxrwxrwx 1 0 Maï   5 15:25 pid -> 'pid:[4026531836]'
lrwxrwxrwx 1 0 Maï   5 15:25 pid_for_children -> 'pid:[4026531836]'
lrwxrwxrwx 1 0 Maï   5 15:25 time -> 'time:[4026531834]'
lrwxrwxrwx 1 0 Maï   5 15:25 time_for_children -> 'time:[4026531834]'
lrwxrwxrwx 1 0 Maï   5 15:25 user -> 'user:[4026531837]'
lrwxrwxrwx 1 0 Maï   5 15:25 uts -> 'uts:[4026531838]
```

The number in square brackets represents the inode of the file itself.

```
$ ls -Li /proc/self/ns/pid
4026531836 pid
```

The flag '*-L*' is used to show the information of the file referenced by the symlink and '*-i*' to show the inode number. There is also one more tool/system call included in the linux kernel called *nsenter* which is used to run a program in different namespaces. The *nsenter* command executes the program in the namespace(s) that are specified in the command-line options [36]. This means that we can ingest into a Docker container using this command.

```
# docker run -it ubuntu bash
root@769df66dd1e1:/# ls
bin   dev  home  lib32  libx32  mnt  proc  run   srv  tmp  var
boot  etc  lib   lib64  media   opt  root  sbin  sys  usr
root@769df66dd1e1:/#
```

Then:

1. Find the process of the container
2. Use the PID as nsenter's target to grab the relevant namespaces

```
$ sudo nsenter -t 256932 --all /bin/bash
[sudo] password for pangiann:
root@769df66dd1e1:/# ls
bin   dev  home  lib32  libx32  mnt  proc  run   srv  tmp  var
boot  etc  lib   lib64  media   opt  root  sbin  sys  usr
root@769df66dd1e1:/#
```

With the help of nsenter we achieved ingesting into a Docker container. That is pretty much everything we need to know about namespaces. Interested readers may want to look at the following paragraphs where we focus on specific namespaces like pid, mount and network. Otherwise, you can jump immediately to the next section.

Mount namespace

Mount namespaces were the first namespace type added to Linux, appearing in 2002 in Linux 2.4.19. They isolate the list of mount points seen by the processes in a namespace. Or, to put things another way, each mount namespace has its own list of mount points, meaning that processes in different namespaces see and are able to manipulate different views of the single directory hierarchy [37].

When the system is first booted, there is a single mount namespace, the so-called "initial namespace". New mount namespaces are created by using the CLONE_NEWNS flag with either the clone() system call (to create a new child process in the new namespace) or the unshare() system call (to move the caller into the new namespace). When a new mount namespace is created, it receives a copy of the mount point list replicated from the namespace of the caller of clone() or unshare().

Mount namespaces serve a variety of purposes. For example, they can be used to provide per-user views of the filesystem. Other uses include mounting a /proc filesystem for a new PID namespace without causing side effects for other processes and chroot()-style isolation of a process to a portion of the single directory hierarchy.

With the unshare command and the '-m' flag we can move the running process to a new mount namespace. As we mentioned earlier, the new mount namespace replicates the mount points of the parent namespace.

1. We move to a new mount namespace
2. Then, we create a new directory and mount the tmpfs device
3. We edit the new mount by adding some files
4. Finally, we inspect if anything changed in the host system (and the initial mount namespace)

```
$ sudo unshare -m
# mkdir mount-dir
# mount -t tmpfs tmpfs mount-dir
# touch mount-dir/{0,1,2}
# findmnt
…
└─/home/vagrant/mount-dir             tmpfs      tmpfs     rw,relatime
```

Back to the host system, we notice that mount-dir exists (why? We leave this as an exercise). However, findmnt cannot discover the new mount and our new files are not

there.

```
$ ls mount-dir
[null]
$ findmnt | grep mount-dir
[null]
```

PID namespace

The PID namespace was introduced in Linux 2.6.24 (2008) and isolates the process ID number space. This means that processes which reside in different namespaces can own the same PID. The PID namespaces can be nested following a tree structure like the processes themselves. Thus, if a new process is created it will have a PID for each namespace from its current namespace up to the initial PID namespace. The first process created in a PID namespace gets the number 1 and gains all the same special treatment as the usual init process.

One of the main benefits of PID namespaces is that containers can be migrated between hosts while keeping the same process IDs for the processes inside the container [38]. PID namespaces also allow each container to have its own init (PID 1), the ”ancestor of all processes” that manages various system initialization tasks and reaps orphaned child processes when they terminate.

This was the first thing we noticed before even mentioning anything about namespaces, remember right? If not, go back to the subsectoin 2.3.2. Each process on a Linux system has a */proc/PID* directory that contains pseudo-files describing the process. This scheme translates directly into the PID namespaces model. Within a PID namespace, the */proc/PID* directories show information only about processes within that PID namespace or one of its descendant namespaces. However, in order to make the */proc/PID* directories that correspond to a PID namespace visible, the proc filesystem needs to be mounted from within that PID namespace.

```
$ sudo unshare -fp --mount-proc
root@vagrant:/home/vagrant# ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.4   8960  4152 pts/0    S    18:29   0:00 -bash
root          12  0.0  0.3  10612  3252 pts/0    R+   18:29   0:00 ps aux
```

The *–mount-proc* flag is needed to re-mount the proc filesystem from the new namespace. Otherwise we would not see the PID subtree corresponding with the namespace.

Another option would be to manually mount the proc filesystem via *mount -t proc proc /proc*, but this also overrides the mount from the host where it has to be remounted afterwards [32].

Network namespace

As the name would imply, network namespaces partition the use of the network—devices, addresses, ports, routes, firewall rules, etc.—into separate boxes, essentially virtualizing the network within a single running kernel instance. A process running in a distinct network namespace has its own networking devices, routing tables, firewall rules [38]. During initial creation, a network namespace contains only a loopback interface, and as you would expect the system starts with an initial network namespace within which all processes belong unless specified otherwise.

We can create a new network namespace with ip as command and netns as subcommand. There is no major difference between ip netns add and unshare -n in case you asked. The former chooses by default where it will mount the new network information. For the latter we can decide by ourselves. For the next commands note that we have to be root to run them.

```
# ip netns add pan
# ip netns list
pan
```

Wait! We just mentioned that from the system initialization and afterwards all processes are placed into the default network namespace. Then, why is it not listed? The reason for this is that ip creates what is called a named network namespace, which simply is a network namespace that is identifiable by a unique name. Only named network namespaces are shown via list and the initial network namespace isn't named. As we mentioned before, *ip netns add* mounts the relevant files created for the new network namespaced under a default path which is */var/run/netns*. Now, time to grab a shell in this namespace and explore the new super clean and isolated network.

```
# ip netns exec pan bash
P# ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

Indeed, our new network namespace is pretty much isolated. The only interface that shows up is loopback. We cannot even ping the loopback interface as it is down. First

of all, let's bring it back to life.

```
P# ip link set dev lo up
P# ping 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.034 ms
P# ping google.com
ping: google.com: Temporary failure in name resolution
```

Our process is able to communicate only with the loopback interface. That is, communication is feasible only for members of this network namespace. Our process definitely feels a bit lonely. Besides, our namespace seems to be useless, if communication with the outside world is in our plans. We can relax the isolation by creating a tunnel through which processes in 'pan' can interact with processes in our initial namespace. We will use a virtual ethernet network device (or veth for short) to fulfill our need. Veth devices are always created as a pair of devices in a tunnel-like fashion so that messages written to the device on one end comes out of the device on the other end.

veth The veth devices are virtual Ethernet devices. They can act as tunnels between network namespaces to create a bridge to a physical network device in another namespace, but can also be used as standalone network devices. A particularly interesting use case is to place one end of a veth pair in one network namespace and the other end in another network namespace, thus allowing communication between network namespaces [39].

From the manual page of *veth* we realize that we could easily have one end in the initial network namespace and the other in our child network namespace and have all inter-network-namespace communication go via the respective veth end device.

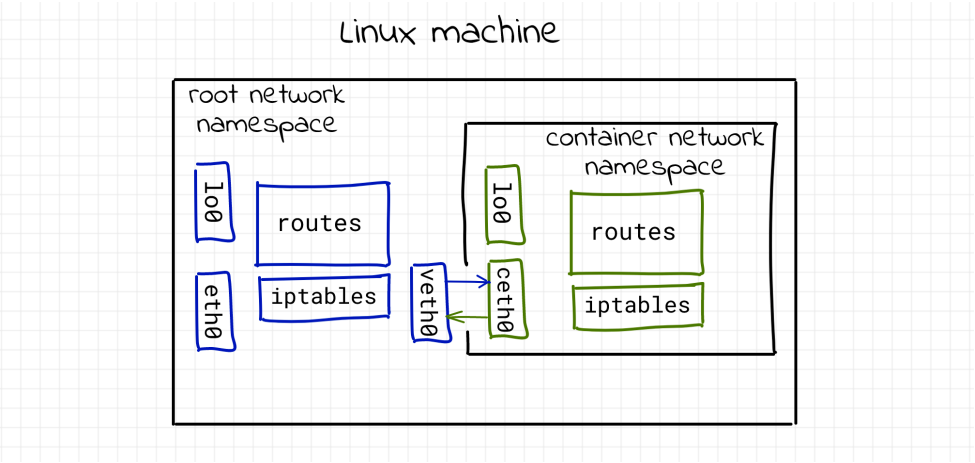


Figure 2.8: *Simple visualization of how veth pairs connect namespaces [40]*

Time to add a veth pair in both namespaces, the command is:

```
ip link add <p1-name> netns <p1-ns> type veth peer <p2-name> netns <p2-ns>
```

We will skip the first netns with <pi-ns> because the initial network namespace doesn't have a name.

```
# ip link add veth0 type veth peer veth1 netns pan
# ip link list
# ip link show veth0
3: veth0@if2: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT
 group default qlen 1000
    link/ether aa:c5:e1:0c:a5:cb brd ff:ff:ff:ff:ff:ff link-netns pan
P$ ip link show veth1
2: veth1@if3: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT
 group default qlen 1000
    link/ether fa:73:55:c4:61:4d brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

Notice that from our initial namespace the veth interface is linked to the pan namespace (link-netns pan). Respectively, from the pan namespace the veth interface is linked to the network namespace with id equal to 0. To achieve the desired communication both of the interfaces need an IP address in the same subnet.

```
# ip addr add 10.1.1.1/24 dev veth0
# ip link set dev veth0 up
$P ip addr add 10.1.1.2/24 dev veth1
$P ip link set dev veth1 up
$P ip addr show veth1
2: veth1@if3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
 group default qlen 1000
```

```
    link/ether fa:73:55:c4:61:4d brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.1.1.2/24 scope global veth1
       valid_lft forever preferred_lft forever
    inet6 fe80::f873:55ff:fec4:614d/64 scope link
       valid_lft forever preferred_lft forever
```

The last command proves that the new IP address is properly assigned to the veth1 interface. The big time has come. Can we truly communicate with the initial namespace?

```
P$ ping 10.1.1.1
PING 10.1.1.1 (10.1.1.1) 56(84) bytes of data.
64 bytes from 10.1.1.1: icmp_seq=1 ttl=64 time=0.067 ms
```

2.3.5 Linux Control Groups

The Control groups mechanism, usually referred to as "cgroups", are a Linux kernel feature which allows processes to be organized into hierarchical groups whose usage of various types of resources can then be limited and monitored. The kernel's cgroup interface is provided through a pseudo-filesystem called cgroupfs. Grouping is implemented in the core cgroup kernel code, while resource tracking and limits are implemented in a set of per-resource-type subsystems (memory, CPU, and so on).

With the term cgroup we refer to a collection of processes that are bound to a set of limits or parameters defined via the cgroup filesystem. In a nutshell, cgroups limit the amount of resources a process can consume and allows organizing processes into hierarchical groups. These cgroups are nothing more than values defined in particular files under the pseudo-fs cgroupfs. A subsystem is a kernel component that modifies the behavior of the processes in a cgroup. There are various subsystems, which thereby provide a variety of functionality, such as limiting the amount of CPU time and memory available to a cgroup, accounting for the CPU time used by a cgroup, and freezing and resuming execution of the processes in a cgroup. Subsystems are sometimes also known as resource controllers (or simply, controllers).

The cgroups for a controller are arranged in a hierarchy. This hierarchy is defined by creating, removing, and renaming subdirectories within the cgroup filesystem. At each level of the hierarchy, attributes (e.g., limits) can be defined. The limits, control, and accounting provided by cgroups generally have effect throughout the subhierarchy under-

neath the cgroup where the attributes are defined. Thus, for example, the limits placed on a cgroup at a higher level in the hierarchy cannot be exceeded by descendant cgroups [41].

Cgroup directories are living at */sys/fs/cgroup/*. Inspecting this directory divulges the various available controllers. We present two of the most well-known controllers (cpu, memory) to let any reader who is unfamiliar with cgroups better comprehend their purpose.

- **cpu & cpuacct**

  These two controllers were merged into one in the second version of cgroups. The cpu controller guarantees a minimum number of CPU shares when the system is busy. At the same time provides a CPU bandwidth control by the means that it is possible to set an upper limit on the CPU time allocated within a scheduling period to the processes in cgroup. cpuacct provides accounting for CPU usage by groups of processes. CPU stats and usage are stored within a file:

  ```
  cat /sys/fs/cgroup/cpu,cpuacct/docker/<<pid>>/cpuacct.stat
  ```

- **memory**

  The memory controller, which is present since Linux 2.6.25, supports reporting and limiting of process memory, kernel memory, and swap used by cgroups.

  ```
  /sys/fs/cgroup/memory/docker/
  ```

We can modify those files accordingly and configure cgroups. One of the basic properties of Docker (or generally containers) is the ability to control memory limits. By default containers have no limit on the memory. Editing the *memory.limit_in_bytes* sets different memory limits for a process. For example:

```
echo 8000000 > /sys/fs/cgroup/memory/docker/<<pid>>/memory.limit_in_bytes
```

2.3.6 Seccomp & AppArmor

All actions with Linux are done via syscalls. The kernel has 330 system calls that perform operations such as read files, close handles and check access rights. All applications use a combination of these system calls to perform the required operations.

- **AppArmor** is an application defined profile that describes which parts of the system a process can access.
- **Seccomp** provides the ability to limit which system calls can be made, blocking aspects such as installing Kernel Modules or changing the file permissions.

2.3.7 Capabilities

Traditional UNIX implementations distinguish two categories of processes:

- *privileged processes*, whose effective user ID is 0, referred to as superuser or root and
- *unprivileged processes* whose effective UID is nonzero

Privileged processes bypass all kernel permission checks, while unprivileged processes are subject to full permission checking based on the process's credentials (usually: effective UID, effective GID, and supplementary group list) [42].

Starting with kernel 2.2, Linux divides the privileges traditionally associated with superuser into distinct units, known as capabilities, which can be independently enabled and disabled. Briefly, Linux capabilities split root's permission into smaller categories that distinguish what different root processes have permission to do. These capabilities might cover multiple system calls or actions, such as changing the system time or hostname.

Some of the capabilities are: CHOWN, CAP_SYS_ADMIN, SYS_CHROOT, SYSLOG, SYS_TIME.

It is very important to have these chapters as a reference and come back to them whenever required. Especially, the network namespace will help you effortlessly understand everything related to Kubernetes networking concepts. Although the secret parts of containers are not the protagonists of this diploma thesis, we felt that it is a nice opportunity to unfold them and keep the relevant sections as a guide every time you get baffled playing around with containers.

To summarize:

- namespaces → what you can see
- cgroups → what you can use
- capabilities → splits root privileges
- seccomp→ limits syscalls
- AppArmor/SELinux → limits various access (syscalls, files, etc) via LSM hooks

2.4 Container Images

Multiple articles mention containers as a worthy substitute of Virtual Machines. On the contrary, even though containers offer some kind of isolation and protection (not to the extent of VMs), they mostly emerged as a technology to cover the need for reusability and reproducibility of software applications. Virtual Machines offer great isolation and containers great flexibility. In the 2.3 our center of attention was the way containers protect themselves using namespaces and cgroups. To make them flexible, transferable and plugable to any possible environment, software engineering teams needed a way to freeze the state of containers, so that they can later ship these frozen containers to users that will run the containerized applications.

This frozen version of a container is called a container image, and it's a concept first introduced by Docker. Using a programming metaphor, if an image is a class, then a container is an instance of a class – a runtime object. A container image essentially is a static representation that determines the execution of a container. This means that it contains information about both the structure of the containerized filesystem and configurations related to the runtime execution. In a few words, a container image is an immutable file which essentially describes a snapshot of the container. The image's file system is created by stacking up a list of read-only layers, which – in the case of Docker – may represent instructions given in a "Dockerfile". The layers are read-only, and each layer is merely a set of differences from the layer before it, as they are stacked on top of each other.

Containers and Layers

When we create a new container from an image, in reality we add a new writable layer

on top of the underlying layers, which is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. When the container is deleted, the writable layer is also deleted. The major difference between a container and an image is the top writable layer. The underlying image remains unchanged. Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state [5].

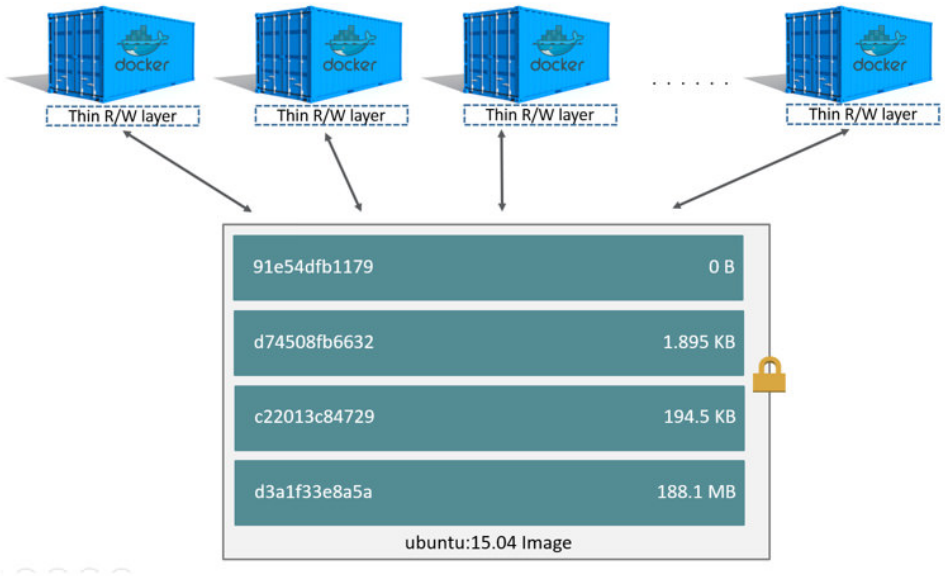


Figure 2.9: *Docker containers with their own writable container layer and the same underlying image [5]*

Storage Drivers

Docker uses storage drivers to store image layers. Docker calls storage driver the mechanism that handles the details about the exact manner that all these layers interact with one another. The storage driver controls how images and containers are stored and managed on the host system. Various storage drivers are available, and some of them are based on union filesystems, there is one such storage driver based on AUFS and two storage drivers based on OverlayFS [5].

When images need to be downloaded from a registry, each layer is downloaded separately. Then, it is cached locally by Docker daemon for future use: in the case that two or more images that must be used have some common layers, those layers will not be downloaded multiple times. Since each container has its own writable container layer,

and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. This is effectively a Copy-On-Write mechanism in action: if a file or directory exists in a lower layer within the image, and another layer – including the writable layer – needs read access to it, it just uses the existing file. On the first time that another layer needs to modify the file, either while building the image or while running the container, the file is copied into that layer and that copy is modified.

2.5 Container Runtimes

There are many ways to run containers and the best known one is, with no doubt, Docker, which sky-rocketed the container adoption. There is also Podman and Buildah. CRI-O is also an option acting as a higher level container runtime which has been written on purpose to be used with the Kubernetes CRI. All of the above are called container runtimes with a common purpose which is to run containers. To clear things out a little, let's talk about the tool that is at the core of Docker, Podman, CRI-O and Containerd: runc. runc is the original container runtime on which most of the tools are being developed.

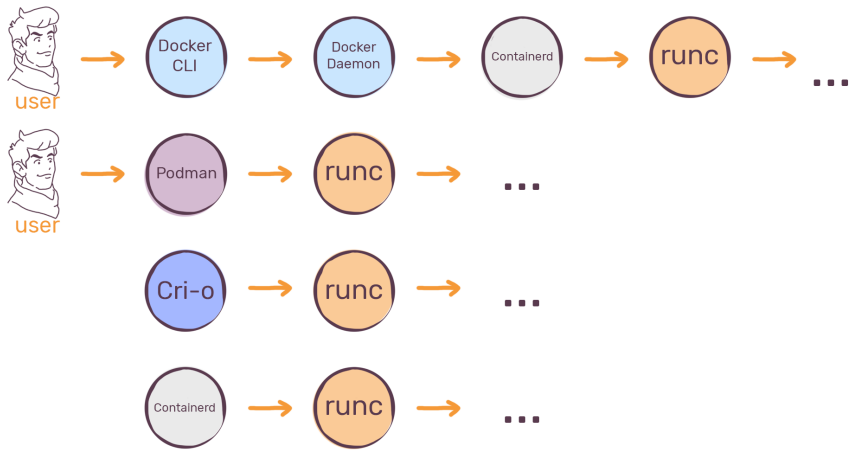


Figure 2.10: *Most of the container runtimes eventually call runc [6]*

runc runc is a command line client for running applications packaged according to the Open Container Initiative (OCI) format and is a compliant implementation of the Open Container Initiative specification [7]

runc doesn't have a concept of "images", like Podman or Docker do. You cannot just execute *runc run ubuntu:latest*. Instead, runc expects you to provide an OCI runtime bundle, which is basically a root filesystem and a configuration file. The definition of a bundle is only concerned with how a container, and its configuration data, are stored on a local filesystem so that it can be consumed by a compliant runtime.

A Standard Container bundle contains all the information needed to load and run a container. This includes the following artifacts:

- *config.json*: contains configuration data. This REQUIRED file MUST reside in the root of the bundle directory and MUST be named config.json.
- *container's root filesystem*: the directory referenced by root.path, if that property is set in config.json.

Docker, Podman, CRI-O are responsible for unpacking a container image into a runtime bundle. Then runc takes over and runs the container from the bundle. For more on how to convert an OCI container image into a OCI runtime bundle refer to the official documentation: [8]

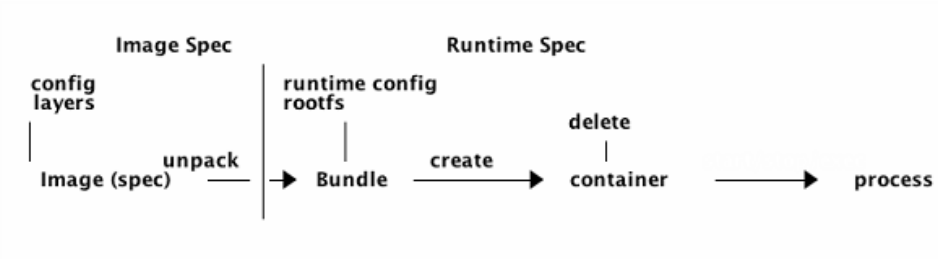


Figure 2.11: *From a container image to a running process [9].*

2.6 Docker Architecture and Workflow

In the words of its backing company, Docker Inc., Docker is an open platform for developing, shipping, and running applications which are separated from the underlying infrastructure, thus enabling easy and quick software delivery and infrastructure management. This is achieved, basically, by providing the ability to package and run applications in containers via the Docker Engine, its container runtime engine. In addition,

Docker provides tooling and a platform to manage the lifecycle of the containers, aiming to aid the development of the applications and their supporting components using containers, render the container the unit for distributing and testing these applications, and finally deploy them into production environments, regardless of whether that is a local datacenter, a cloud provider or a hybrid of the two.

Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing Docker containers. The Docker client and daemon can run on the same system, or a Docker client can be connected to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface [10].

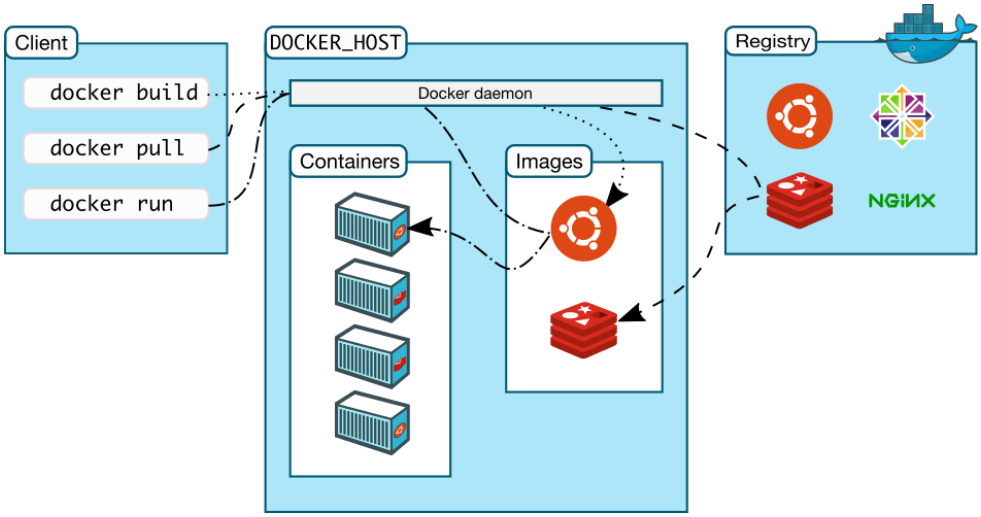


Figure 2.12: *Docker architecture [10]*

By default, the Docker daemon automatically starts *containerd*. *containerd* is also a daemon which manages the complete container lifecycle of its host system, from image transfer and storage to container execution and supervision to low-level storage to network attachments and beyond. Most of the container feature sets are handled via runc. In the high level architecture as depicted in figure 2.12, we can observe one more component other than the Docker host and the Docker client, which may or may not run on the same host machine. This component is the Docker registry. A Docker registry is merely a repository for storing Docker images. There are registries that are publicly available, such as the Docker Hub. However, it is also possible for anyone to run their own private Docker registries, to organize the stored images as deemed best for each set of use cases.

2.7 Kubernetes

Containers provide a way for applications to run inside isolated, immutable and reproducible environments. The last few years the procedure of launching a container has become trivial and negligible because every developer does it on a regular basis. The rise of containers and microservices technology resulted in applications that now comprise hundreds or sometimes thousands of containers. Managing those loads of containers across multiple environments, executing health-checks or recovering from a failure using scripts and self-made tools can be really complex or even impossible. This scenario increased the demand for a proper way of managing a large number of containers. Kubernetes is an open source container orchestration tool that satisfies the need of abstracting the control, supervision and administration of numerous containers. The name Kubernetes originates from Greek, meaning helmsman or pilot. K8s as an abbreviation results from counting the eight letters between the ”K” and the ”s”. Google open-sourced the Kubernetes project in 2014 [11].

Kubernetes guarantees high availability or no downtime, which in simple terms means that an application deployed on Kubernetes will be always accessible by the user. For example, if a container goes down, another container needs to start. Moreover, K8s claims scalability based on demand, providing ways to scale applications dynamically. Thus, it offers high performance and high response rates. This includes scaling down CPU, Memory and Network resources when workload levels are low, saving tons of money for the one hosting the app. K8s mechanism provides disaster recovery and self-healing, from restarting failing containers to backing up data and restoring an app. Kubernetes is ubiquitous in today's cloud computing landscape mostly because complexity is hidden by simple abstractions.

What makes Kubernetes so special is that it works on the basis of declaration of intent, instead of carrying out imperative commands. You declaratively define the desired state for the system and the system will be constantly monitoring itself and strive to achieve this state. It comprises a set of independent, composable control processes that continuously drive the current state towards the provided desired state. This automatically removes a hard requirement of knowing the inner workings of Kubernetes.

2.7.1 Kubernetes Architecture

Kubernetes operates at the container level rather than at a hardware level. It offers features such as deployment, scaling, and load balancing which imitate platform specific services. It expands in one or more physical or virtual machines called nodes. All in all, when someone deploys Kubernetes gets back a cluster consisting of one or more nodes with enhanced capabilities that run containerized applications. At the end of the day, Kubernetes is a software that runs containers on Nodes with special care. To do so, it follows a Master-Slave architectural pattern. The worker node (Slave) host the components of the application workload (Pods). The control plane (Master) 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. In a nutshell, Nodes describe their desired state and Control Plane tries to reconcile the actual/current state of the cluster with the desired state.

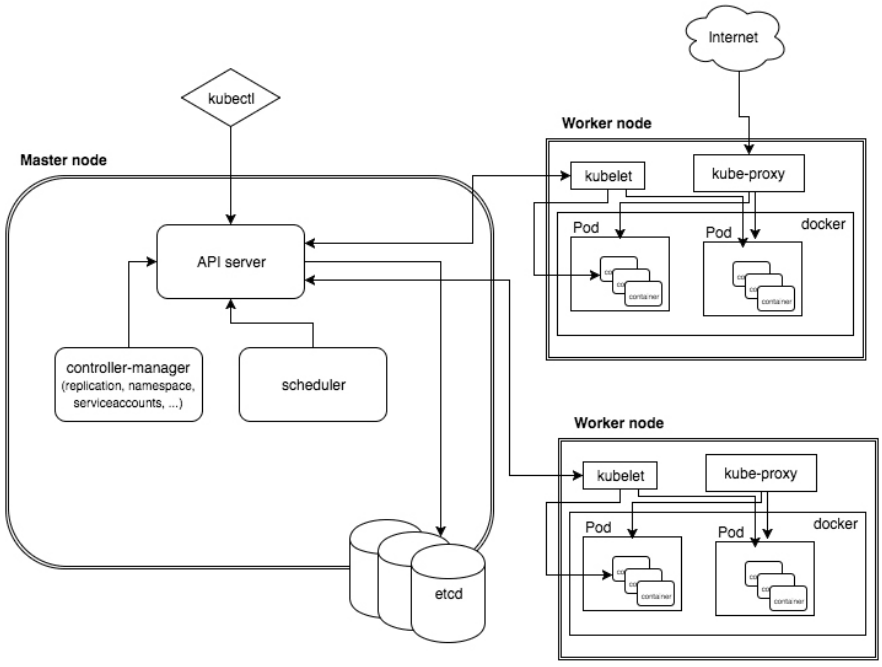


Figure 2.13: *A simplified view of Kubernetes' Architecture*

Both the Control Plane and the Nodes consist of some necessary components needed to have a complete and working Kubernetes cluster. As we said, the Master node manages all the worker nodes, thus it takes global decisions about the whole cluster [43]. To do

so, it consists of:

etcd
The central storage of K8s. etcd is a strongly consistent, distributed key-value store that provides a reliable way to store data belonging to a cluster of machines. Every single piece of persistent data in K8s is stored in etcd.

kube-apiserver
The API server, as the name suggests, exposes the Kubernetes API and listens to all the relevant REST requests. In other words, it is the front end for the Kubernetes Control Plane. This means that every request for modifications goes first through the API server.

kube-controller-manager
The controller-manager runs and manages all the controller processes, which monitor the state of K8s objects and adjust them to the desired state that they imply. In essence, instead of increasing complexity with separate processes for each controller, they are all combined into one controller manager.

kube-scheduler
Scheduler's job is to assign Pods to Nodes based on their specifications. The scheduler takes into consideration the resource requirements, software constraints, and data locality as factors for the final decision.

Node components run on every node and they are responsible for running containers inside that node.

kubelet
The kubelet is an agent that runs on every node in a Kubernetes cluster and is responsible for, among other things, managing the lifecycle of Pods. This means it handles all of the translation logic between the abstraction of a "Pod" (which is really just a Kubernetes concept) and its building blocks, containers. At first the Control Plane schedules a Pod into a Node. Then it is kubelet's (and so the node's) responsibility to initiate containers in that Pod and maintain the Pod's desired status. Therefore, it queries the kube-apiserver to synchronize the Pod's state and modify it if any discrepancies exist. Eventually, we can think about kubelet as a controller functioning in the inner levels of a Pod. On the contrary, control plane controllers perceive Pods as a K8s abstraction and manipulate them as a whole entity.

kube-proxy

kube-proxy is a network proxy that runs on each node in the cluster, implementing part of the Kubernetes Service concept. kube-proxy maintains network rules on nodes. These network rules allow network communication to the Pods from network sessions inside or outside of the cluster. kube-proxy uses the operating system packet filtering layer (i.e. iptables) if there is one and it's available. Otherwise, kube-proxy forwards the traffic itself. Essentially, the network packets visit the kube-proxy at first which redirects/forwards them to the appropriate network service tightly coupled with Pods.

Container Runtime

The container runtime is the software that is responsible for running containers. K8s supports container runtimes such as containerd, CRI-O, and any other implementation of the Kubernetes CRI (Container Runtime Interface). Kubelet communicates with container runtimes to initiate the process of starting and running containers in Pods.

2.7.2 Objects

We referred a lot of times to Pods. We mentioned that Pods are a K8s construct, that is, kubernetes objects. But what does that exactly mean? Kubernetes objects are persistent entities in the Kubernetes system stored in the etcd and they can be expressed in a .yaml format [44]. They describe and represent the state of the cluster. For example, a Pod object indicates a Pod's existence in the cluster. An object by itself operates as a record of intent. Once it is created, the system will work to ensure that it exists. The Kubernetes API exposes the endpoints to create, modify and delete objects. The two important fields in an Object are its Spec and its Status.

Spec For objects that have a spec, it is set when creating the object, providing a description of the characteristics we want the resource to have: its *desired* state.

Status The status describes the current state of the object, supplied and updated by the K8s system and its components. The Kubernetes control plane continually and actively manages every object's actual state to match the desired state we supplied.

For example, a Deployment is an object that can be used to represent an application running on the cluster. When creating a Deployment, we can specify in the Spec that

we want 2 replicas (Pods) of the application up and running. The Deployment Controller watches for Deployment Objects on the API Server, and when it notices the new Deployment, it creates 2 Pod Objects. If any of those Pods fail (a change which will be reflected in the Status field) the Deployment Controller will try to reconcile the difference between desired-actual state by launching a replacement instance. Here is an example of an nginx Deployment, which is a popular kind of Object:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

As you can see a Deployment is a collection of Pods and a Pod is a superset of containers. There are numerous objects with a variety of purposes like services and networking, storage, configuration objects and more.

2.7.3 Controllers

Controllers are responsible for managing a Kubernetes Object. The technique followed is a non-terminating loop that regulates the state of Object, known also as control loop. A very representative example of a control loop is a thermostat in a room. We set the temperature at a desired state and the actual temperature is the current state. The thermostat (controller) acts to bring the current state closer to the desired state, by turning equipment on or off [45]. Pretty much everything in Kubernetes is a Controller, diverging from an imperative modus operandi, which makes it remarkable and unique. Finally, a controller tracks one or more K8s resource types. That is, a Deployment controller is in control of deployment objects. Regularly, a controller will communicate with the API server, sending the appropriate requests that have the necessary side effects for making the current state come closer to the desired state of the system.

2.7.4 Pods

A Pod is a collection of one or more containers tied together with vital information related to storage, network resources and specification for how to run containers, i.e. the entrypoint, the environment variables and more. It is the smallest deployable K8s unit and stands as the base for more complex objects. The containers in a Pod are automatically co-located and co-scheduled on the same physical or virtual machine in the cluster. The containers can communicate with one another and share resources. Note that to scale an application horizontally the recommended way is to use multiple Pods, also referred to as replicas.

2.7.5 The Kubernetes Networking Model

Kubernetes was built to run distributed systems over a cluster of machines. The very nature of distributed systems makes networking a central and necessary component of Kubernetes deployment. Kubernetes makes opinionated choices about how Pods are networked [46]. In particular, Kubernetes dictates the following requirements on any networking implementation:

- Containers within a Pod use networking to communicate via loopback

- all Pods can communicate with all other Pods without using network address translation (NAT)
- all Nodes can communicate with all Pods without NAT
- the IP that a Pod sees itself as is the same IP that others see it as

Given the above constraints we need to examine different networking situations:

1. container-to-container
2. pod-to-pod
3. pod-to-service
4. internet-to-service
5. DNS

Container-to-Container Networking

A pod consists of one or more containers that exist on the same host and are configured to *share* the same network stack and other resources such as volumes. Technically this means that all the containers in a pod can reach each other on localhost. This is cool but also provokes some minor problems such as every container has to listen on a different port which is not so different from the situation of running multiple processes on a single host. In reality, the situation is more subtle than that. In Linux, each running process communicates within a network namespace that provides a logical networking stack with its own routes, firewall rules, and network devices. In essence, a network namespace provides a brand new network stack for all the processes within the namespace. It's somewhat obvious that by default Linux assigns every process to the root network namespace to provide access to the external world. So, eth0 lives inside the root network namespace.

However, one of the main/crucial parts of containerization is namespaces. This means that K8s shouldn't work in a different way. Inside a node, we have pods and in terms of Docker constructs, a Pod is modeled as a group of Docker containers that share a network namespace. Containers within a Pod can find each other via localhost since they reside in the same namespace. Moreover, containers can be accessed through the

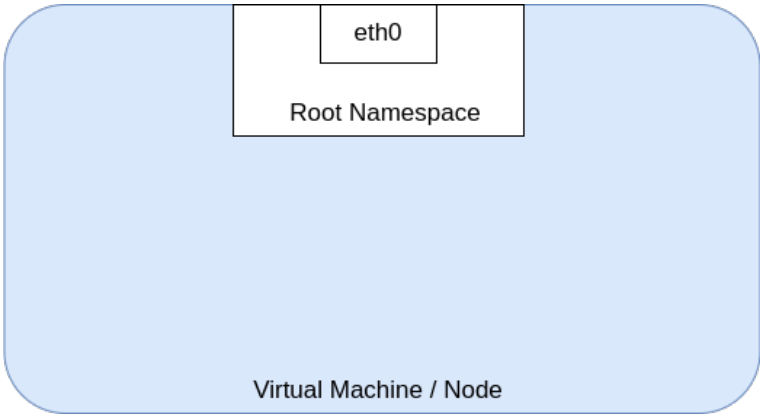


Figure 2.14: *The root network namespace*

Pod's IP address and specific port assigned to them. We can translate this to : Containers within a Pod all have the same IP address. To conclude, inside a Pod we know exactly how the communication works (using localhost). However this is not enough (or maybe it's almost useless). The very heart of Kubernetes' design requires that pods be able to communicate with other pods even if they live in different Nodes.

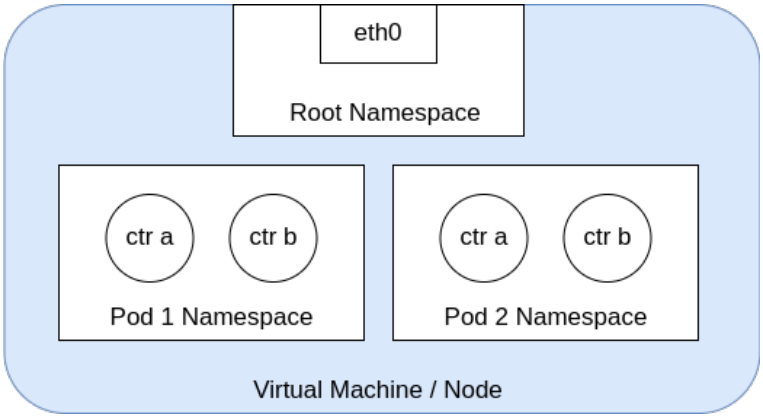


Figure 2.15: *A network namespace per pod*

Pod-to-Pod Networking

Kubernetes assumes that pods can communicate with other pods, regardless of which host they land on. Kubernetes gives every pod its own cluster-private IP address, so you do not need to explicitly create links between pods or map container ports to host ports [47]. This means that containers within a Pod can all reach each other's ports on localhost, and all pods in a cluster can see each other without NAT.

In k8s every Pod has a real IP address and each Pod communicates with other Pods using that IP address. As we saw in figure 2.15, every Pod lives in its own different namespace and also has its own ethernet interface. Each pod needs to communicate with the other network namespaces on the same Node using its *eth0* interface, right?

Thankfully, namespaces can be connected using a Linux Virtual Ethernet Device or *veth pair* consisting of two virtual interfaces that can be spread over multiple Namespaces. Veth devices are always created as a pair of devices in a tunnel-like fashion so that messages written to the device on one end comes out of the device on the other end. To connect Pod namespaces, we can assign one side of the veth pair to the root network namespace, and the other side to the Pod's network namespace. You can fresh your memory about network namespaces in the relevant paragraph of 2.3.4 section.

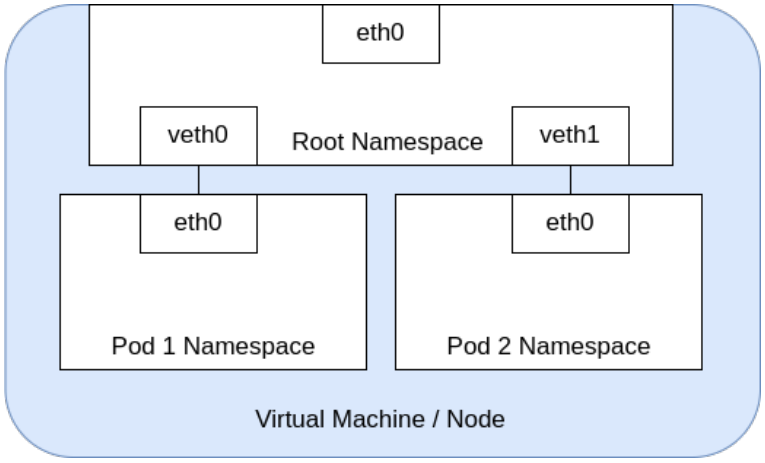


Figure 2.16: *veth pairs per Pod*

With the above setup we achieved the connection of the root namespace with the pods' namespaces. This of course is not enough, because we want our pods to communicate with each other. If *pod1* pings the IP address of *pod2*, packets will eventually reach the *veth0* interface, but what happens next? Someone has to send the packets to the right destination. What if there were 10 pods? Enter *Network Bridge*

Network Bridge

Bridges are self-taught and learn every information they want by sending *arp* packets. A Linux Ethernet bridge is a Layer 2 networking device used to unite two or more network segments, working transparently. In brief, The bridge operates by maintaining a forwarding table between sources and destinations by examining the destination of the data packets that travel through it and deciding whether or not to pass the packets to other network segments connected to the bridge.

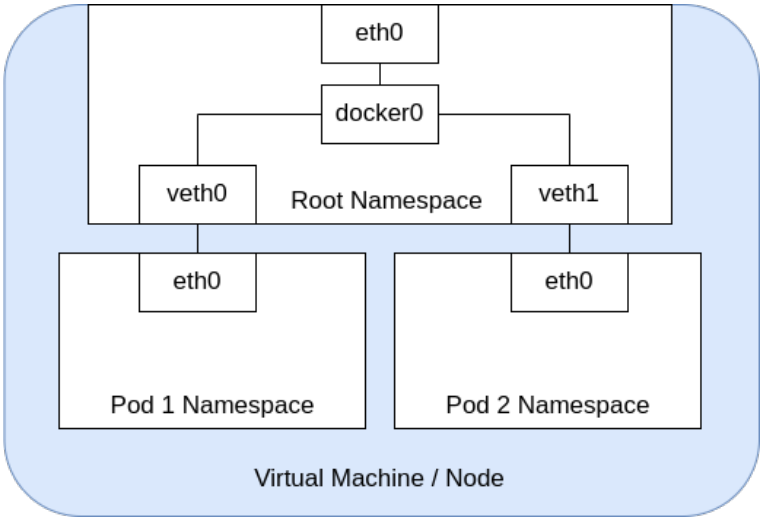


Figure 2.17: *Connecting namespaces using a bridge*

Pod-to-Service Networking

There's something very very important that we've missed till this time. Everything works great until we need to deal with change. Kubernetes Pods are mortal: they are born and when they die, they are not resurrected. Pod IP addresses are not durable and will appear and disappear in response to scaling up or down, application crashes, or Node reboots. Each of these events can make the Pod IP address change without warning. In other words, the set of Pods running in one moment in time could be different from the set of Pods running that application a moment later. If some set of Pods (call them "backends") provides functionality to other Pods (call them "frontends") inside your cluster, how do the frontends find out and keep track of which IP address to connect to, so that the frontend can use the backend part of the workload [48]? Enter Services.

Services Services were built into Kubernetes to address this problem. A Kubernetes Service manages the state of a set of Pods, allowing you to track a set of Pod IP addresses that are dynamically changing over time. Services act as an abstraction over Pods and assign a single virtual IP address to a group of Pod IP addresses. Any traffic addressed to the virtual IP of the Service will be routed to the set of Pods that are associated with the virtual IP. This allows the set of Pods associated with a Service to change at any time — clients only need to know the Service's virtual IP, which does not change.

ClusterIP

Every time we create a new k8s service (whatever type it is) a new virtual IP called cluster IP is created on our behalf. Note that this IP is indeed virtual by the means of it doesn't exist at all. For example you'll not find this IP using a *nettool* like *ifconfig*. Anywhere within the cluster, traffic addressed to the virtual IP will be load-balanced to the set of backing Pods associated with the Service. How k8s achieves this? Answer comes with two suspects: *netfilter* and *iptables*

- *Netfilter* is a framework provided by Linux that allows various networking-related operations to be implemented in the form of customized handlers. Netfilter offers various functions and operations for packet filtering, network address translation, and port translation, which provide the functionality required for directing packets through a network, as well as for providing the ability to prohibit packets from reaching sensitive locations within a computer network.

- *Iptables* is a user-space program providing a table-based system for defining rules for manipulating and transforming packets using the netfilter framework. In Kubernetes, iptables rules are configured by the kube-proxy controller that watches the Kubernetes API server for changes. When a change to a Service or Pod updates the virtual IP address of the Service or the IP address of a Pod, iptables rules are updated to correctly route traffic directed at a Service to a backing Pod.

kube-proxy watches the Kubernetes control plane for the addition and removal of Service and Endpoint objects. For each Service, it installs iptables rules, which capture traffic to the Service's clusterIP and port, and redirect that traffic to one of the Service's backend sets. For each Endpoint object, it installs iptables rules which select a backend Pod.

Internet-to-Service Networking

So far we've looked at how k8s achieve communication within a cluster. This is all fine and good but unfortunately at some point we will want to expose our service to the outside world. We have to deal with the below concept: How to forward traffic from a k8s service out to the Internet and vice versa.

In order for pods in different nodes to communicate, we unite/connect nodes with a cluster gateway which has the appropriate entries to forward traffic. Moreover, every Node is assigned a private IP address that is accessible from within the K8s cluster. To make traffic accessible from outside the cluster, we use an Internet gateway (not the same with the cluster gateway) which is attached to our cluster network. The Internet gateway serves two purposes:

1. *Forward traffic* from/to Internet to/from nodes using the corresponding entries of the route table

2. *Perform Network Address Translation (NAT)*. Nodes have private IP addresses which are not accessible from the public Internet. The NAT translation is responsible for changing the Node's internal IP address that is private to the cluster to an external IP address that is available in the public Internet

Getting traffic into our cluster is a surprisingly tricky problem to solve. For starters, let's define the problem. The cluster IP of a service is only reachable from a node's ethernet interface. Nothing outside the cluster knows what to do with addresses in that range. This wonderful illustration will make you understand the problem.

Kubernetes network unites all the "VMs" of the k8s cluster using one cluster gateway and one switch. But then, how can a machine in the same LAN but outside the k8s network communicate with services and pods inside the cluster? There's really no reliable way to do this using routing without some active management of the router, which is exactly kube-proxy's role in managing netfilter. In general, routing internet traffic to K8s is divided into three solutions that work on different parts of the network stack:

NodePort In simple terms a NodePort makes LAN to Inner Cluster Communication possible. NodePort services open a consistent port on every node in the cluster and map traffic that comes in on that port to a service inside the cluster. (Kube-proxy is what's responsible for redirecting traffic coming in on the Node Port to the service.) NodePort service is somewhat misnamed, NodesPort service would have been a better name as when this service is created a port is, by default, randomly chosen from the range 30000-32767, and every node in the cluster starts listening on that port.

LoadBalancer Load Balancing is the process of distributing network traffic across multiple servers. This ensures no single server bears too much demand. By spreading the work evenly, load balancing improves responsiveness. Load balancing methods also increase availability of applications and websites for users. Modern applications cannot run without load balancing.

Ingress Ingress means to enter. Kubernetes Ingress refers to external traffic being routed to a Kubernetes Ingress Controller, which is really just a Layer 7 Load Balancer that exists inside of a cluster [49]. The Ingress Controller dynamically configures itself based on Ingress yaml objects, which are snippets of configuration that declaratively describe desired Layer 7 Routing. Ingress is not a service type like NodePort, ClusterIP, or LoadBalancer. Ingress actually acts as a proxy to bring traffic into the cluster, then uses internal service routing to get the traffic where it is going. Under the hood, Ingress will use a NodePort or LoadBalancer service to expose itself to the world so it can act as that proxy. An Ingress is a higher-level HTTP load balancer that maps HTTP requests to Kubernetes Services. The life of a packet flowing through an Ingress is very similar to that of a LoadBalancer. The key differences are that an Ingress is aware of the URI's path allowing and can route traffic to services based on their path.

DNS for Services and Pods

Kubernetes creates DNS records for services and pods. You can contact services with consistent DNS names instead of IP addresses. Every Service defined in the cluster (including the DNS server itself) is assigned a DNS name. By default, a client Pod's DNS search list includes the Pod's own namespace and the cluster's default domain. A DNS query may return different results based on the namespace of the pod making it. DNS queries that don't specify a namespace are limited to the pod's namespace. We can access services in other namespaces by specifying it in the DNS query [50]. For example, consider a pod in test namespace. A data service is in the prod namespace. If we get inside the pod and make a dns query for data using nslookup returns no results, because it uses the pod's test namespace. However, a query for data.prod returns the intended result, because it specifies the namespace.

In Kubernetes land 3 networks tend to exist:

1. Internet
2. LAN
3. Inner Cluster Network

Each of these 3 networks have their own DNS.

1. *Internet DNS*: 8.8.8.8, 1.1.1.1, 9.9.9.9 (google, cloudflare, quad9 or whatever public internet DNS the router is configured to point to.)
2. *LAN DNS*: 192.168.1.1 (LAN DNS hosted on your router)
3. *CoreDNS*: 10.43.0.10 (10th IP of the CIDR range of the inner cluster network)

The crucial question is who can resolve what.

- A *pod* can resolve DNS entries hosted at any of these 3 levels of DNS.
- The OS hosting the Kubernetes Cluster can only resolve DNS entries hosted on LAN DNS or Internet DNS. (the OS isn't scoped to have visibility into the existence of CoreDNS/Inner Cluster Network.)
- *kubelet* + *docker/containerd/cri-o* exist at the OS level in the form of systemd services and thus don't have scope to Inner Cluster DNS names.

2.8 Kubeflow

Most industries working with large amounts of data have recognized the value of machine learning technology. By gleaning insights from this data – often in real time – organizations are able to work more efficiently or gain an advantage over competitors. A common practice today is to deploy ML jobs as containers managed by the Kubernetes orchestrator. Creating an ML model that can predict what we want it to predict from an offline holdout dataset we ourselves have fed is easy. However, creating an ML model that is reliable, fast, accurate, and can be used by a large number of users is difficult. To build an ML system that will continuously operate in production is a significantly heavy job for a data scientist. Monitoring, debugging, versioning and reproducibility of ML models are all burdensome procedures for them too. Many teams have data scientists and ML researchers who can build state-of-the-art models, but their process for building and deploying ML models is entirely manual. This may be sufficient when models are rarely changed or trained. In practice, models often break when they are deployed in the real world. The models fail to adapt to changes in the dynamics of the environment, or changes in the data that describes the environment.

Kubeflow is a machine learning toolkit for Kubernetes strived to make deployments of machine learning (ML) workflows on Kubernetes simple, portable and scalable. Its goal is to enable using machine learning pipelines to orchestrate complicated workflows running on Kubernetes [12]. An ML pipeline is a workflow, that is a series of procedures that are chained together and that ultimately produce a ready-to-use, machine learning model. Pipelines consist of a number of steps, each one of which runs as an independent process with regards to the others. Each step performs a specific task of a machine learning workflow, e.g: preprocessing/cleaning a dataset, training a model, producing predictions on a test subset and serving a model. Of course, such pipelines can run locally, on a single machine. Though, if one wants to leverage more than what a single machine can offer, they need to run such pipelines distributedly (e.g: on entire clusters, on Kubernetes).

With Kubeflow, data scientists and engineers are now able to develop a complete pipeline composed of segmented steps. These segmented steps in Kubeflow are loosely coupled components of an ML pipeline, allowing pipelines to become easily reusable and modifiable for other jobs. That is, someone is able to develop and build a shareable and

reusable Kubeflow Component. This added flexibility has the potential to save a lot of labor necessary to develop a new data pipeline for each specific use case. Through this process, Kubeflow aims to simplify Kubernetes deployments while also accounting for future needs of portability and scalability.

Ultimately, Kubeflow's dream is to have a set of simple manifests that gives an easy-to-use ML stack anywhere Kubernetes is already running, and that can self configure based on the cluster it deploys into [12]. Apart from that, Kubeflow manages to diminish the gap between development and production level, by uniting the two environments into one offering all the necessary tools for each stage.

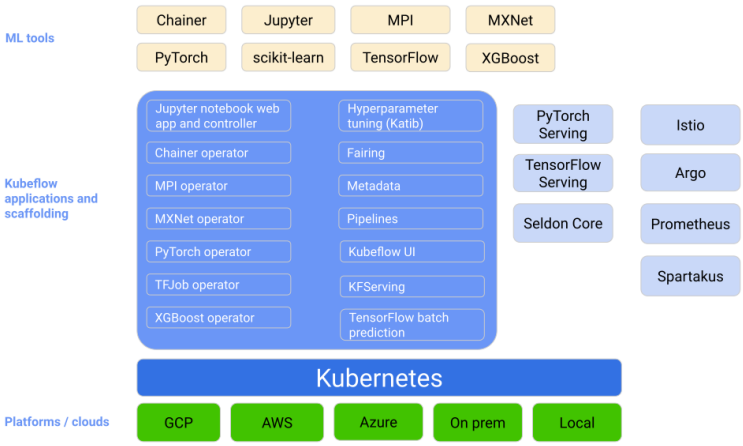


Figure 2.18: *Kubeflow Platform and tools*

Architecture

The Machine Learning workflow consists of several specific stages on which we continuously iterate. Specifically, we analyze and evaluate the outputs of each stage and then we modify and fine tune if needed the model and its parameters to keep producing the results we want. We can split the ML workflow into two main phases, the *experimental* and the *production* phase [51].

1. In the *experimental phase* we identify the problem we want to solve using ML, then we collect and analyze the data that will train our model, we experiment with them and with training. Finally, hyperparameter tuning takes place to achieve best performance and accuracy.

2. In the *production phase*, we first transform the data into the appropriate format (this process should not differ from experimental to production phase). We train

the ML model and then we serve it for online prediction. Finally, we monitor the model's performance and continuously feed the results for tuning and retraining. Kubeflow's ultimate goal is to provide useful tools that will automate and make each stage of the ML workflow easier for the data scientists. The below image illustrates what Kubeflow offers to all phases.

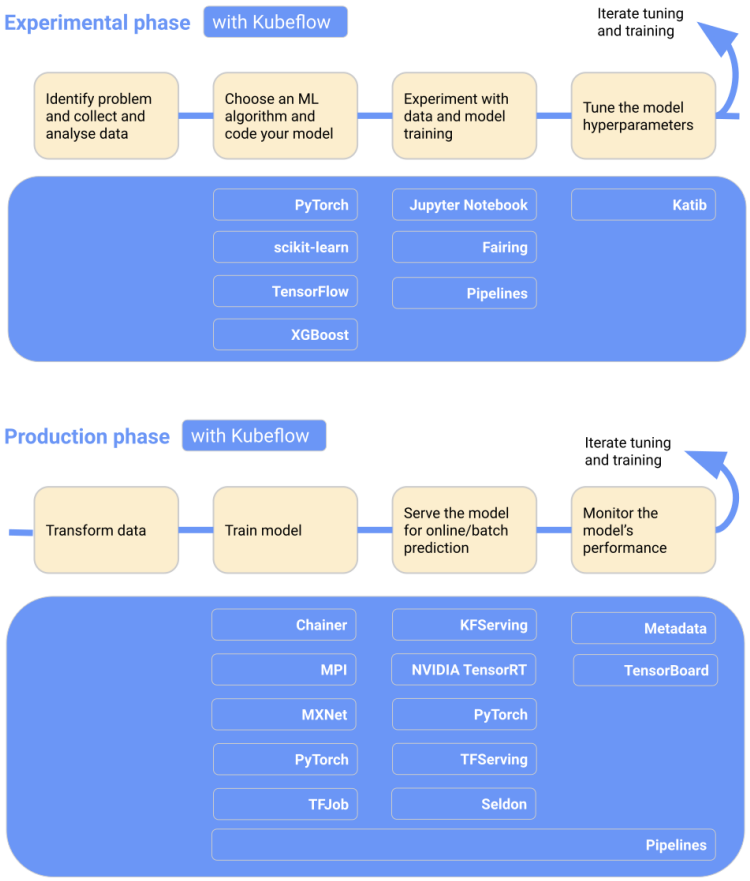


Figure 2.19: *Kubeflow workflow overview*

Central Dashboard

In the Kubeflow's central dashboard we can quickly access most of the components including notebooks, pipelines, experiments, runs, models, artifacts and many more.

Kubeflow brought together the development and production phases of Machine Learning, uniting all the immersive ML tools into one platform on Kubernetes. Kubeflow is an ML space containing everything a Data Scientist would wish for. However, it hasn't yet reached the point where ML engineers will use them seamlessly. Regarding Kubeflow Pipelines which represent an ML workflow, users are forced to do a lot of changes in their

initial code to integrate it into Kubeflow and convert it into a pipeline. Moreover, it is not feasible to convert a Jupyter notebook (Data Scientists' main tool) into a Kubeflow Pipeline.

A pipeline consists of multiple components. A pipeline component is a self-contained set of user code, packaged as a Docker image, that performs one step in the pipeline. Users must build their own Docker Image for the component, including all the component's dependencies. Apart from the fact that this forces them to acquire knowledge related to DevOps, it also makes it a lot harder to debug and quickly deploy their model. In this diploma thesis we'll see how we extend Kale (the data scientist's superfood) to confront this problem.

2.9 Kale & Rok

KALE (Kubeflow Automated pipeLines Engine) is a project that aims at simplifying the Data Science experience of deploying Kubeflow Pipelines workflows.

Figure 2.20: *The Kale's Logo*

Kubeflow is a great platform for orchestrating complex workflows on top of Kubernetes and Kubeflow Pipeline provides the means to create reusable components that can be executed as part of workflows. The self-service nature of Kubeflow make it extremely appealing for Data Science use, as it provides an easy access to advanced distributed jobs orchestration, reusability of components, Jupyter Notebooks, rich UIs and more. Still, developing and maintaining Kubeflow workflows can be hard for data scientists, who may not be experts in working orchestration platforms and related SDKs. Additionally, data science often involves processes of data exploration, iterative modeling and interactive environments (mostly Jupyter notebook) [13].

Kale bridges this gap by providing a simple UI to define Kubeflow Pipelines workflows directly from the JupyterLab interface, without the need to change a single line of code.

The Kale SDK provides the simplest way possible to convert any repository of Python code into fully reproducible Kubeflow Pipelines (KFP) runs without changing the source code [14]. Essentially, Kale offers:

1. A *Jupyter UI* extension that allows turning Jupyter Notebooks to Kubeflow Pipelines. Kale makes this feasible by letting users annotate code cells with specific tags to dictate the steps of the workflow-pipeline and the dependencies between them. Then, Kale is responsible for converting the user's annotated Notebook to a working Kubeflow Pipeline, as well as taking care of the data-passing between steps.

2. An *SDK*, for turning plain Python code into Kubeflow Pipelines. The Kale Software Development Kit allows users to write Python, function-based code and convert it to a Kubeflow pipeline without making almost any change to the original source code.

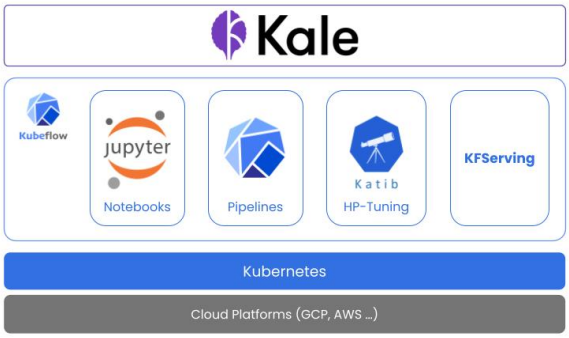


Figure 2.21: *Kale operates on top of Kubeflow*

But how does Kale manage to build the required Docker Images for each Pipeline component? In fact, it doesn't! This is where Rok comes into play. Instead of building the component's image, Rok snapshots the data scientist's working environment (code + data), clones it and attaches it to the Pods where each component will run. The Rok data management and storage platform gives you the best of both worlds: the performance of local storage and the flexibility of shared storage.

Rok allows you to run your stateful containers over fast, local NVMe storage on-prem or on the cloud, and still be able to snapshot the whole application, along with its data and distribute it efficiently: across machines of the same Kubernetes cluster, or across

distinct locations and administrative domains over a decentralized network. This hybrid approach to container networking makes data management for machine learning and real-time big data applications simple, without sacrificing performance [15].

In this fashion, we can easily generate production-ready ML pipelines with "point and click" operations and roll back instantly to any pipeline step for rapid debugging and collaboration. Unfortunately, with this approach, we cannot produce reusable and shareable KFP components, since a component should be packaged into a Docker image. Enter this diploma thesis, where we transparently build images on the fly making the process of building reusable and reproducible KFP components as seamless as possible.

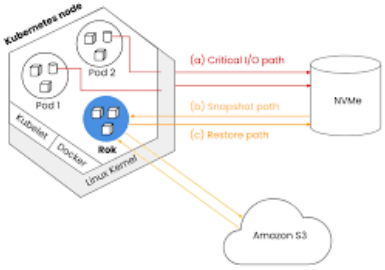


Figure 2.22: *Rok Data Management*

3 Design

3.1 Overview

Developing and maintaining Kubeflow workflows is hard and challenging for data scientists with no expertise in working orchestration platforms and related SDKs. The Kubeflow Pipelines Python SDK is a great tool to automate the creation of these pipelines, especially when dealing with complex workflows and production environments. Still, when presenting this technology to ML researchers or Data Scientists that don't have strong software engineering expertise, KFP can be perceived as too complex and hard to use. Kubeflow pipelines still have a big automation and reproducibility gap, since they cannot yet detach users from Container and Kubernetes concepts. Consequently, users are frequently asked to build Docker images for their Kubeflow Pipeline. Data Science is often a matter of prototyping new ideas, exploring new data and models, experimenting fast and iteratively. In these scenarios one would prefer to just run some rough code and analyze the results rather than setting up complex workflows with a specific SDK.

Kale with Rok made a first huge step and bridged this gap by providing a simple UI to define Kubeflow Pipelines workflows directly from the JupyterLab interface, without the need to change a single line of code. In addition, the Kale SDK provided the simplest way possible to convert any repository of Python code into fully reproducible Kubeflow Pipelines (KFP) runs without changing the source code. With the help of Rok Snapshots Kale replicates and transfers the user's working environment into the Kubernetes Pods where each step of the Pipeline will run. This approach, though, comes with a significant drawback. It does not allow Kale to produce standalone, reproducible and shareable

KFP components, since all the required code is not packaged between the base Docker image of the component and its entrypoint as it should, but it is attached through clones of the snapshotted volumes created by Rok.

In order to create proper KFP components the first part of this thesis comes to grips with container images and how to build them as seamlessly as possible, making the process of creating reusable KFP components a point and click operation. Specifically, we thoroughly analyze and gradually unveil well known concepts around container images. This information will help us sculpt our algorithm for extending container images in unprivileged environments. With that in our hands, we design and develop a build image mechanism that will be the keystone to create proper KFP components.

In the second part of the chapter, we dive into Kubeflow concepts around building reproducible pipelines and components. We examine the vulnerabilities of the approach, how Kale with Rok managed to save us from most of Kubeflow's boilerplate and how our contribution streamlined the process of building and deploying shareable and reproducible KFP components.

Eventually, this chapter showcases and designates the purpose and the rationale behind this thesis. Starting with the flaws of the current approaches we unravel the reasoning that led us to the implementation of our system.

3.2 The Anatomy of Container Images

In the previous sections we thoroughly discussed containers and how engineers leveraged various linux kernel technologies to imitate virtual machines by isolating (namespaces) and restricting (cgroups, capabilities, seccomp) linux processes. In other words, we mainly touched containers from the virtualization perspective. Let's remind ourselves of the container's definition:

container A container is any receptacle or enclosure for holding a product used in storage, packaging, and transportation, including shipping. Things kept inside of a container are protected on several sides by being inside of its structure.

We focused on the ways to protect our package, to seclude and restrict it. This may mislead us from the initial objective of containers which goes beyond just providing

protection for the contents. A well-designed container will also exhibit convenience and usability, that is, it is easy for the worker to open or close, to insert or extract the contents, and to handle the container in shipment. This section's purpose is to demonstrate how Linux containers manage to fulfill and satisfy most (if not all) of the characteristics that delineate them.

As we already mentioned, containers have a frozen state in which they are files saved on disk. This is what we call a container image. In a few words, a container image is an immutable file which essentially describes a snapshot of the container. The image's file system is created by stacking up a list of read-only layers, by using a union filesystem. Then, when a container is instantiated from this image, a thin writable layer is added on top of the read-only ones. This layer is also called the "container layer".

At the end of this section you should be able to answer the following questions:

1. What a container image is
2. What happens when you build an image
3. How to manually modify an image
4. How to move from rest (container image) to run (container)

Have you ever questioned what a container image looks like? It is important to know how Container Runtimes, like Docker, build, store and then share and ship images. Then, how are these images used by containers? What does *node:latest* or what dockerfile instructions like *FROM ubuntu:latest* do really represent?

```
$ sudo docker run -it node:latest /bin/bash
Unable to find image 'node:latest' locally
latest: Pulling from library/node
6aefca2dc61d: Pull complete
967757d56527: Pull complete
c357e2c68cb3: Pull complete
c766e27afb21: Pull complete
32a180f5cf85: Pull complete
3507b5066a40: Pull complete
3c68557c340d: Pull complete
925b4ef5803f: Pull complete
3c263125b4e4: Pull complete
Digest: sha256:71c779ea8a157e6efadcafdae79f00fb39b7f3fdb2819ebef3984315c281540f
```

```
Status: Downloaded newer image for node:latest
```

What about all these little details hiding in the above output? It feels like chaos at first, but on the contrary container images are very well defined.

Docker was the first back in 2013 who was able to pack containers into images. This enabled users to move container images between machines, which marked the birth of container based application deployments. That is, being able to package your application with all the required libraries, dependencies and files into a versioned, reproducible and shareable image. As of today, multiple versions of the container image format exist, whereas the Docker developers decided to create version 2 (V2) schema 1 of the image manifest back in 2016 and therefore deprecated version 1. After multiple iterations of various image format improvement approaches, schema 2 has been released to supersede the existing schema 1 in 2017. The V2 image specification was later donated to the Open Container Initiative (OCI) as the base for the creation of the OCI image specification.

3.2.1 Open Container Initiative

Going back to the container's definition really helps to realize why engineers made some decisions in the first place. What would happen if we tried to ship, stack and transport irregular and erratic shapes and sizes of boxes? Probably havoc. The OCI image specification comes to solve this problem defining how to package a container into an image.

This specification defines an OCI Image, consisting of a) a manifest, b) an image index (optional), c) a set of filesystem layers and d) a configuration [16]. Yes, that's all we need to describe a container image, whereas everything except the layers are written in JavaScript Object Notation (JSON). The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run.

1. An *Image Index* is an annotated index of image manifests
2. At a high level an *Image Manifest* is a document describing the components that make up a container image
3. A *Filesystem Layer* is a changeset that describes a container's filesystem

4. An *Image Configuration* is a document determining the layer ordering and configuration of the image suitable for translation into a runtime bundle

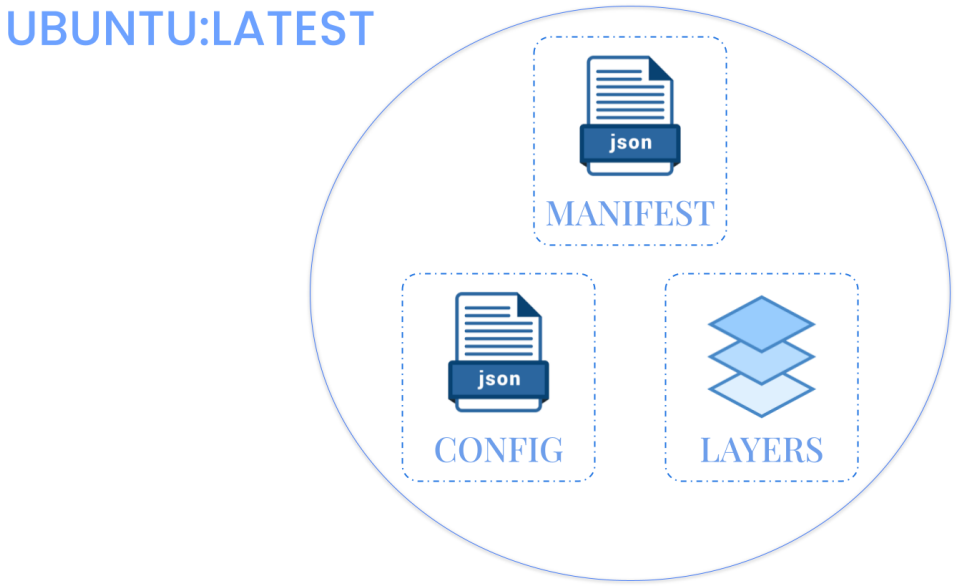


Figure 3.1: *The Anatomy of Container Images*

In this section we follow a top-down approach in order to present the components and the relationships between them. Before moving on, *skopeo* tool will help us get a first idea of what an OCI container image looks like. *skopeo* is a command line utility that performs various operations on container images and image repositories.

We pull the *busybox* image (one of the simplest) from the docker hub (image registry) in an OCI format. Note that an OCI format is different from a Docker Image format, although they share a lot of similarities. A full comparison of the two of them will soon be presented.

```
oci-image$ skopeo copy docker://busybox:latest oci:busybox:latest
Getting image source signatures
Copying blob 8ec32b265e94 done
Copying config da3528d175 done
Writing manifest to image destination
Storing signatures
oci-image$ ls
busybox
```

Skopeo brings the image's files to our local filesystem in an OCI format allowing us to inspect and manipulate them.

```
oci-image/busybox$ tree .
.
├── blobs
│   └── sha256
│       ├── 8ec32b265e94aafb0d43ab71f1d8f786122c19afb37d25532aea169f414f8881
│       ├── b4ebf27383f009f3b9dc1298cf1d536b64575c0143585be469e7aea11b180eee
│       └── da3528d17538d46636ffaf7bde44c68b9a210f55dfb859626930c141e9519d00
├── index.json
└── oci-layout

2 directories, 5 files
```

At first sight we notice the Image Index (→ *index.json* file) and three completely obscure file names. Where is the *Image Manifest*, the *Configuration* and the *Layers*?

```
busybox/blobs/sha256$ file *
2a22f49c2fb14bae16fb907f6f36a9583a1f38f0044647077883221c7fa3645c: JSON data
4ff913783dfa58b73c1234c3111419ad7da07bd1f58462334a631bc1bb6d2417: JSON data
50e8d59317eb665383b2ef4d9434aeaa394dcd6f54b96bb7810fdde583e9c2d1: gzip
compressed data, original size modulo 2^32 1459200
```

Given that the image manifest and the configuration are documents written in JSON format, an educational guess would be that they correspond to the first two files and the final one is the image's layer which we haven't yet defined its format. Albeit, we know that it describes the container's filesystem.

3.2.2 Image Index

The image index is a higher-level manifest which points to specific image manifests, ideal for one or more platforms. While the use of an image index is optional for image providers, image consumers should be prepared to process them [17]. *jq* command is used as a JSON processor [52] and helps us inspect the contents of the *index.json* file.

```
{
  "schemaVersion": 2,
  "manifests": [
    {
      "mediaType": "application/vnd.oci.image.manifest.v1+json",
      "digest": "sha256:2a2...45c",
      "size": 347,
```

```
      "annotations": {
        "org.opencontainers.image.ref.name": "latest"
      }
    }
  ]
}
```

Apparently, the image index contains an *array of manifests*("manifests"). In fact, it serves as an array of pointers that points to the manifest files. It looks like the image index is just a higher-level manifest, which contains pointers to more specific image manifests. These manifests seem to be valid only for their dedicated target platforms, which contain a specific operating system (os) like linux and an architecture, like amd64.

The ID that uniquely identifies them is a hash *digest* computed (exported by) from the contents of the relevant image manifest. As already mentioned, the image index is fully optional.

mediaType is something that we will come across a lot of times. It describes the type and the format of the corresponding content and can be seen as an identifier of the object. For example, in this case the media type is:

```
application/vnd.oci.image.manifest.v1+json
```

That is, the file is a version 1 manifest file of an OCI image with JSON format. Eventually, our educational guess was correct, since:

$$2a22f49c2fb14bae16fb907f6f36a9583a1f38f0044647077883221c7fa3645c$$

is the name of one of the files we spotted in the image's file structure.

3.2.3 Image Manifest

We referred a lot of times to the Image Manifest. It is the right moment to solve the mystery. It is one of the most important components for a container image. Unlike the image index, which contains information about a set of images that can span a variety of architectures and operating systems, an image manifest provides a configuration and set of layers for a single container image for a specific architecture and operating system

[18]. Actually, it contains metadata about: 1) *the configuration file* and 2) the *filesystem layers*. These metadata contain the:

- *mediaType*: Identifies the format of the corresponding resource. For example, a tar gzipped Docker image layer has the type: *application/vnd.docker.image.rootfs.diff.tar.gzip*
- *digest*: Checksum of the corresponding resource computed with a hash algorithm like sha256
- *size*: Size of the resource

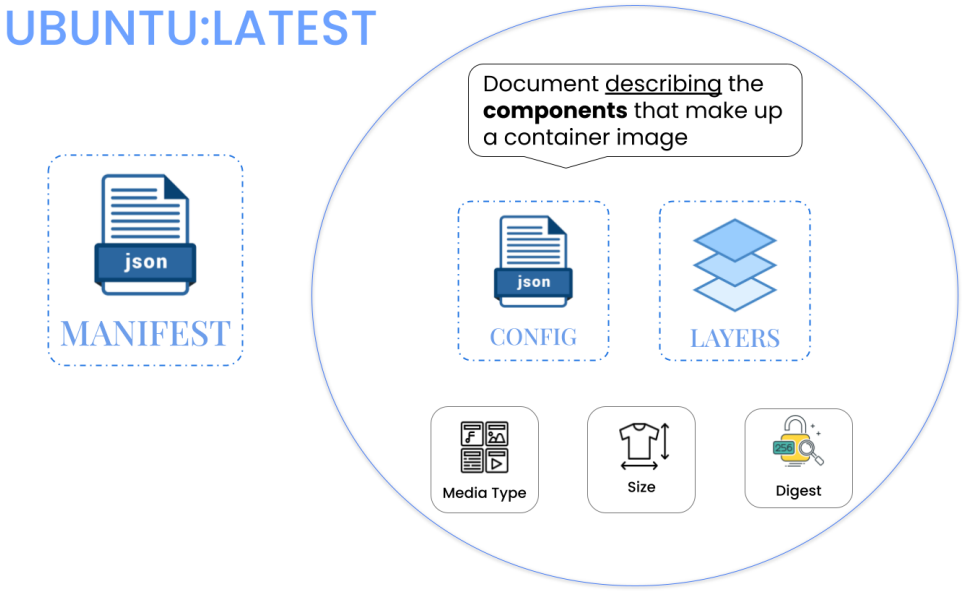


Figure 3.2: *The Anatomy of Container Images: Image Manifest*

Consequently, we could say that the image manifest provides the location to the configuration and the set of layers along with crucial information like each object's size. The *Image Index* revealed to us the *manifest's* location which will reveal to us the configuration and layer's location. This time we investigate the contents of 2a22f49c2fb14bae16fb907f6f36a9583a1f38f0044647077883221c7fa3645c file (the manifest file according to index.json).

```
{
  "schemaVersion": 2,
  "config": {
```

```
    "mediaType": "application/vnd.oci.image.config.v1+json",
    "digest": "sha256:4ff...417",
    "size": 575
  },
  "layers": [
    {
      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
      "digest": "sha256:50e...2d1",
      "size": 772812
    }
  ]
}
```


Indeed, the Image Manifest contains a field called *config* and another one called *layers*. The first one tells us the "address", the *size* and *mediaType* of the configuration file. The latter is an array of pointers to all the filesystem image's layers along with their size and mediaType. Notice that the layer is a tar gzip file:

$$application/vnd.oci.image.layer.v1.tar + gzip$$

as we already saw in the beginning while investigating the image's structure. The digest is computed from the compressed version of the layer.

3.2.4 Image Configuration

We are now able to obtain further information about the image configuration.


```
{
  "created": "2022-04-14T02:29:36.517566461Z",
  "architecture": "amd64",
  "os": "linux",
  "config": {
    "Env": [
      "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
    ],
    "Cmd": [
      "sh"
    ]
```

```
  },
  "rootfs": {
    "type": "layers",
    "diff_ids": [
      "sha256:eb6b01329ebe73e209e44a616a0e16c2b8e91de6f719df9c35e6cdadadbe5965"
    ]
  },
  "history": [
    {
      "created": "2022-04-14T02:29:36.368193089Z",
      "created_by": "/bin/sh -c #(nop) ADD file:1c8...ce02 in / "
    },
    {
      "created": "2022-04-14T02:29:36.517566461Z",
      "created_by": "/bin/sh -c #(nop)  CMD [\"sh\"]",
      "empty_layer": true
    }
  ]
}
```


On the top of the JSON we find some metadata, like the *created* date, the *architecture* and the *os* of the image. The environment (Env) as well as the command (Cmd) to be executed can be found in the *config* section of the configuration. This section can contain even more parameters, like the working directory (WorkingDir), the User which should run the container process or the Entrypoint. This means, if you create a container image via a Dockerfile, then all these parameters will be converted into the JSON-based configuration.

Notice that an OCI Image is an ordered collection of root filesystem changes (rootfs) and the corresponding execution parameters for use within a container runtime [19]. The configuration indicates that the rootfs is split into layers, whereas the diff_ids are different from the actual layer digests found in the manifest file, because they reference the digests of the uncompressed tar archives. Do not confuse DiffIDs with layer digests, often referenced in the manifest, which are digests over compressed or uncompressed content. **Layer ChainID**

How do we ensure that the DiffIDs will be extracted in the right order? Enter ChainID. For convenience, it is sometimes useful to refer to a stack of layers with a single iden-

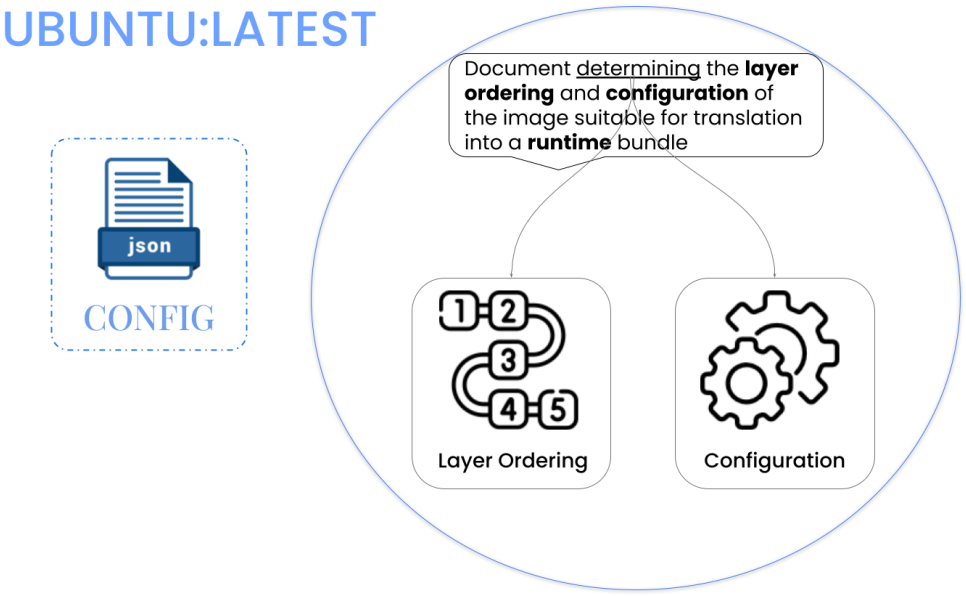


Figure 3.3: *The Anatomy of Container Images: Image Configuration*

tifier. While a layer's DiffID identifies a single changeset, the ChainID identifies the subsequent application of those changesets. This ensures that we have handles referring to both the layer itself, as well as the result of the application of a series of changesets. Use in combination with rootfs.diff_ids while applying layers to a root filesystem to uniquely and safely identify the result.

```
ChainID(₀L) =  DiffID(₀L) ChainID(₀L|...|ₙ₋₁L|ₙL) =
Digest(ChainID(₀L|...|ₙ₋₁L) + " " + DiffID(ₙL))
```

3.2.5 What is a Layer?

Until now we've seen that an OCI image consists of an *Index*, a *Manifest* and a *Configuration* file. All of these are JSON documents which describe some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes. This JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers.

In other words, each image comprises JSON documents providing required information related to the container runtime and a number of layers that compose the final container's filesystem.

In particular, each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer. Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer.

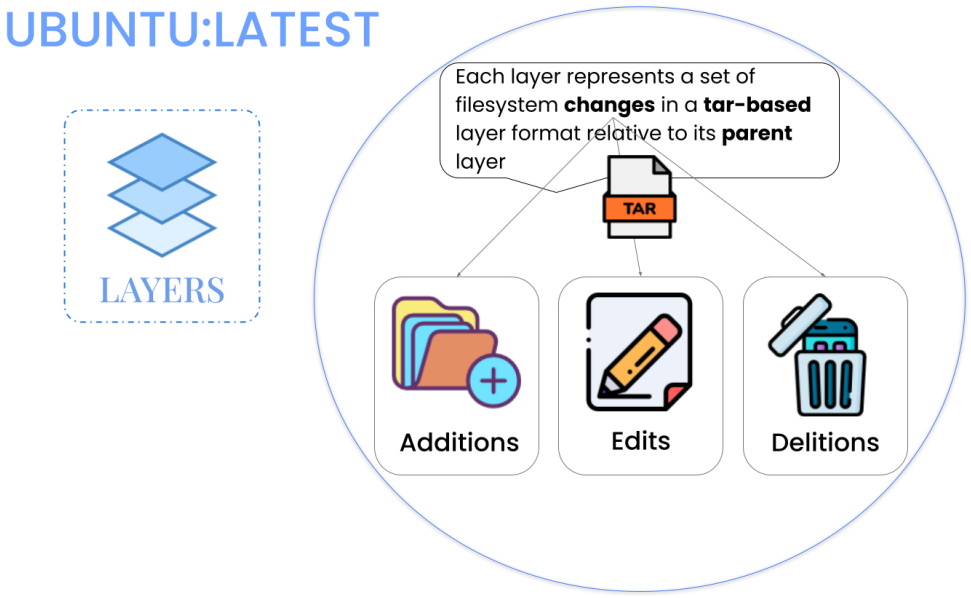


Figure 3.4: *The Anatomy of Container Images: Image Layer*

How are these changes really represented? Decompressing and extracting a layer to investigate its contents will reveal the truth.

```
$ cd blobs/sha256
blobs/sha256$ mkdir layer-files
blobs/sha256$ tar -xvf \
> 50e8d59317eb665383b2ef4d9434aeaa394dcd6f54b96bb7810fdde583e9c2d1 \
> -C layer-files/
blobs/sha256$ ls layer-files/
bin  dev  etc  home  root  tmp  usr  var
```

In this case, the image is composed of only one layer, so we cannot actually realize what a change means practically. However, it is clear that layers are a bunch of directories and files, literally formulating a filesystem. An easy win would be to guess how someone adds new files to the existing image. Apparently, a new layer stacked upon the initial one containing only the new additions would produce the desired result. Let's assume that our initial filesystem contains the below:

```
rootfs/
```

```
etc/
    my-app-config
bin/
    my-app-binary
    my-app-tools
```

The first and initial layer will be a plain *tar archive* with relative path to rootfs. The entries of the tar file are:

```
./
./etc/
./etc/my-app-config
./bin/
./bin/my-app-binary
./bin/my-app-tools
```

In our example we would like the following changes to happen:
Added: /etc/my-app.d/
Added: /etc/my-app.d/default.cfg
Modified: /bin/my-app-tools
Deleted: /etc/my-app-config

This reflects the *removal* of */etc/my-app-config* and the *creation* of a file and directory at */etc/my-app.d/default.cfg*. */bin/my-app-tools* has also been *replaced* with an updated version. A tar archive is then created which contains only this changeset representing the new layer:

- Added and modified files and directories in their entirety
- Deleted files or directories marked with a whiteout file

The resulting tar archive includes the following:

```
./etc/my-app.d/
./etc/my-app.d/default.cfg
./bin/my-app-tools
./etc/.wh.my-app-config
```

Essentially, a new directory is added, then a new file under that directory is added too. Regarding modifications, we just add the updated file to the tar archive. Finally, to

signify that the resource *./etc/my-app-config* must be removed when the changeset is applied, the basename of the entry is prefixed with *.wh.* (whiteout file) [53].

All in all, layers are mainly tarballs which can be decompressed and extracted with conventional tools like GNU tar, although they require special care in cases where whiteout files are included in the archive. Only in the absence of any whiteout files in a layer changeset, the archive is extracted like a regular tar archive. This means that we can download the layers in parallel, but they have to be extracted sequentially. This takes time and is a significant drawback when it comes to pulling images from remote locations.

To summarize, layers portray **filesystem changes**. This is the reason why in the configuration file - under rootfs field - the name to represent the layer's content hashes is called diff_ids. Note that diff_ids are the digest over the layer's **uncompressed** tar archive, which means that the hashes actually represent the differences between layers. In essence, the point of the configuration file and the image's layers is to be unpacked to a runtime bundle, which is a set of files organized in a certain way, and containing all the necessary data and metadata for any compliant runtime to perform all standard operations against it.

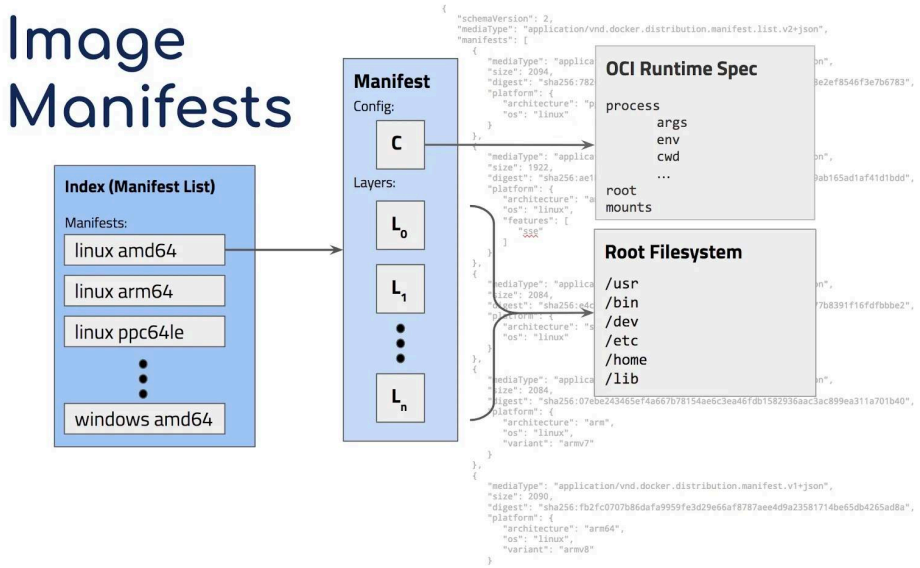


Figure 3.5: *The real purpose of manifests is to produce content-addressable images*

3.2.6 Sharing is caring

The real purpose of manifests is to produce content-addressable images, by supporting an image model where the image's configuration can be hashed to generate a unique ID for the image and its components [18]. More precisely, multiple manifest files can point to different combinations of configuration files and layers producing different images with the same amount of layers. That is, images can share layers in the same space with no need to replicate layer's files. Manifests will tell which layers and configuration files belong to which images. In other words, they are the source of truth, giving great flexibility, space and time efficiency. There is no need to download an image as a whole entity, some layers may already exist in your filesystem, so you only pull the ones missing. Notice that the familiar to most of us image tags are tightly coupled with the manifest. Each tag points to one manifest file. When tags are empty, a digest is used to point to a manifest file. In fact, this digest is the checksum of the manifest's content.

Imagine that you have two different Dockerfiles. The first one creates an image called '*my-base-image:1.0*'.

```
FROM alpine
RUN apk add --no-cache bash
```

The second one is based on '*my-base-image:1.0*', but has some additional layers:

```
FROM my-base-image:1.0
COPY . /app
RUN chmod +x /app/hello.sh
CMD /app/hello.sh
```

The second image contains all the layers from the first image, plus new layers created by the COPY and RUN instructions. Docker already has all the layers from the first image, so it does not need to pull them again. The two images share layers they have in common [5]. As you can see in Figure 3.6, the remote registry comprises two manifests that represent two different image tags, that is, two different images. However, the images share the first layer and therefore it appears only once. When executing:

```
docker pull docker.io/my-registry/my-image:v1
```

Docker grabs the manifest file with v1 tag, opens it and finds, with respect to the digest, the configuration file and the layers of the image to bring in the local filesystem. Consequently, a registry repository contains multiple images with different tags or digests,

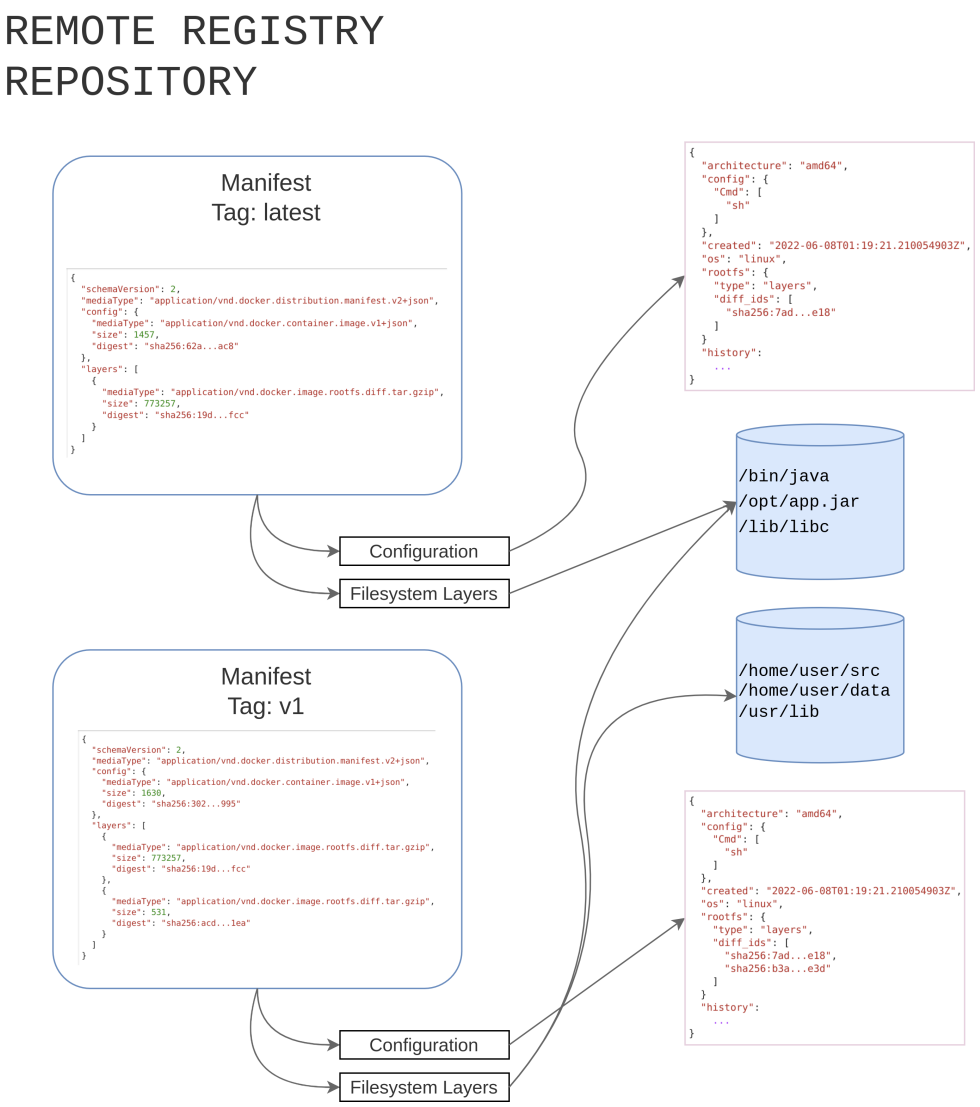


Figure 3.6: *Two manifest files point to different layers and configuration file composing two different images.*

that is, different manifest files. Layers for all images are stored together, deduplicating potential shared layers.

3.2.7 Docker Images

In the beginning of this chapter we mentioned that Docker was the first to introduce a way to pack containers into images. Their first version was released in 2013. After multiple iterations Docker ended up with a version 2 schema 2 approach for container images. This image specification was later donated to the Open Container Initiative as the base for the creation of the OCI image specification which we thoroughly analyzed

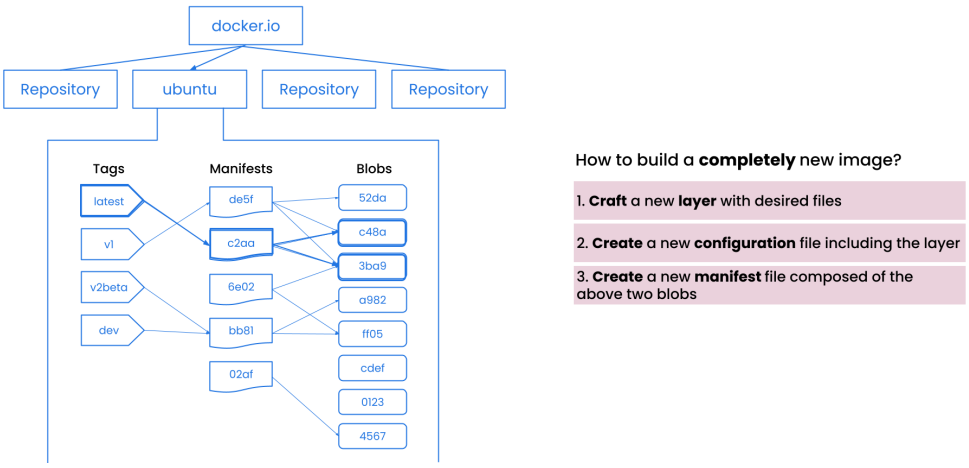


Figure 3.7: *A registry contains multiple repositories, each one of them comprise different tag and manifests composing multiple images.*

previously. As of today, multiple versions of the container image format exist. Having said that, what are the differences between the v2_2 Docker Image specification and the OCI Image specification? Inspecting a Docker image in the v2_2 format will reveal the truth. Note that *skopeo* is still used to bring an image to our local filesystem in the desired format, here v2 schema2 Docker image.

```
$ skopeo copy docker://busybox:latest dir:busybox
Getting image source signatures
Copying blob 8ec32b265e94 done
Copying config 42b97d3c2a done
Writing manifest to image destination
Storing signatures
$ tree busybox/
busybox/
├── 42b97d3c2ae95232263a04324aaf656dc80e7792dee6629a9eff276cdfb806c0
├── 8ec32b265e94aafb0d43ab71f1d8f786122c19afb37d25532aea169f414f8881
├── manifest.json
└── version
```

Notice that the *index.json* is missing, but as the OCI specification states it is not mandatory. The *manifest.json* file is definitely the image manifest we exhaustively examined before. The other two (apart from the version file) are the configuration file and the image's layer.

```
$ file busybox/*
42b97d3c2ae95232263a04324aaf656dc80e7792dee6629a9eff276cdfb806c0: JSON data
8ec32b265e94aafb0d43ab71f1d8f786122c19afb37d25532aea169f414f8881: gzip
compressed data, original size modulo 2^32 1454592
```

```
manifest.json:    JSON data
version:          ASCII text
```

Is the Image Manifest the same as the OCI one?

```
{
  "schemaVersion": 2,
  "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
  "config": {
    "mediaType": "application/vnd.docker.container.image.v1+json",
    "size": 1456,
    "digest": "sha256:42b...6c0"
  },
  "layers": [
    {
      "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
      "size": 766708,
      "digest": "sha256:8ec...881"
    }
  ]
}
```

Indeed! The manifest.json contains one field called config and another one called layers. Due to the index.json absence, the manifest now has a permanent name instead of using its digest. In essence, it is now the root of the image pyramid.

3.3 Extending a Container Image

In this section we're taking advantage of the knowledge we gained in the previous chapters regarding container images and we present a way to extend an OCI image. We dived deeper into what a container image is and what hides behind a layer. We discovered the purpose of the manifest and the configuration file and the relationship between them. We realized that the JSON documents that compose the final container image are considered to be immutable and changing them means creating a new derived image. This is because the computed ID of the corresponding file would change and as a consequence the final ImageID.

Consequently, manipulating these documents results in the creation of a whole new image. As we saw in Figure 3.6, a slightly different manifest file can point to different layers and configuration file. In addition, we already know what a layer is. Therefore, we could begin by constructing a new - from zero - layer, that hypothetically includes an app's dependencies. Then, starting from a base image, we will edit its manifest and configuration files appropriately in order to integrate our fresh layer in the base image. We could say that we simulate the addition of a new instruction in a Dockerfile, but without by any means using Docker and its restriction to run only in privileged environments.

The Algorithm

First of all, we need to craft our new layer that will be placed on top of a base image. Remember that a layer that complies with the OCI image specification is a compressed tar archive. However, before compressing it we should first compute the digest of the tar archive - representing the difference between itself and the parent layer - in order to be appended in the configuration file. Then, the compressed layer's metadata will be integrated in the manifest file producing the new image. Below, we present the algorithm that shows how to extend a base image in steps.

1. Download a base image using Docker, Skopeo or any other tool that ships images from a source registry to a destination registry.

2. Decide the contents to include in the image and build the corresponding tar archive

   (a) Produce the tar archive composed of the layer's contents

   (b) Compute the sha256 digest of the uncompressed layer contents

3. Edit the configuration file

   (a) Find the configuration's filename in the manifest file

   (b) Append to the configuration file the metadata of the uncompressed layer. That is, add the digest of the uncompressed layer to the list of rootfs.diff_ids found in the file describing the differences between the layers.

   (c) Calculate the updated configuration's metadata (size & checksum) and rename it to its new digest

4. Compress the layer tar archive and compute the digest and the size of the compressed layer contents

5. Edit the manifest file

   (a) Append the new layer's metadata to the manifest's layers list

   (b) Modify the config field with the new metadata computed in the step 3c

   (c) Compute the new manifest's metadata (size & checksum) and rename it to its new digest

6. Edit the index file

   (a) Amend the manifest field to contain the up to date manifest's metadata in order to point to the right file

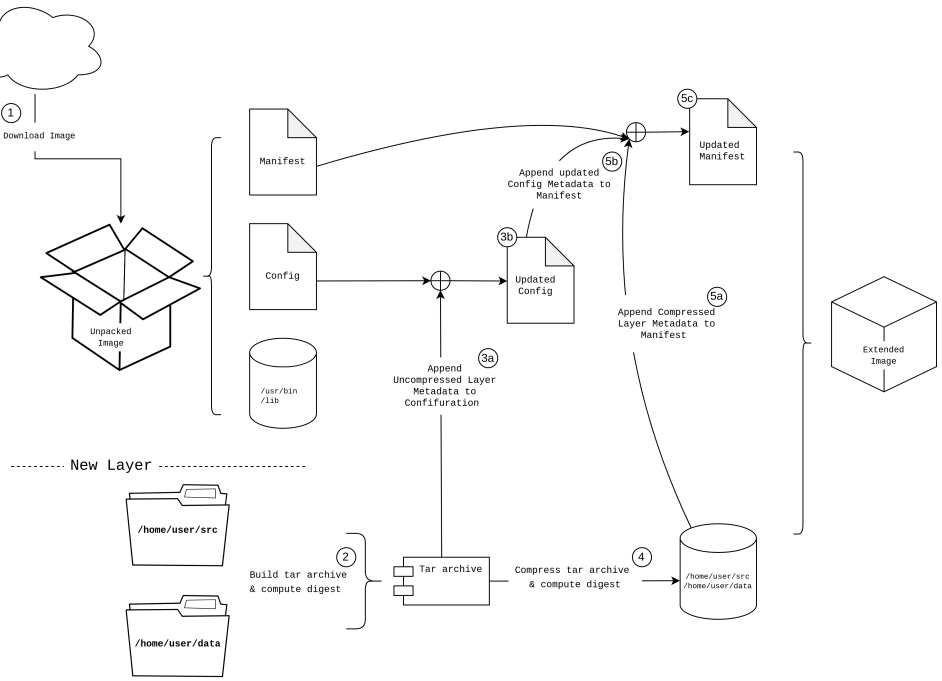


Figure 3.8: *Visualization of the algorithm describing how to extend a base image*

Example

The following example demonstrates the extend image algorithm. It illustrates the way to add some contents to a base image. Eventually, the changes will be noticeable when running a container with the new image.

NOTE: Intentionally for clearness and less noise most of the hash digests are stripped. We keep only the first and last 3 characters, separating them with *ellipsis points*.

1. Download the 'busybox' base image using skopeo

```
$ skopeo copy docker://busybox:latest oci:busybox:latest
Getting image source signatures
Copying blob 50e8d59317eb done
Copying config 4ff913783d done
Writing manifest to image destination
Storing signatures
$ tree busybox/
 busybox/
 ├ blobs
 │  └── sha256
 │      ├── 2a22f49c2fb14bae16fb907f6f36a9583a1f38f0044647077883221c7fa3645c
 │      ├── 4ff913783dfa58b73c1234c3111419ad7da07bd1f58462334a631bc1bb6d2417
 │      └── 50e8d59317eb665383b2ef4d9434aeaa394dcd6f54b96bb7810fdde583e9c2d1
 ├── index.json
 └── oci-layout
2 directories, 5 files
```

At this time the above output should be recognizable and familiar to us. Running a container with the above image will show its current filesystem, making it easier to compare it with the final one, after extending the image. Podman [54] will help us with this as a container runtime with almost no differences with Docker.

```
$ podman run --name busybox -it oci:busybox
Getting image source signatures
Copying blob 50e8d59317eb done
Copying config 4ff913783d done
Writing manifest to image destination
Storing signatures
/ # ls
bin   dev   etc   home  proc  root  run   sys   tmp   usr   var
```

2. Decide the contents of the layer and build the corresponding tar archive

   (a) For simplicity, our layer will contain one text file that includes some characters

```
$ mkdir my-layer
$ cd my-layer/
my-layer$ touch requirements.txt
my-layer$ echo "everything" > requirements.txt
my-layer$ cat requirements.txt
everything
```

(b) Produce the tar archive comprising the layer's contents. For this task, we will use *tar* linux archive utility [55]. The *'-c'* flag instructs tar to create the relevant archive, *'-v'* flag is related to output's verbosity and *'-f'* is used to define the output tar file.

```
my-layer$ tar -cvf my-layer.tar requirements.txt
requirements.txt
my-layer$ tar -tvf my-layer.tar
-rw-rw-r-- pangiann/pangiann 11 2022-05-09 22:46 requirements.txt
```

(c) Compute the sha256 digest of the uncompressed layer contents. Notice that we're still making use of *sha256sum* command.

```
my-layer$ sha256sum my-layer.tar | cut -d " " -f 1
4fc3a83a9c49b0ac374eeccee707a1459c9090524f149b23c2d119a574b4e130
```

3. Modify appropriately the configuration file

(a) Retrieve the manifest digest from the index file to acquire the manifest filename. Remember that manifest's name is its hash digest. That's why hashes act as pointers. *jq* is our JSON processor friend. It takes the first element from the manifests list and then fetches the digest field.

```
$ cd busybox/
busybox$ jq '.manifests[0].digest' index.json
"sha256:2a2...45c"
```

(b) Retrieve the configuration digest from the manifest file to acquire the configuration filename (we did the same for the manifest), albeit only 2 JSON documents exist (manifest & config file) under the blobs/sha256/ directory, so the process of elimination would be more than enough.

```
busybox$ cd blobs/sha256
busybox/blobs/sha256$ jq '.config.digest' \
> 2a22f49c2fb14bae16fb907f6f36a9583a1f38f0044647077883221c7fa3645c
"sha256:4ff...417"
```

(c) Edit the configuration file by appending the digest of the uncompressed layer tar archive under the rootfs.diff_ids array. The relevant digest is: 4fc3a83a9c49b0ac374eeccee707a1459c9090524f149b23c2d119a574b4e130. The updated **rootfs** field of the configuration file is:

```
"rootfs": {
    "type": "layers",
    "diff_ids": [
      "sha256:eb6...965",
      "sha256:4fc...130"
    ]
}
```

(d) Compute the new **(i) checksum** and **(ii) size** of the configuration file, since we modified it, and rename it accordingly. The size can be found with the *'ls -la'* command [56]. To rename the file we use the well-known *mv* command [57].

```
busybox/blobs/sha256$ sha256sum \
> 4ff913783dfa58b73c1234c3111419ad7da07bd1f58462334a631bc1bb6d2417 \
> | cut -d " " -f 1
7b59ae3ca33c5f7bf85b1983ff8ebb65fc7dcf063df4b0be15f9bc187e266bd0
busybox/blobs/sha256$ ls -la 4ff...417
-rw-r--r-- 1 pangiann pangiann 650 Μαϊ   9 23:44
busybox/blobs/sha256$ mv 4ff...417 7b5...bd0
```

4. Compress the layer tar archive and compute the digest and the size of the compressed layer contents. For the compress operation *gzip* linux command is used [58].

   (a) Move back to the layer's folder, compress the layer and compute its digest and size

   ```
   my-layer$ gzip my-layer.tar
   my-layer$ sha256sum my-layer.tar.gz  | cut -d " " -f 1
   82d377b6dfbf1af3f90545069345dfe94884cfa3ba3b3a60bc4faf6788916ebb
   my-layer$ ls -la my-layer.tar.gz
   -rw-rw-r-- 1 pangiann pangiann 156 Μαϊ   9 22:47 my-layer.tar.gz
   ```

   (b) Finally, rename the compressed tar archive to its computed digest in order to act in accordance with the image specification.

   ```
   my-layer$ mv my-layer.tar.gz 82d...ebb
   ```

   (c) Move the new layer under the busybox/blobs/sha256 directory

   ```
   my-layer$ mv 82d...ebb busybox/blobs/sha256/
   ```

5. Edit the manifest file

   (a) Append the new layer under the **layers** field with all the required metadata we computed earlier (size and digest). The mediaType will be *application/vnd.oci.image.layer.v1.tar+gzip*.

   (b) Edit the **config** field by updating the digest and the size of it with the ones we found in step 3d.

   (c) Compute the new checksum and size of the manifest's contents and rename it accordingly.

```
busybox/blobs/sha256$ sha256sum \
> 2a22f49c2fb14bae16fb907f6f36a9583a1f38f0044647077883221c7fa3645c \
> | cut -d " " -f 1
797a89b2ce3b2dff2d2aaf8a87de901fefea837e6b63294c6ae0749844e1059f
busybox/blobs/sha56$ mv 2a2...45c 797...59f
busybox/blobs/sha256$ ls -la 797...59f
-rw-r--r-- 1 pangiann pangiann 502 Μαϊ  10 00:17
```

The new manifest will be very similar to this:

```
{
  "schemaVersion": 2,
  "config": {
    "mediaType": "application/vnd.oci.image.config.v1+json",
    "digest": "sha256:7b5...6bd0",
    "size": 650
  },
  "layers": [
    {
      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
      "digest": "sha256:50e...2d1",
      "size": 772812
    },
    {
      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
      "digest": "sha256:82d...ebb",
      "size": 156
    }
  ]
}
```

6. Finally, edit the index file to point to the right manifest file with the correct size. The final index will be:

```
{
  "schemaVersion": 2,
  "manifests": [
    {
      "mediaType": "application/vnd.oci.image.manifest.v1+json",
      "digest": "sha256:797...59f",
      "size": 502,
      "annotations": {
        "org.opencontainers.image.ref.name": "latest"
      }
    }
  ]
}
```

It's time to run our new extended image:

```
$ podman run --name busybox-extended -it oci:busybox
Getting image source signatures
Copying blob 50e8d59317eb skipped: already exists
Copying blob 82d377b6dfbf done
Copying config 7b59ae3ca3 done
Writing manifest to image destination
Storing signatures
/ # ls
bin etc proc root sys usr dev home requirements.txt run tmp var
/ # cat requirements.txt
everything
```

We witness the existence of the new layer, that includes the requirements.txt file, in the container. This is one of the most fulfilling experiences of this diploma thesis. In the next chapter we are going to see how we automated this process to develop a build image mechanism in unprivileged environments.

3.4 The Design of Build Image Mechanism

In the previous chapters we explored Linux containers. We inspected the inner workings of a container like chroot, namespaces and cgroups. We realized that a container is

nothing more than a linux process with fancy tricks for isolation and protection leveraging various Linux kernel features. Then, we analyzed who decides the contents of a container and how the final filesystem is determined. A container image composed of a number of layers when unpacked assembles a runtime bundle which then is used to fire up the container. By the end of the previous chapter we realized the magnificence and great value of image layers. Being able to share them between images or extend and shrink an image by adding/removing layers demonstrates the ease of use, scalability and efficiency of the world of containers.

Definitely the process of manually manipulating a container image aiming to extend it is rather cumbersome. This is why container runtimes like Docker exist, correct? Yes and No! Docker, Podman or any other container runtime demands understanding of containers, images and registry concepts. Writing a Dockerfile means that you're comfortable and familiar with Docker. How about an ML engineer or Data scientist that wants to deploy in production their model, or serve it? Why bother building images with all of the libraries, code, data and complex dependencies? What happens in case of a mistake or bug in the code? You must fix the bug and rebuild the image with the appropriate changes and then another bug and again rebuild and the loop never ends.

Apart from that, one major disadvantage of Docker is the hard requirement of root privileges to run a container and build images. In fact, Docker forces users to communicate with the docker daemon. Therefore, in case someone wants to run containers inside a container, Docker is not a direct option. And why is not? Because, safety plays a big role when it comes to Kubernetes Clusters and production environements. No one wants a container that runs with root privileges. Our mechanism does its work in user space mode without daemons running behind the scenes wasting also CPU resources.

In this chapter we present a Python SDK that allows users to extend a base image with any sort of files or folders. Using this feature, users will be able to seamlessly build Docker images on the fly in unprivileged environments. This library stands as the first step to provide complete transparency and automation in the process of deploying machine learning workflows.

The mechanism is not designed necessarily to be used directly for the end user. In the next chapters we're going to see how it will be integrated to the well-known Kale (see more in section 2.9) tool in order to build with the click of a button ML workflows that

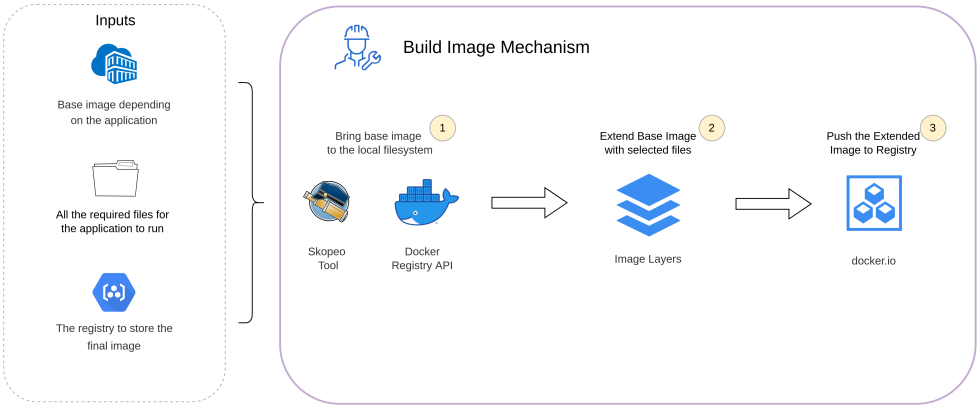


Figure 3.9: *High level visualization of build image mechanism*

will be deployed on Kubernetes. Nevertheless, its intuitiveness allows someone to build and extend images of their choice, providing the minimum required information related to images and containers. The mechanism's responsibility is to extend a base image with a new layer, composed of a bunch of files/folders. The steps it follows are:

1. Retrieves the image from a source registry to the local filesystem
2. Builds a tarball encompassing files and folders like source code, installed libraries, data and complex dependencies
3. Transforms the tarball into an image layer and appends it on top of the base image
4. Finally, pushes the new image to a registry that a container runtime can access

For steps (1) and (4) the mechanism leverages the capabilities of **Skopeo** around image transportation from a source to a destination.

The Computer Scientist as the person that develops, tests and eventually deploys their application will do the following:

1. They develop their application consisting of installed libraries, complex dependencies such as data and code
2. They choose all the required files for their application to run
3. They call *build_image* providing only a base image name and the files of their desire to containerize their application

Let's suppose that we are developing a fabulous Python app. Our base image will probably be a Python image. Then, our app consists of:

- Our source code which lives under $CWD/src directory
- Our data living under $CWD/data directory and
- Our installed python libraries living under $HOME/.local directory

Our goal is to build a new image based on the Python one that will include all of the above necessary files to run our application. To do so we call the *build_image* function from our library:

```
>>> build_image(base_image="python:latest",
. . .           include_paths=["/home/user/src", "/home/user/data",
. . .                          "/home/user/.local"],
. . .           dst_image="docker.io/pangiann/fabulous-app:v1")
'docker.io/pangiann/fabulous-app:v1@<digest>'
```

This is all we need to do to build a new ready-to-use image for our application and push it to a registry to make it accessible publicly. What if we forgot to add a crucial file? This is not a problem for the build image mechanism:

```
>>> build_image(base_image="docker.io/pangiann/fabulous:v1",
. . .           include_paths=["/home/user/icons"],
. . .           dst_image="docker.io/pangiann/fabulous-app:v2")
'docker.io/pangiann/fabulous-app:v2@<digest>'
```

Note that Skopeo searches in specific places to retrieve any authorization and certification files to authenticate us to the selected registry. For the authentication file the first path is *$XDG_RUNTIME_DIR/containers/auth.json*, which is set using skopeo login. If the authorization state is not found there, *$HOME/.docker/config.json* is checked, which is set using docker login. As you may imagine, this doesn't meet everyone's needs. This is why the mechanism offers the choice to provide the authentication and certification files for both the source and the destination registry.

```
>>> build_image(base_image="docker.io/pangiann/fabulous:v1",
...             include_paths=["/home/user/icons"],
...             dst_image="docker.io/pangiann/fabulous-app:v2",
...             dst_auth_file="/user/kale/auth.json")
'docker.io/pangiann/fabulous-app:v2@<digest>'
```

The function's signature is:

```
def build_image(base_image: str,
                include_paths: List[Union[str, Tuple[str, str]]],
                dst_image: str,
                src_auth_file: str = None,
                src_cert_dir: str = None,
                dst_auth_file: str = None,
                dst_cert_dir: str = None) -> str:
```

The arguments are:

- *base_image(str)*: A valid base image with format: *<registry>/<name>:<tag>@digest*. Digest is optional and the tag defaults to latest. It is especially easy to start by pulling an image from a public repository like docker hub.
- *include_paths(List[Union[str, Tuple[str, str]]])*: a List of paths or/and tuples of a src and a dst path
- *dst_image(str)*: A valid name for the new image with format: *<registry>/<name>:<tag>*
- *src_auth_file(str)*: A path to the authentication file to connect to the src registry (optional).
- *src_cert_dir(str)*: A path to the certification file for a secure trusted connection with the src registry (optional).
- *dst_auth_file(str)*: A path to the authentication file to connect to the dst registry (optional).
- *dst_cert_dir(str)*: A path to the certification file for a secure trusted connection with the dst registry (optional).

In case we have brought a base image in our local filesystem and we want to extend it like we did in section 3.3, the library offers this choice too with the *extend_image* function which returns a *DockerImage* object that mirrors a Docker Image in an unpacked format with various image operations included.

```
>>> new_image = extend_image(base_image_path="/home/user/python_image",
...                          include_paths=["/home/user/src",
...                                         "/home/user/.local"])
>>> print(new_image.image_digest)
'sha256:<digest>'
```

The *extend_image()* function has the below definition:

```
def extend_image(base_image_path: str,
                 include_paths: List[Union[str, Tuple[str, str]]]
                 ):
```

The arguments are:

- *base_image_path(str)*: A string representing the path to a valid v2_2 Docker image.
    - *path* can be either: (i) An absolute path or (ii) a relative path.
- *include_paths(list[str | tuple(str, str)])*: A list of source and destination paths to declare which files/folders from the local filesystem need to be copied into the new layer and at which location.
    - *source* can be either an absolute or relative path
    - *destination* can be either:
        * *Empty destination*: the source absolute path will be added as is in the new layer
        * *An absolute destination path*: the folder/file pointed by the source path will be added under the destination path

3.5 KFP Components

Kubeflow is a free and open-source project designed to make running machine learning workflows on Kubernetes clusters simpler and more coordinated. This is a cloud-native framework for employing machine learning in containerized environments in Kubernetes. Kubeflow's integration with and extension of Kubernetes has become seamless and Kubeflow has been designed to run everywhere Kubernetes runs [12]. At its core, Kubeflow offers an end-to-end ML stack orchestration toolkit to build on K8s as a way to deploy, scale and manage complex systems. Data scientists and engineers are now able to develop a complete pipeline composed of segmented steps. These segmented steps in Kubeflow are loosely coupled components of an ML pipeline.

3.5.1 What is a Kubeflow Pipeline Component?

A Kubeflow pipeline is a portable and scalable definition of a machine learning (ML) workflow, based on containers. A pipeline is composed of a set of input parameters and a list of the steps in this workflow. Each step in a pipeline is an instance of a component. When designing a pipeline, we should consider how to split the ML workflow into pipeline components. Each component should have a single responsibility. Having a single responsibility makes it easier to test and reuse a component. For example, a component that loads data can be reused for similar tasks that load data [20]. This showcases the significance and importance of creating standalone, reusable and shareable components for the ML community.

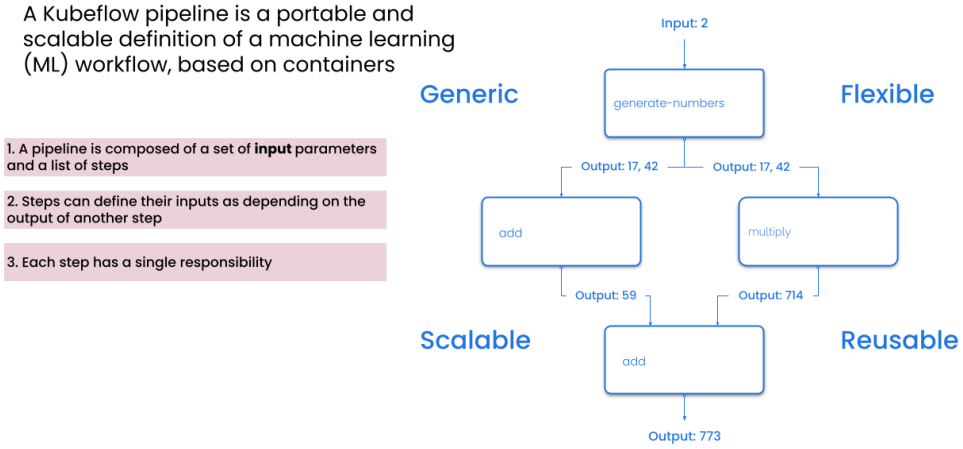


Figure 3.10: *A Kubeflow Pipeline example in a form of a graph with inputs and ouputs*

A **Kubeflow Pipelines component** is a containerized application - self-contained set of code - that performs one step in an ML workflow [21]. It is nothing more than a YAML specification file that is composed of:

- The component code, which implements the logic needed to perform a step in the ML workflow
- A component specification which defines the following:
  - *The component's metadata*: its name and description
  - *The component's interface*: the component's **inputs** and **outputs**
  - *The component's implementation*: the **Docker container image** to run, how to pass inputs to your component code, and how to get the component's outputs.

Ultimately, pipelines are composed of component instances, also called steps. Steps can define their inputs as depending on the output of another step. The dependencies between steps define the pipeline workflow graph. To build a pipeline, means to build individually each component that will construct the final pipeline. Here is a simple example of a Kubeflow Pipeline component:

```
name: Get Lines
description: Gets the specified number of lines from the input file.

inputs:
- {name: Input 1, type: String, description: 'Data for input 1'}
- {name: Parameter 1, type: Integer, default: '100', description: 'Number of lines to copy'}

outputs:
- {name: Output 1, type: String, description: 'Output 1 data.'}

implementation:
  container:
    image: gcr.io/my-org/my-image@sha256:a172..752f
    # command is a list of strings (command-line arguments).
    # The YAML language has two syntaxes for lists and you can use either of them.
    # Here we use the "flow syntax" - comma-separated strings inside square brackets.
```

```
command: [
  python3,
  # Path of the program inside the container
  /pipelines/component/src/program.py,
  --input1-path,
  {inputPath: Input 1},
  --param1,
  {inputValue: Parameter 1},
  --output1-path,
  {outputPath: Output 1},
]
```

3.5.2 Build Components and Pipelines with Kubeflow

A component is analogous to a function, in that it has a name, parameters, return values, and a body. When we run a pipeline, one or more Kubernetes Pods for each component (step of the pipeline) are launched. The Pods themselves start Docker containers which in turn start our program. To make this happen, KFP uses the *Argo Workflow Executor* underneath.

KFP provides a way to convert a Python function into a reusable and shareable component. There are two options KFP offers to build a component from a Python based function. The provided API includes the *create_component_from_func()*. By default, it will use a Python base image and extend it with the user's code and the bare minimum argument parser in order to be able to read inputs and save the corresponding outputs. Containers make use of an entrypoint to start their execution. Therefore, KFP auto generates an argument parser to pass:

(A) the inputs through the entrypoint and

(B) the path to the file where the outputs will be stored in order for the downstream steps to access them.

This approach suffers from a lot of implications and restrictions. The Python function must be standalone, that is:

1. It should not use any code declared outside of the function definition

2. Import statements must be added inside the function
3. Helper functions must be defined inside the function

All of the above force Kubeflow users to follow the second option which is to build themselves the Docker image of the component, including all the complex dependencies required for it to run. Immediately, though, this approach violates the "rule" that Data Scientists should not be occupied with Container/Kubernetes operations. This process also makes it a lot harder to debug, modify your code and quickly deploy your model as we have to build Docker Images iteratively with the updated contents. A simple example of the KFP dsl is shown below:

```
# This is a simple function that will be executed in a KF Pipeline step.
# Notice that the inputs and outputs are simple Python values, so we
# annotate them with their type (float).
def add(a: float, b: float) -> float:
    """Calculate sum of two arguments."""
    return a + b

# This is the reproducible component that is created out of the
# above function. We'll use for both of the KF Pipeline steps.
add_op = create_component_from_func(add)


@dsl.pipeline(
    name='Calculation pipeline',
    description='An example pipeline that performs arithmetic calculations.'
)
def calc_pipeline(a='1'):
    """The Pipeline function."""
    first_task = add_op(a, 4)

    # Since the 'add()' function returns just one single value as output,
    # we can access the output of the 1st task through `first_task.output`.
    second_task = add_op(first_task.output, 4)

Compiler().compile(calc_pipeline, 'workflow.yaml')
```

As you will see our thesis focuses on addressing this issue using our way to build images on the fly (see section 3.4). Apart from this, KFP requires a deep understanding of

the provided SDK and its components (i.e. ContainerOp, Compiler, Client etc.) and a lot of changes in the initial code in order to develop the pipeline graph, compile it and eventually run it. Meanwhile, most of the data scientists use a JupyterLab working environment which makes it impossible to directly convert the notebook into a pipeline. In the next section we'll see how Kale stands on top of KFP facilitating the creation and execution of pipelines, converting notebooks to pipelines with the click of a button.

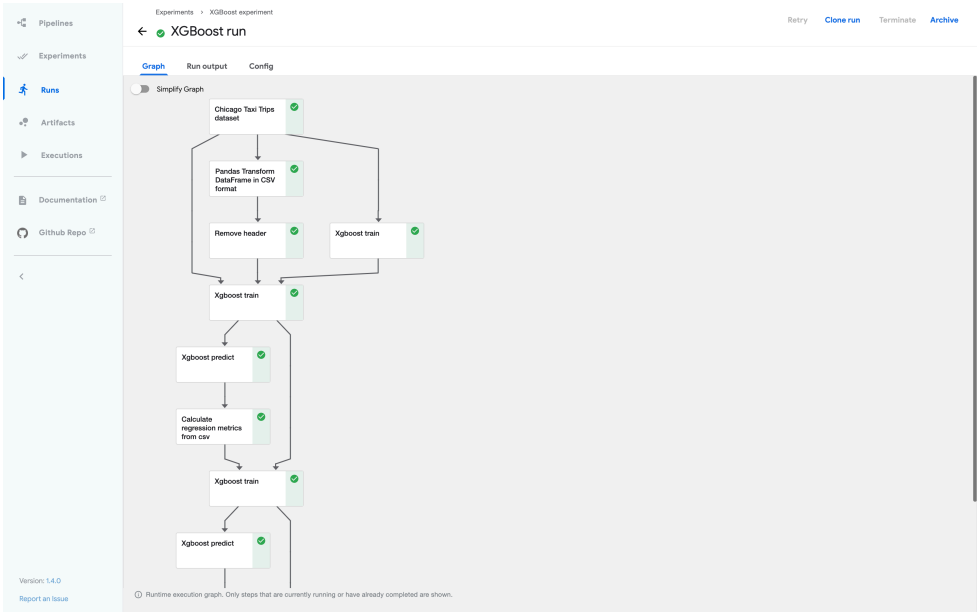


Figure 3.11: *A Pipeline graph representation in the Kubeflow UI*

When Kubeflow Pipelines executes a component, a container image is started in a K8s Pod. To simplify, we assume that the component contains a function that receives some inputs and produces an output. The component's inputs must be passed in as command-line arguments. The entrypoint ideally should be:

```
python3 script.py
```

where inside the script is our component's function. Well, clearly there must be a way to pass the inputs of the function and the path to store the outputs. Here is a simple approach. The component's code is extended with an argument parser containing flags like *--input* and *--output-path*. Output will always be saved under a local file.

```
# Defining and parsing the command-line arguments
parser = argparse.ArgumentParser(description='My program description')
# Paths must be passed in, not hardcoded
```

```
parser.add_argument('--input1', type=float,
  help='Input data')
parser.add_argument('--input2', type=float,
  help='Input data')
parser.add_argument('--output-path', type=str,
  help='Path of the local file where the Output data should be written.')
args = parser.parse_args()
```

We encapsulate this logic in the container's entrypoint:

```
python3 script.py --input1 17.0 --input2 42.0 --output_path <path_to_file>
```

The argument parser is easy to implement but also a monotonous process. Let alone the fact that we need to change the parser every time we edit the function's arguments.

Recall that for Kubeflow Pipelines to run the component, the component must be packaged as a Docker container image and published to a container registry that the Kubernetes cluster can access. At this point things are starting to become more complicated and Kubeflow is not an ally. In brief, the simplest approach is to create a Dockerfile for the container. A Dockerfile specifies:

1. Any dependencies that need to be installed for our code to run.
2. Files to copy into the container, such as the runnable code for this component.

```
FROM python:3.7
RUN python3 -m pip install keras
COPY ./src /pipelines/component/src
```

Afterwards, we use Docker to build the container image and push it to a container registry that our Kubernetes cluster can access. Indirectly, this requires also to have already deployed a registry. To convert the code and the container image into a Kubeflow Pipelines Component we need to define the component's implementation and interface.

For the **implementation** the container image and the entrypoint are required:

```
implementation:
  container:
    image: gcr.io/my-org/my-image@sha256:a172..752f
```

```
    # command is a list of strings (command-line arguments).
    command: [
      python3,
      # Path of the program inside the container
      /pipelines/component/src/script.py,
      --input1,
      {inputValue: Input 1},
      --input2,
      {inputValue: Input 2},
      --output1-path,
      {outputPath: Output 1},
    ]
```

Instead of the actual values, which are unknown before runtime, we use the KFP's input/output placeholders. At runtime, placeholders are switched with the real values.

There are three types of input/output placeholders [21]:

- *inputValue:* <*input-name*>: This placeholder is replaced with the value of the specified input. This is useful for small pieces of input data, such as numbers or small strings.
- *inputPath:* <*input-name*>: This placeholder is replaced with the path to this input as a file. Your component can read the contents of that input at that path during the pipeline run.
- *outputPath:* <*output-name*>: This placeholder is replaced with the path where your program writes this output's data. This lets the Kubeflow Pipelines system read the contents of the file and store it as the value of the specified output.

We are now ready to define the component's interface. The interface demonstrates the inputs and outputs of the component. Each input or output list contains a name, a description, the type and a default value.

```
inputs:
- {name: Input 1, type: Float, description: 'Data for input 1'}
- {name: Input 2, type: Float, description: 'Data for input 2'}

outputs:
- {name: Output 1, type: String, description: 'Output 1 data.'}
```

In the end, the component requires a name and description that serve as metadata. The next step is to use our component in a pipeline. The easiest way is the:

- load_component_from_file()

method. The object returned is a factory function that can be used in the final pipeline function:

```
create_step_add = comp.load_component_from_file("component.yaml")
def my_pipeline():
    create_step_add(17, 42)

client = kfp.Client()
client.create_run_from_pipeline_func(my_pipeline, arguments={})
```

3.5.3 Argo Workflow Executor

The above component definition is converted into an Argo Workflow. An Argo Workflow is essentially a Kubernetes CustomResource (CR) that describes the entire DAG-graph and execution of a KubeFlow Pipeline. It defines the workflow where each step in it is a container. It contains the code that each Pipeline step will run, and also specifies the dependencies and data passing between steps. The KFP SDK is capable of producing the corresponding Argo Workflow starting from a Pipeline. The pipeline is basically compiled to an Argo Workflow which then is applied to a Kubernetes cluster and the Pipeline starts to run.

The Argo specification includes a spec field that contains all of the important and necessary information. In particular, it is made of 3 main parts:

- Entrypoint
- Templates
- Arguments

The Templates field is used to describe the steps of the workflow, which in our case is the pipeline. Each element contains a name, some metadata, a container field that

specifies the base image, the entrypoint, its arguments and an inputs/outputs field. One of the templates element, called **dag**, is used to describe the dependencies between the steps that depict the final graph. In other words, in a DAG, we list all our tasks and set which other tasks must complete before a particular task can begin. Tasks without any dependencies will be run immediately [59]. In this example A runs first. Once it is completed, B and C will run in parallel and once they both complete, D will run:

```
- name: diamond
dag:
  tasks:
  - name: A
    template: echo
  - name: B
    dependencies: [A]
    template: echo
  - name: C
    dependencies: [A]
    template: echo
  - name: D
    dependencies: [B, C]
    template: echo
```

The entrypoint field defines what the ”main” function will be – that is, the template that will be executed first. For the Kubeflow pipeline we provided earlier the produced Argo subset of the workflow for the first step call is:

```
- name: add
container:
  args: [--input1, '{{inputs.parameters.a}}', --input2, '4', '--output-path',
↪  /tmp/outputs/Output/data]
  command:
  - python3
  - script.py
  inputs:
    parameters:
    - {name: a}
  outputs:
    parameters:
    - name: add-Output
```

```
      valueFrom: {path: /tmp/outputs/Output/data}
  ...
```

Notice how the input and output **placeholders** are now replaced with either plain values or with fields of the Argo workflow that will be filled with values when the corresponding task has been completed.

User's experience to produce Kubeflow Pipelines feels rather cumbersome. Surprisingly, the situation deteriorates. What should we do in case of a bug? Update our code, change the argument parser, build a new Docker image and change the component's implementation. In Machine Learning, which is our case study, this may be sufficient when models are rarely changed or trained. In practice, models often break when they are deployed in the real world. The models fail to adapt to changes in the dynamics of the environment, or changes in the data that describes the environment, thus we must continuously iterate on our code and data.

3.6 Run Kubeflow Pipelines with Kale & Rok

Needless to say, we need a way to easily generate production-ready ML pipelines with "point and click" operations and roll back instantly to any pipeline step for rapid debugging and collaboration. Kale with Rok made the dream come true by snapshotting the user's working environment and attaching it in the Pod's where the steps will run. Unfortunately, in order to create proper KFP components, relying on Rok snapshots is not a direct option, since all the required code needs to be packaged between the base docker image and the container's entrypoint.

3.6.1 Kale Jupyter & SDK

The main idea behind Kale is to exploit the JSON structure of Notebooks to annotate them, both at the Notebook level (Notebook metadata) and at the single Cell level (Cell metadata) [14]. These annotations allow us to:

1. Assign code cells to specific pipeline components
2. Merge together multiple cells into a single pipeline component

3. Define the execution dependencies between them

All in all, Kale makes it fairly easy to conceptualize Jupyter Notebook cells as steps of an ML workflow. Notice that steps/Jupyter cells without dependencies will run in parallel in Kubernetes Pods, maximizing the Pipeline's performance.

For example, the image below shows multiple cells that are part of the same pipeline step. They have the same blue color and they depend on a previous pipeline step. The only thing one needs to do to define pipeline steps is edit the corresponding cell, name the pipeline step as they desire, and define its dependencies. A dependency can be any of the pipeline steps. Kale discovers all the defined pipeline steps automatically and presents a dropdown list with them to the user.

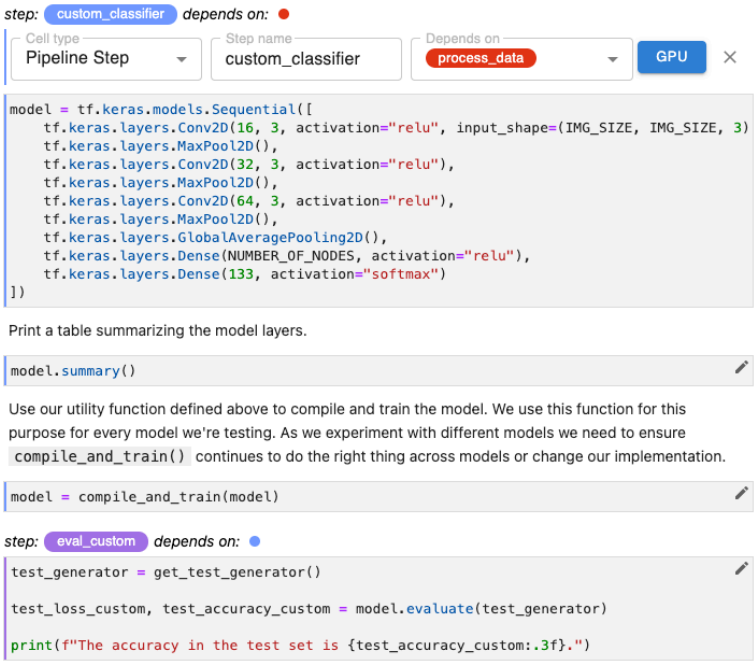


Figure 3.12: *Kale's JupyterLab extension lets us annotate cells as pipeline steps and define their dependencies.*

After annotating appropriately the Jupyter Cells we just need to hit Compile & Run. Now, Kale takes over and transforms our notebook, by converting it to a KFP pipeline. Also, because Kale integrates with Rok, it will take a snapshot of the notebook's volumes. Rok takes care of data versioning and thus allows us to reproduce the whole environment as it was when we clicked the COMPILE AND RUN button. This way, we have a time machine for our data and code, an exact versioned and reproducible point from where our pipeline starts, and our pipeline will run in an identical environment with the one we developed our code on, without needing to build new docker images. Keep in mind that this approach has also some implications and we will see in this thesis what our different approach offers in the end user, and how we can combine both mechanisms (ours and Rok's) to leverage them to the maximum and gain the best results.

We can undeniably and emphatically admit the value of Kale on Kubeflow Pipelines. In the previous section we realized that using directly Kubeflow Pipelines most of the times requires along with a deep understanding of the KFP's SDK, numerous changes in the initial source code and knowledge of containers, images and Kubernetes. Apart from this, data scientists are not able to convert a Jupyter Notebook to a Kubeflow Pipeline only with the KFP's SDK. Kale comes with great and significant solutions, making the ML engineer's life fairly easier, requiring only to describe the workflow to Kale and hit one button.

Apart from the JupyterLab extension Kale offers an SDK providing the simplest way to convert any repository of Python code into fully reproducible Kubeflow Pipelines runs without changing the source code [22].

3.6.2 Pipeline & Steps

Similarly to the JupyterLab extension, ML engineers need to describe to Kale what will be the steps (nodes) of the Pipeline and their dependencies (edges), or in other words the final graph. From the JupyterLab we saw that Kale provides us with cell annotations. From the SDK, Kale offers two function decorators: *step* and *pipeline*.

Step Class

The functions we want to be steps of the ML workflow we need to decorate them with the step decorator.

```
@step(name="data_loading")
def load(random_state):
    """Create a random dataset for binary classification."""
    rs = int(random_state)
    x, y = make_classification(random_state=rs)
    return x, y
```

In the above code snippet we define the dataset loading and classification as an ML workflow step, which is a pretty common tactic. Behind the scenes, Kale instantiates a Step object including vital information required to convert it into a KFP component that will be part of the final Kubeflow Pipeline. The *Step class* includes:

1. The step's function to run under a *do_run* attribute.
2. The step's metadata, its name and description
3. The step's interface, that is, the inputs and outputs of the step
4. The component's implementation, the Docker container image to run, the entrypoint and the CLI arguments.
5. The marshalling logic which wraps the Python function and implements Kale's data passing mechanism between steps, i.e. feed the outputs of one step to the inputs of a downstream step.

The above procedure reminds us a lot of the KFP's creation of components starting from a Python function. However, either with our build image mechanism (3.4) - already presented in this thesis - or Rok alongside Kale, we overcome the restrictions that KFP sets, i.e. imports and helper functions should be included in the step function. On the contrary, users are able to define complex data and code dependencies, that is, call functions from other files and use data living in their working space and filesystem without any implications or the frustration of building custom Docker Images.

KFP renders the Python function itself in the Argo's template and runs the germane code with a ' *python -c* "" ' entrypoint. Furthermore, it extends the source code with an argument parser that receives the inputs and outputs of the component. Kale differentiates from Kubeflow's approach. It manages to relocate the source file and its dependencies to the Pod where each step will be executed. On top of that, the entrypoint runs, under

the aegis of Kale, the specific step of the Python file. A Kale entrypoint would look like this:

```
python3 -m kale source.py --step step1 --in in1 17 --in in2 42
```

Note that Kale's CLI is configured to receive inputs and outputs with the respective flags and there is no need to extend the initial source file. Eventually, a special flow path is followed from Kale to execute a step.

Pipeline Class

In the JupyterLab extension the depends on cell annotation option shows the order in which steps will be executed. In the Kale SDK, a pipeline function decorated with the *pipeline* decorator contains the step decorated function calls that portray the final *pipeline graph*.

```
@pipeline(name="binary-classification", experiment="kale-tutorial")
def ml_pipeline(rs=42, iters=100):
    """Run the ML pipeline."""
    x, y = load(rs)
    x, x_test, y, y_test = split(x, y)
    train(x, x_test, y, iters)
```

The *load*, *split*, and *train* are all step decorated functions. Kale will automatically discover the order and dependencies between the steps and it will produce the relevant graph. Eventually, the pipeline function has an end goal to represent the final ML workflow and not to implement business logic.

The decorator instantiates a new Pipeline object. The Pipeline class lives under Kale's pipeline module and is used to define a Kale pipeline, its steps and all their dependencies. It extends the networkx's DiGraph [60] class for directed graphs with self-loops to exploit its underlying graph-related algorithms but also provides helper functions to work with Kale Step objects instead of standard networkx "nodes". This makes it simpler to access and traverse the steps of the pipeline and their attributes. Finally, it includes a *PipelineConfig* object that includes vital metadata for the Pipeline like the volumes that will be used, the working directory, the marshal directory that implements the data passing between steps and more.

Kale Domain Specific Language

The whole essence of Kale's DSL is to allow writing Python functions that describe the

architecture of a pipeline that, without a @pipeline decorator, can be run as-is, locally. The only operation required to turn it into a Pipeline object should be applying the @pipeline decorator. The DSL (domain specific language) currently allows the following Python statements to be used in a pipeline function:

1. Function calls with input parameters that are either pipeline parameters or other function call outputs.

```
def dsl(param”=”hello):
    step1(param)
    step2()
```

2. Assignments from (step) function calls. The assigned values are the step outputs. Multiple outputs can be retrieved with tuple assignments

```
def dsl(param=”Hello”):
    my_out = step_1(param)
    step_2(my_out)
```

3. If-statements with boolean conditions. These conditions must have just one comparison between pipeline parameters and constant values.

```
def dsl(param=”Hello”):
    res1 = step_1(param)
    if param == ”yes”:
        res2 = step_2(res1)
```

4. For-loops with a boolean loop condition

```
def dsl(param=”Hello”):
    res = generate()
    for x in res:
        squared(x)
```

Although the above may feel like a kind of restriction, this is not the case. Besides, you don't need more than the above to represent the ML workflow-graph. As we already mentioned, the pipeline function's main responsibility is to portray the final pipeline.

3.6.3 Pipeline Processing

After the successful development of our Python code, what's left is to instruct Kale to deploy and run our code as a KFP pipeline:

```
python3 -m kale pipeline_script.py --kfp
```

Notice that we don't run the *source.py* file, instead we start the Kale module and pass as a parameter the pipeline script. After instantiating the step objects - the step decorator is responsible for this - it is the pipeline's decorator turn. The first thing Kale does is to process the pipeline function. In particular, it validates the pipeline to check that it complies with the DSL's regulations. That is, it parses the source AST tree and checks that it contains only assignments, calls, if and for statements. Then, it configures the below:

1. Volumes that will be used during the pipeline's execution
2. Absolute working directory
3. Marshal directory where steps will store marshal data
4. The container image to be used in internal steps

Finally, it registers the steps in the pipeline. Recall that a Pipeline object inherits the *DX.graph class* [60]. Therefore, Kale discovers all the step calls and their dependencies and constructs the pertinent graph.

3.6.4 Pipeline Compilation

After the pipeline processing, Kale has all the necessary information to compile the Pipeline object into a KFP pipeline. Kale first creates a Pipeline object, via the *Python-Processor* that we described before, and then uses a Compiler object to convert it to a KFP pipeline. To do so,

1. It converts each step into a KFP component that includes the name, inputs/outputs and implementation of the component.
2. Then, it produces a KFP task, which is a subset of the Argo workflow represented as a Python object.
3. Finally, it unites all the KFP tasks and exports the final workflow.

Last but not least, Kale submits the YAML Argo workflow to run the Kubeflow pipelines in Kubernetes.

For the below step decorated function:

```
@step(name="add")
def add(num1: int, num2: int) -> int:
    """Add two integer numbers."""
    return num1 + num2
```

The part of the Argo workflow produced from the above step should be very similar to this:

```
- name: add
  container:
   args: [--in, a, '42', --in, b, '13', --out, out, /tmp/outputs/out/data]
   command: [python3, -u, -m, kale, /home/jovyan/script.py, --step, add,
↪   --marshal-path, /home/jovyan/.pipeline.kale.marshal.dir]
   env:
   - {name: PYTHONUNBUFFERED, value: '1'}
   image: gcr.io/my-org/my-component@sha256:a172...752
   securityContext: {runAsUser: 0}
   volumeMounts:
   - {mountPath: /home/jovyan/.pipeline.kale.marshal.dir,
     name: kale-marshal-volume}
   workingDir: /home/jovyan
```

Notice that all we did was to decorate our function, then Kale handles everything else for us.

3.6.5 Pipeline Execution

What may feel bizarre and absurd at first is that the step's execution requires the Kale library. This is subsequent to the fact that the whole source code is transferred to the step's location (reminder: each step of the workflow runs in a Kubernetes Pod). Therefore, Kale is required to run individually and in a special manner the corresponding step found in the source file. The entrypoint is the big bang of the step's execution:

```
python3 -u -m kale script.py --step add
--in a 42
--in b 13
--out out /tmp/outputs/out/data
```

It is clear that the source file does not run as is. Kale is invoked to run the step that appears in the source file. Input values are passed through the Kale's CLI --in flag. The output path is configured in the same way. Kale's main purpose is to wrap the runtime execution with Marshalling logic. More precisely, a Marshaller object is initialized which is responsible for passing data between steps. A new shared PVC known also as kale-marshal-volume is provisioned and mounted to each step. The volume is treated as a shared folder for the communication between steps in case of dependencies. The Marshaller requires:

1. Step's function
2. Runtime inputs inferred by the *--in* entrypoint argument
3. Runtime outputs inferred by the *--out* entrypoint argument
4. Marshal path where the marshal data will be stored

Before actually running the step, Kale loads all the required input values to run the step. Inputs can be either plain values or marshal data. For the former, Argo loads the input values from a predefined path and Kale deserializes them based on their type. For the latter, under the same predefined path Kale has already stored a marshal path from the step that produced the marshal data. This marshal path lives in the shared PVC where all steps can have access. For Argo the input value is just a path, however Kale detects it and goes to the corresponding file, reads the real more complex/marshal data and deserializes them in a special manner. Kale's marshalling mechanism includes various backends like numpy, pandas, sklearn and more, in order to store/load different complex object types.

Finally, Kale runs the function and stores the result either as a plain value under the Argo provided path or as marshal data under the marshal path and then saves the marshal path under the predefined Argo path, such that downstream steps can find them.

3.6.6 Data Passing between Pipeline steps

Marshalling is a mechanism to seamlessly pass data between steps. This system requires a shared folder where Kale can serialize and deserialize data. Kale uses a hidden folder as the shared marshalling location. Steps read and write their inputs/outputs in a shared space facilitating the data passing between them.

Kale transparently provisions a volume that will serve as the shared space between steps. In reality, Kale will always check if the working space environment is mounted under a Rok volume. In such a case, Kale uses the cloned Rok workspace volume for marshalling purposes too. However, using the workspace volume to marshal data can become problematic in case the pipeline steps need to pass large assets between them. This is why Kale, if asked explicitly, provisions a volume dedicated to data passing, deciding its size and mount location.

Kale provisions the new marshal volume prior to the pipeline's execution. Afterwards, it picks the first step from the pipeline and runs it. This is when the whole marshalling process takes place.

3.6.7 Typing System

In an earlier chapter, we discussed the Step class implementation. We observed that a step decorated function is not just a plain Python function. Both KFP's and our thesis purpose is to convert a function into a standalone and independent component. Kubeflow Pipelines does it already with serious implications. Our thesis strives to make the procedure as seamless as possible without setting any barriers, rules and restrictions. The inputs and outputs, along with their types, are the main characteristic of a component. The component's interface is inferred from the function's signature. This requirement obliges the step function's parameters to have type hints. Specifically, there are two options:

1. Use regular Python type-hints, such as int or even type-hints from Python's typing module, such as List. These type-hints declare that the step expects its inputs to be values (of the declared type), passed in the args of the component.

2. Use Kale's *MarshalData[mytype]* type-hint which declares that the step expects

this input to be a path to a marshalled object of type mytype which is a regular Python type. At the start of the step's execution, Kale's runtime will take over to unmarshal the object.

Consider the below function that receives two Integer inputs and produces one Integer output:

```
def add(a: int, b: int) -> int:
    '''Calculate sum of two arguments'''
    return a + b
```

The generated KFP component will be:

```
name: add

inputs:
- {name="a", type="Integer"}
- {name="b", type="Integer"}

outputs:
- {name="out", type="Integer"}

implementation:
  container:
    image: gcr.io/my-org/my-component@sha256:a172..752f
    command: [
      python3,
      -u,
      -m,
      kale,
      /home/jovyan/script.py::add
   ]
   args:
     - --in
     - a
     - {inputValue: a}
     - --in
     - b
     - {inputValue: b}
     - --out
```

```
    - out
    - {outputPath: out}
```

Undoubtedly, the inputs and outputs fields are an integral part of the component.

3.6.8 Rok Snapshots

Kale relies on Rok snapshots to run KFP pipelines. Rok provides an enterprise data management layer that makes it possible to instantly snapshot containers for local and offsite backups. Take immutable, group consistent snapshots of apps and keep these snapshots in a backup store, e.g. S3. Inevitably, Rok snapshots is the best choice for Kale in its effort to avoid KFP's implications and coercion to build Docker images for more complex pipelines.

Specifically, when we execute:

```
python3 -m kale kale_sdk.py --kfp
```

the following happens:

1. Kale, via Rok, takes a snapshot of the volumes (data + code) where we develop your code, i.e. a notebook server.
2. Kale compiles the Python code into a KFP pipeline and mounts clones of the snapshots to every step of the pipeline
3. Every pipeline step has an entrypoint similar to '*python3 -m kale kale_sdk.py ...*'. This means that the entrypoint expects the source file or notebook and all file's dependencies (i.e. installed libraries, data) to be present in the Pod's filesystem where the step will actually run.
4. In this way, we are not obligated to build images containing source and data related to our work. Rok manages to transfer everything using snapshots.
5. Eventually, taking snapshots is an essential requirement for Kale to run pipelines

Kale relying on snapshots proved to be a huge obstacle in our effort to:

- Build KFP reusable and standalone components
- Publish models as packaged Docker containers

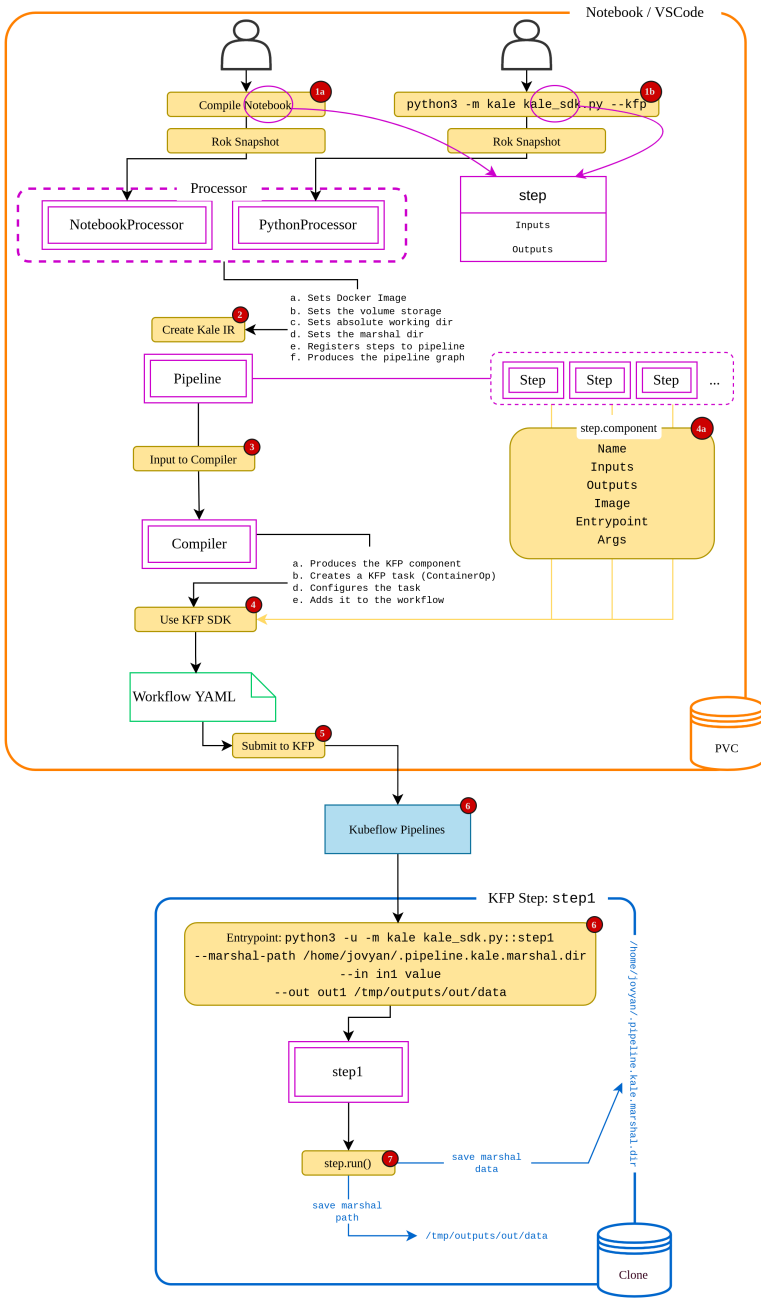


Figure 3.13: *Kale workflow execution path. Kale: 1) Snapshots the user's working environment with Rok 2) Processes the pipeline script 3) Compiles it into a Kubeflow Pipeline 4) Runs each step*

3.7 Kale Components & Build Image Library

For this reason we developed a mechanism to build images in unprivileged environments. For the purpose of this thesis, we extend the Kale SDK to be able to produce standalone, reproducible and shareable Kubeflow Component seamlessly. Until now, Rok snapshots the volumes of the data scientist's working environment, clones them and attaches them to each Pod where the steps will run. In this way, users won't get in trouble with Docker Images. Their environment (including code and data) is fully replicated to where each step will run. However, it is impossible to share a component to others not using the Rok storage system, or serve a model in demanding environments where softwares like Rok cannot be attached to (i.e. a car). This is where our build image mechanism comes into play and integrated with Kale achieves producing completely independent KFP components that can be run anywhere from anyone.

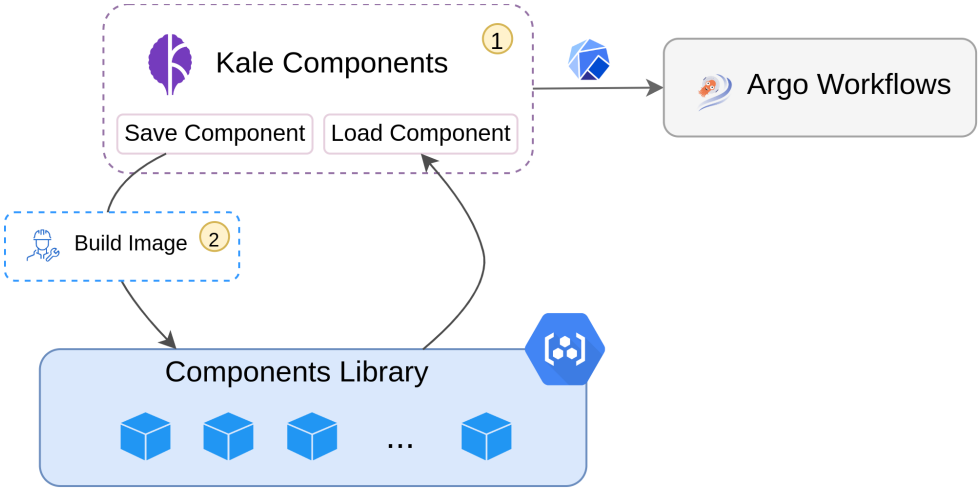


Figure 3.14: *Our contribution to Kale SDK: Components*

3.7.1 Run Kubeflow Pipelines with Kale & Build Image SDK

As the first step, we need to integrate the mechanism of building images on the fly into Kale and detach the latter's execution from Rok. Until now, Kale offered a way to compile a pipeline with Rok by snapshotting the user's workspace and attaching it to the Pods where steps will run. In this way, Kale managed to transfer the user's code and data to the step's execution environment. To disassociate Kale from Rok, steps require a Docker image that contains the necessary dependencies to run (i.e. the source code). As

we mentioned earlier, the Kubeflow component's implementation includes the Docker container image for its execution.

In particular, the image reflects the contents of a container, as the file system of the container is determined by it. Therefore, the component's filesystem and subsequently the new image, must contain the source file and all the required libraries, code and data dependent on this file. To achieve this, we extend Kale's SDK to compile and run user's code as KFP pipeline without the snapshot mechanism, but leveraging the build image library we developed in 3.4. Kale's responsibility is to build a new image for the workflow's steps doing its best to meet the minimum requirements for the pipeline to run. For this purpose, we amalgamate the build image mechanism we developed with Kale. What is awesome with this mechanism is that it doesn't require root privileges or a daemon. This is super important to keep our clusters secure and safe. Rootless containers are more secure for many reasons [61].

From the Data Scientist's perspective, they only need to develop their ML code with the appropriate step and pipeline decorators. Then, we introduce a new --build-image flag, that along with Kale will take care of all necessary procedures to convert their code into a KFP pipeline by building the necessary Docker imager for it.

```
@step(name="step1")
def add(x: int, y: int) -> int:
    return x + y


@pipeline(name="my-pipeline")
def my_pipeline(x = 42: int):
    res = add(x, 17)


if __name__ == "__main__":
    my_pipeline()
```

Specifically, from now on Kale now offers a new option to produce KFP components and run a pipeline starting from a Python function:

```
$python3 -m kale script.py --compile --build-image
```

1. Kale retrieves the base image of the notebook server where we develop our code.

2. By default Kale builds a new image, starting from the base Notebook server, doing its best to meet the minimum requirements to create and run the KFP pipeline. Thus, besides the source file Kale also includes in the component's filesystem:

    - Our code → Kale assumes that lives under CWD/src
    - Our data → Kale assumes that lives under CWD/data
    - Installed Python libraries → They live under $HOME/.local

There may be cases where Kale's default behavior does not fulfill our needs; we can use the *--build-path* flag to include in the final image the files/folders of our choice and overwrite Kale's default ones. For example, our source code and data are all under $HOME/fabulous-app. The new command will be:

```
$ python3 -m kale script.py --compile --build-image --build-path /home/user/
fabulous-app
```

Kale gives the image a name by default but we can always change this with the *--build-image-name* flag. The well-known proper format is <*name*>:<*tag*>. At this point, Kale will try to retrieve information related to the destination registry from some relevant environment variables. By default we set these env vars accordingly to force Kale use a private Docker registry to store the newly built image and let kubelet run a container (e.g. a step) with it. We can use our own registry by setting the *--registry-host* flag or changing the *KALE_REGISTRY_HOST* environment variable. When required by our repository we also have to define *--registry-auth-file* and *--registry-cert-file* paths to connect and securely communicate with the registry.

```
$ python3 -m kale  /home/jovyan/kale_sdk.py --kfp --build-image \
--build-image-name my-pipeline:v1 --registry-host gcr.io/my-registry/ \
--registry-auth-file /home/user/auth.json
```

All the above settings can be summarized to the below:

1. *--build-image*: Build the image that the compiled pipeline will use to run its steps.
2. *--build-path*: Path to a file/folder that will be part of the filesystem where the pipeline will run
3. *--build-image-name*: The name of the image that Kale will build. The expected format is: <name>:<tag>

4. *--registry-host*: The registry where the image will reside in
5. *--registry-auth-file*: Path to the authentication file of the credentials to log in to the registry.
6. *--registry-cert-file*: Path to the certification file (.crt, *.cert,*.key) to connect to the registry.

Each one of them has a representative environment variable which can be configured and decide for the build image advanced settings in case the corresponding CLI arguments are empty:

- --build-image-name → KALE_BUILD_IMAGE_NAME
- --registry-host → KALE_REGISTRY_HOST
- --registry-auth-file → KALE_REGISTRY_AUTH
- --registry-cert-file → KALE_REGISTRY_CERT

Compared to what Kubeflow Pipelines does we straight away realize the convenience our contribution provides. Data scientists won't interfere with either Docker images and their code's dependencies, or with registries, containers and Kubernetes.

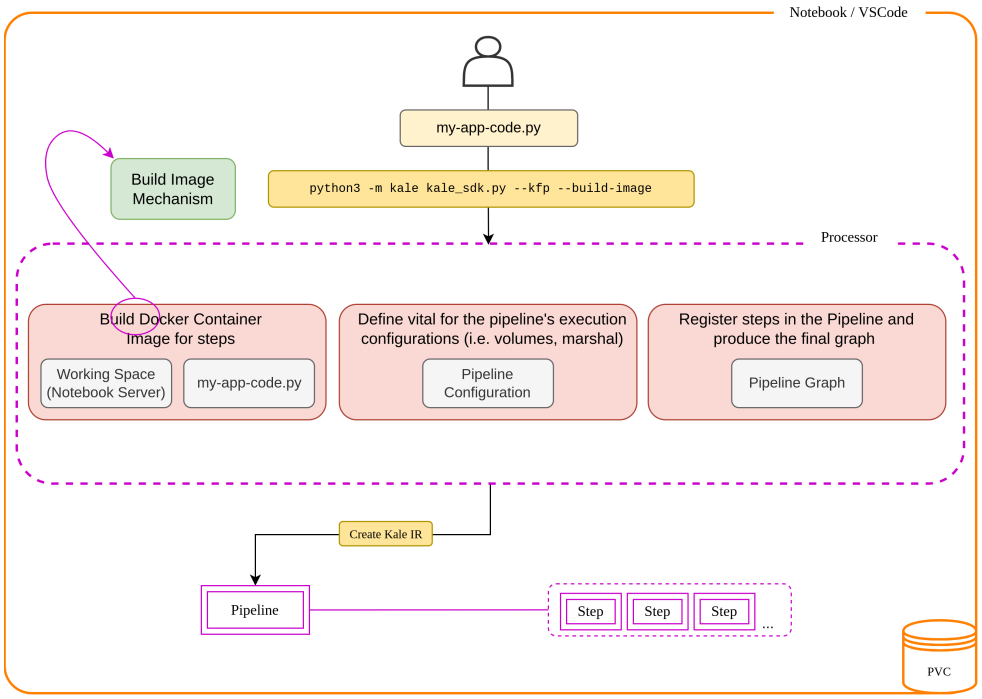


Figure 3.15: *Kale workflow execution path using the Build Image mechanism. Now, the processing part contains building behind the scenes an image for the pipeline steps.*

3.7.2 Private Docker Registry

In our effort to completely detach Data Scientists from Containers and Docker we extended Kubeflow's environment to include and deploy a Private Docker Registry, which is what makes the above approach so special. We have already emphasized that for Kubeflow Pipelines to run a component (or else a Pipeline step), the component must be packaged as a Docker container image and published to a container registry that the Kubernetes cluster can access. Our mechanism to build images on the fly in unprivileged environments would still require users to provide us with a registry to store the image. Definitely, it is not comparable to what users have to face when constructing Kubeflow Pipelines with the traditional way. However, we wanted to automate the process as much as possible for those with more conserving needs. We still offer users the ability to choose their own registry either from the relevant environment variables or the Kale's CLI.

A registry is an instance of the registry image provided by Docker. Therefore, it is pretty easy to deploy one in a Kubernetes cluster. What makes the process στριφνή is that the registry must be accessible from all nodes, and in particular, kubelet or container runtimes. This is strictly related to the kubernetes networking (2.7.5) we thoroughly ex-

amined in the background section of this thesis. What matters is the Kubelet-ClusterIP communication.

Kubelet-ClusterIP Communication

Kubelet under the hood will use a container runtime like cri-o to pull an image from a registry either local or remote. This kind of daemons exist at the host/OS level in the form of systemd services. That is, we only need to examine if the host of our cluster can communicate with a ClusterIP service. In the subsection 2.7.5 we prove that such a communication is feasible.

Eventually the host system, hence kubelet, can communicate with a ClusterIP service. All we need to do is expose our private Docker registry with a ClusterIP service and we are ready to go. We extend the Kubeflow's manifests with the required YAML Kubernetes resources to deploy alongside Kubeflow our Private Docker Registry. How will Kubeflow users know the name or address of the private registry? To achieve this, we define a new **environment** variable called *KALE_REGISTRY_HOST* in which we have saved the registry's name. Of course, users will be totally free to change this variable to a registry of their choice, if they think that the already deployed registry doesn't meet their requirements. Since Kubeflow runs on Kubernetes and each Notebook server runs on a different Pod, we make use of the *PodDefault* Custom Resource [62] in order to attach to new generated Pods the corresponding ENV variable.

Note that a PodDefault is a Kubeflow construct not a K8s construct. The goal of PodDefaults is to inject common data (env vars, volumes) to pods (e.g. notebooks). K8s has a resource called PodPreset with similar use-case.

The way we leverage the resource is:

1. We create PodDefault manifests which describe additional runtime requirements (i.e., volume, volumeMounts, environment variables) to be injected into a Pod at creation time. PodDefaults use label selectors to specify the Pods to which a given PodDefault applies.

2. Kubeflow components, which are in charge of creating pods (e.g., notebook controller) add some of available PodDefault labels to the pods when required

3. Admission webhook controller in general, intercepts requests to the Kubernetes

API server, and can modify and/or validate the requests. Here the admission webhook is implemented to modify pods based on the available PodDefaults

If we want to push an image to our private docker registry we should do the following:

```
docker pull nginx:latest
docker tag nginx:latest ${KALE_REGISTRY_HOST?}/my-nginx:latest
docker push ${KALE_REGISTRY_HOST?}/my-nginx:latest
```

3.7.3 Act I: Build Python function-based Components

Running Kubeflow Pipelines with our mechanism of building images on the fly was only the beginning. As stated in the thesis introduction, our ultimate goal is to create reusable and reproducible KFP components. By that we mean to not only build the image for the component but also to export its definition in the local filesystem to make it shareable and reusable. Thus, compiling and running a pipeline with the build image mechanism is not enough.

We want to create a community of ML engineers that can easily share, discover and reuse workflow components. Kubeflow, enhanced with Kale, will be the de facto ML platform that will bring Data scientists together to address the most difficult of ML tasks in the best possible way and share them with the community. ML engineers will be able to store and reuse KFP components in various different combinations saving valuable time towards finding the best ML workflow for their needs. Packaging a model will make serving a piece of cake even in the most demanding environments. Finally, the creation of reusable components will open the road for splitting ML workflows into distinct parts that will run on different clusters.

Our end goal is to produce a KFP component starting from a Python function that will be standalone, reproducible, reusable and transferable. In this thesis we describe a way to:

1. Compile a Python function into a reusable KFP component containing:
   (a) Inputs and Outputs
   (b) Container's image

   (c) Entrypoint

2. Run individually a KFP component in any pipeline independently of where it was developed or where it may end up residing.

The Data Scientist as the person that develops, tests and eventually deploys a ML Pipeline will do the following:

1. They develop an ML workflow consisting of a pipeline with a number of steps.
   (a) They have the choice to import components from other ML engineers that will fit their needs
   (b) They test different workflows combining ready to use components in their pipeline
2. When their workflow is production ready they instruct Kale to:
   (a) Produce a reusable KFP component starting from a python function.
   (b) Compile, upload and run their ML workflow.

A Machine Learning (ML) project usually consists of multiple interconnected steps. These steps can be thought of as separate processes, with clearly defined inputs and outputs. Thus, it becomes natural to think of ML projects as workflows or pipelines.

In this diploma thesis we implement a way to seamlessly transform a Python function into a Kubeflow Pipeline Component, and deploy it on Kubeflow using the Kale SDK. In particular, we introduce a new function for this purpose:

- save_component()

save_component() is a method of the step object. It transforms the step instance into a standalone KFP component. In particular:

- The step function's arguments and return annotations are used as component input/output types
- A new Docker image is built for the component including all the required dependencies for it to run

- Optionally, the method accepts advanced settings related to the component's image like its contents, name and the registry to be pushed alongside with the required authentication and/or certification files. If not provided, Kale assumes that the relevant values can be found either from their respective environment variables or from the CLI.

```
def save_component(save_path: str = None,
                   **build_kwargs):
```

The function's arguments in detail are:

- *save_path*: Path to a file where the component definition will be saved. If empty, the file is saved under CWD/<component.name>.kale.yaml
- ***build_kwargs*: build keyword arguments related to advanced build image settings (i.e. contents, name, registry)

A simple usage example of the *save_component()* is shown below:

```
>>> @step(name="add")
>>> def add(a: float, b: float) -> float:
>>>     return a + b
>>>
>>> add.save_component(
>>>     save_path="add.component.yaml",
>>>     build_paths=["./component_dep"],
>>>     build_image_name="my-component:v1",
>>>     registry_host="docker.io/my-registry"
>>> )
>>> add.docker_image
docker.io/my-registry/my-component:v1@<digest>
```

The following steps demonstrate the e2e user experience.

1. Define a step decorated function that could form an individual step in an ML pipeline:

```
@step(name="step1")
def add(a: int, b: int) -> int:
    '''Calculate sum of two arguments'''
    return a + b
```

2. Convert our function into a KFP component. Users have two choices to convert a step decorated function into a KFP component, either from the CLI or directly from the source where they develop their code:

   (a) *save_component()*:

   ```
   @step(name="step1")
   def add(a: int, b: int) -> int:
       '''Calculate sum of two arguments'''
       return a + b

   add.save_component(save_path="add.component.yaml")
   ```

   (b) CLI *--build-component* flag:

   ```
   python3 -m kale source.py::add --build-component
   ```

3. Kale uses the function's inputs and outputs to define the component's interface, thus the function's parameters must have type hints. See more details in section 3.6.7.

4. Even if our function has complex dependencies we manage to build a container image for our Python function to run in. We will seamlessly extend the workspace environment and include all the required code, data and installed libraries leveraging our build image mechanism. By default we will include the user's code, data and installed libraries. However, these choices can be overwritten with the build_paths function argument or the *--build-path* CLI flag (3.4).

5. After the successful execution of this command, we can look for the component's YAML file. The KFP component YAML specification should be very similar to this:

   ```
   name: add
   inputs:
   - {name="a", type="Integer"}
   - {name="b", type="Integer"}
   ```

```
outputs:
- {name="out", type="Integer"}
implementation:
  container:
    image: gcr.io/my-org/my-component@sha256:a172..752f
    command: [
      python3,
      -u,
      -m,
      kale,
      /home/jovyan/script.py::add
    ]
    args:
      - --in
      - a
      - {inputValue: a}
      - --in
      - b
      - {inputValue: b}
      - --out
      - out
      - {outputPath: out}
```

With only one command and minimum changes in the initial code we end up with a standalone, reproducible and shareable KFP component. Undeniably, our contribution removes the KFP's hassle to produce components, since the whole procedure is replaced with only one command.

Compile a Python function into a KFP component - Introspection
After executing '*python3 -m kale script.py::add --build-component*' Kale:

1. Processes the step function and defines the component's:
   (a) Name and description
   (b) Inputs and outputs inferred by the function's signature. Immediately Kale detects that the add function expects two inputs and one output with integer type. Thus, it comes to the below conclusions regarding inputs and outputs:
      i. Inputs: {a: Integer, b: Integer}

ii. Outputs: {out: Integer}

2. Configures the command that the container will execute once it is activated.

```
$ python3 -u -m kale /home/jovyan/script.py::add
```

The command instructs Kale to run the corresponding component of the provided pipeline script. Notice that this command assumes that both the script and the Kale library are included in the container image. *Repetitio est mater studiorum*: What is awesome about this is that data scientists don't need to have any expertise related to Docker and containers. We build the image for them behind the scenes with all the prerequisite code, data and libraries included.

3. Configures the container's command line arguments. These are essential in order for Kale to know the input and output values of the component. Since the inputs and output paths are passed in as command-line arguments, the component's code must be able to read inputs from the command line. Kale has already taken care of this. For inputs and outputs it uses the below CLI arguments:

```
--in name value & --out name value
```

This is why - as you may have noticed - the component's arguments follow the above notation. In this way, Kale understands that the component has:

- one input with name: a and value: {inputValue: a} and
- one output with name: out_1 and value: {outputPath: out_1}

The *inputValue* and *outputPath* are placeholders helping KFP to handle each case differently.

4. Finally, Kale builds a new container image in an unprivileged environment that comprises all the required component's code and data (including Kale) in order to be able to deploy and execute it. In brief, Kale retrieves the base image of the user's working environment, extends it with all the required dependencies for the component to run and pushes it to a registry to make it accessible to K8s container runtime.

3.7.4 Act II: Use a Kale Component in the Pipeline

Now we are ready to share our KFP component or retrieve a KFP component from another data scientist and use it in our pipeline. To do so, we extend Kale with the following method:

- kale.common.componentutils.load_component(component_path: str)

This function receives a path to a yaml file representing the component's specification and returns a Kale step instance that we can use in our pipeline. The following example demonstrates how to load the component specification and run it in a two-step pipeline. Arguments:

- *component_path*: Path to the component definition YAML file

1. In the python script that defines our ML workflow we can use the following:

```
@step(name="sum")
def print_num(num:  int):
   """Print a number"""
    print(num)

add = load_component("/path/to/component/yaml")

@pipeline(name="my_pipeline")
def pipeline_func(x1: int = 42):
    res = add(x1, 13)
    print_num(res)
```

2. Deploy and run our code as a KFP pipeline:

```
$ python3 -m kale kale_sdk.py --kfp
```

Note that nothing stops us from using this component into a raw Kubeflow Pipeline.

```
# Load add component
add_op = kfp.components.load_component_from_file("add.component.yaml")

# Define the pipeline function
```

```
def my_pipeline(a='1', b='7'):
    # Passes a pipeline parameter and a constant value to the `add_op` factory
    # function.
    first_add_task = add_op(a, 4)
    # Passes an output reference from `first_add_task` and a pipeline parameter
    # to the `add_op` factory function. For operations with a single return
    # value, the output reference can be accessed as `task.output` or
    # `task.outputs['output_name']`.
    second_add_task = add_op(first_add_task.output, b)

# Specify argument values for your pipeline run.
arguments = {'a': '7', 'b': '8'}

# Initialize KFP Client
client = kfp.Client()

# Create a pipeline run, using the client you initialized in a prior step.
client.create_run_from_pipeline_func(add_pipeline, arguments=arguments)
```

The above proves that Kale can produce reusable and shareable KFP components that can be integrated to either KFP or Kale pipelines.

Let's see again how the add external component looks like:

```
name: add
inputs:
- {name="a", type="Integer"}
- {name="b", type="Integer"}
outputs:
- {name="out", type="Integer"}
implementation:
  container:
    image: gcr.io/my-org/my-component@sha256:a172..752f
    command: [
      python3,
      -u,
      -m,
      kale,
      /home/jovyan/script.py::add
    ]
```

```
    args:
      - --in
      - a
      - {inputValue: a}
      - --in
      - b
      - {inputValue: b}
      - --out
      - out
      - {outputPath: out}
```

Notice the KFP placeholders *inputValue* and *outputPath*. We haven't called the component's function yet, which means that its argument values are still unknown.

The user compiles and runs their ML workflow with *'python3 -m kale kale_sdk.py --kfp'*. Before processing and compiling the pipeline, we **load the external components** and convert them into a step instance, retrieving from them: the inputs/outputs with their types, the name, the docker image and the source path. The step instance produced from the above KFP component is:

```
name: add
ins: (
    a, {
        name: a,
        param_type: Integer,
        param_value: None,
        source: None
    },
    b, {
        name: b,
        param_type: Integer,
        param_value: None,
        source: None
    }
)
outs: (
    out, {
        name: out
        param_type: Integer,
        param_value: None,
```

```
            source: None
        }
    )
    source_path: /home/jovyan/script.py
    docker_image: gcr.io/my-org/my-component@sha256:a172..752f
    marshal_path: None
```

Apparently, a step instance doesn't differ a lot from a component. We could say that a Kale step is a superset of a KFP component containing additional information required for Kale to make ML engineer's life easier. Then, Kale follows the same path to process, compile and run the pipeline. Since we're investigating KFP components it's essential to understand how they evolve during Kale pipeline's compilation.

At first Kale process the pipeline:

1. Configures:
    - Volumes that will be used during pipeline's execution
    - Absolute Working Directory
    - Marshal directory where steps will store marshal data
    - The container image to be used in internal steps
2. Registers the steps in the pipeline
    - Kale produces the final graph after configuring the dependencies between the steps representing the edges in the nodes of the graph.

The updated step instance is:

```
name: add
ins: (
    a, {
        name: a,
        param_type: Integer,
        param_value: None
        source: {
           name: None,
           param_type: Integer,
```

```
            param_value: 42,
            source: None,
            step: None
        }
    },
    b, {
        name: b,
        param_type: Integer,
        param_value: None,
        source: {
            name: None,
            param_type: Integer,
            param_value: 13,
            source: None,
            step: None
        }
    }
)
outs: (
    out, {
        name: out
        param_type: Integer,
        param_value: None,
        source: None,
        step: add
    }
)
entrypoint: python3 -m -u kale script.py::add --marshal-path
↪  /home/jovyan/.pipeline.kale.marshal.dir
docker_image: gcr.io/my-org/my-component@sha256:a172..752f
marshal_path: /home/jovyan/.pipeline.kale.marshla.dir
```

Some very useful observations to make are:

1. The updated marshal path and the entrypoint of the external step instance. Kale will run this step which means it will wrap it with a marshalling mechanism for the data passing between steps. Therefore it updates accordingly the entrypoint appending the marshal path after it is configured in the pipeline processing.

2. The updated *source* field of each input. This is done in (3bi) where Kale registers

the steps in the pipeline and sets the dependencies between them.

Then, Kale compiles the pipeline:

1. Creates a real KFP component for each step consisting of:
   - Name
   - Inputs, Outputs
   - Implementation: image, entrypoint, arguments
2. Produces a KFP task that will be part of the final KFP workflow

The *updated* KFP component will be:

```
name: add
inputs:
- {name="a", type="Integer"}
- {name="b", type="Integer"}
outputs:
- {name="out", type="Integer"}
implementation:
  container:
    image: gcr.io/my-org/my-component@sha256:a172..752
    command: [
      python3,
      -u,
      -m,
      kale,
      /home/jovyan/script.py::add,
      --marshal-path,
      /home/jovyan/.pipeline.kale.marhsal.dir
    ]
    args:
      - --in
      - a
      - {inputValue: a}
      - --in
      - b
      - {inputValue: b}
```

```
    - --out
    - out
    - {outputPath: out}
```

Regarding the Kale component itself the only difference we notice is the marshal path integrated in the entrypoint. Then why do we need to convert it into a step instance? Because, we need to register it in the pipeline, find from where the component will retrieve its inputs and describe the dependencies in the form of a graph that will eventually end up in the Argo workflow. The importance of converting into a step will be shown next where we will examine the produced Argo workflow.

Finally, Kale produces the final Argo Workflow which will be used to run the pipeline to KFP. The part of the workflow related to the external step looks like this:

```
- name: add
  container:
    args: [--in, a, '{{inputs.parameters.x1}}', --in, b, '13', --out, out, /tmp/outputs/out/data]
    command: [python3, -u, -m, kale, /home/jovyan/script.py::add, --marshal-path,
      /home/jovyan/.pipeline.kale.marshal.dir]
    env:
    - {name: PYTHONUNBUFFERED, value: '1'}
    image: gcr.io/my-org/my-component@sha256:a172..752
    securityContext: {runAsUser: 0}
    volumeMounts:
    - {mountPath: /home/jovyan/.pipeline.kale.marshal.dir,
      name: kale-marshal-volume}
    workingDir: /home/jovyan
```

As you can see now the *Input and Output KFP placeholders* have been replaced with either plain values or fields of the Argo that represent outputs of upstream steps. This is why we need to convert the external KFP component into a Step instance. It has to become part of the pipeline and eventually a node of the final graph that is connected with other nodes.

4 Implementation

4.1 Overview

In this chapter we unravel the secrets of our system giving a detailed description of the Build Image library along with the API of creating reusable and reproducible KFP components. In the previous chapter we presented some of the theoretical background, as well as the major design rationale and overall architecture of both systems. Yet, there are several aspects undiscussed. The process of the implementation of each system took place in multiple iterations, much like their design. Part of this chapter are the obstacles encountered while trying to put our design decisions into practice and the steps we took in order to overcome them. In addition, we outline some of the Kale patches we wrote to support our system.

Most of the code is written in the Python programming language. Below we have only included a few select segments of code, in order to aid the reader's understanding of our implementation. These are written in pseudocode, to dodge the complexity of some operations, as it is irrelevant to the purpose. The whole implementation consists of two major Python modules that extend the Kale SDK and are introduced as utilities:

- *imageutils*: A suite of helpers related to image manipulation. It mostly contains functionality to handle unpacked Docker and OCI images and extend them with new layers
- *componentutils*: A suite of helpers related to KFP components. It mostly contains functionality around KFP components (i.e. create KFP component, load compo-

nent and more)

4.2 The Secrets of Build Image Mechanism

In the previous chapter we presented a Python SDK that allows users to extend a base image with any sort of files or folders. This library enables us to seamlessly build Docker images on the fly in unprivileged environments, opening the road to streamline the process of building, training, and deploying machine learning (ML) models in a hybrid cloud environment.

4.2.1 Build Image

The outer level of this mechanism is the *build_image* function. First of all, the function's signature is:

```
def build_image(base_image: str,
                include_paths: List[Union[str, Tuple[str, str]]],
                dst_image: str,
                src_auth_file: str = None,
                src_cert_dir: str = None,
                dst_auth_file: str = None,
                dst_cert_dir: str = None) -> str:
```

Figure 4.1: *The build image function definition*

This function builds a new image from a base image and a set of files/folders. It accepts a valid base image name located in either a private or a public registry and a set of files and directories that will be added to the filesystem of the new image. It also accepts the appropriate credentials and certificates for authentication, when required by the repository.

Specifically the steps it follows are:

1. Brings the base image (specified by its name, tag and digest) to the filesystem
2. Extends the image by adding the provided files and directories to the filesystem of the image
3. Pushes the new image to a destination registry

Note that source authentication and certification files are used when pulling the base image from the source registry. Destination authentication and certification files are used when pushing the new image to a destination registry. The default authentication file path is *$XDG_RUNTIME_DIR/containers/auth.json* and the default certificates directory is */etc/containers/certs.d*. The function returns the name of the new image with <registry>/<name>:<tag>@digest format. Here is the implementation in a more pseudocode format:

```
def build_image(...) -> str:
    # Pulling image
    copy_image(from_registry, to_local_fs)

    # Extending image
    extended_image = extend_image(base_image_path, include_paths)

    # Pushing image
    copy_image(from_local_fs, to_registry)

    # New image successfully built
    return dst_image + "@" + extended_image.image_digest
```

Figure 4.2: *The build_image() implementation in pseudocode format*

As you can see we call *copy_image()* for the image shipment (pull and push operations). This function copies an image from a source to a destination location using the Skopeo tool. Skopeo works with API V2 container image registries such as docker.io and quay.io registries, private registries, local directories and local OCI-layout directories. When required by the repository, Skopeo can pass the appropriate credentials and certificates for authentication.

'skopeo copy' replaces this::

```
docker pull internal.registry/myimage:latest
docker tag internal.registry/myimage:latest prod.registry/myimage:v1
docker push production.registry/myimage:v1.0
```

with this::

```
skopeo copy docker://internal.registry/myimage:latest \
            docker://production.registry/myimage:v1
```

with a major advantage: No need of root privileges or a docker daemon installed.

Some image format examples are the following:

- *containers-storage:docker-reference*: An image located in a local containers/storage image store.
- *dir:path*: An existing local directory path storing image files
- *docker://docker-reference*: An image in a registry implementing the Docker registry HTTP API v2
- *docker-daemon* An image docker-reference stored in the docker daemon internal storage.
- *oci:path:tag* An image tag in a directory compliant with "Open Container Image Layout Specification" at path.

Unfortunately, Skopeo does not provide a Python SDK, therefore we need to use the executable directly. Later in this chapter we're going to see an alternative and more efficient way to pull and push images for our purpose using the Docker Registry API. We haven't yet implemented to the extent we want in order to work in a production level. We used skopeo as a proof of concept for this thesis.

4.2.2 Extend Image

Now *extend_image* comes into play. This function automates the whole process we thoroughly explained in the previous chapter 3.3. The function's signature is:

```
def extend_image(base_image_path: str,
                 include_paths: List[Union[str, Tuple[str, str]]]
                 ) -> DockerImage:
```

It accepts a base image which is a path to a valid image's files and extends it by appending a new layer composed of a bunch of files and/or folders. The procedure followed by the function is:

1. Builds the tarball consisting of the provided files/folders

2. Appends a new layer on top of the base image using the above tarball following the algorithm we showed in section 3.3

3. Returns the new extended image

In the *extend_image()* function we do the following:

1. Validate the base image path, that is, ensure that the path exists. The *os.path.exists()* [23] function is our right-hand for this.

2. Instantiate a new *DockerImage* object for the above base image

3. Process and validate the paths that will comprise the new layer

4. Build a new tar archive that is will be finally appended on top of the base image.

Below we show the function's implementation in a pseudocode format:

```
def extend_image(...) -> DockerImage:
    # Validating base image
    validate_base_image()
    image = DockerImage(base_image_abspath)

    # Processing paths that will comprise the tar archive
    tar_members = process_include_paths(include_paths)

    # Building tar archive
    tar_file = build_tar(tar_members)

    # Appending new layer
    image.append_layer(tar_file)
    return image
```

Figure 4.3: *The extend_image() implementation in pseudocode format*

The *process_include_paths()* function validates and processes a list of paths. In particular, it produces valid tuples of source and destination paths destined for arguments of the tar build procedure. Source represents the path to a file in the local filesystem and destination the path within the tar. Here is a simple example of the function's usage:

```
>>> include_paths = [”/home/test”]
>>> processed_paths = _process_include_paths(include_paths)
>>> print(processed_paths)
[(”/home/test”, ”/home/test”)]

>>> include_paths = [(”/home/test”, ”/hello/world”)]
>>> processed_paths = _process_include_paths(include_paths)
>>> print(processed_paths)
[(”/home/test”, ”/hello/world”)]
```

In the end, *tarfile.open* [24] along with *tar.add* [25] will receive an absolute/relative path to the source file and an absolute/relative path to the destination within the tar archive. However, users may not want to provide a different destination path within the tar, thus we manage to adjust accordingly their input to what tar.add expects. During the implementation of this mechanism, we strived to make it handy and user friendly by allowing different variations of inputs and raising an informative error when the input format is not right.

```
def process_include_paths(include_paths):
    processed_paths = []
    if path is not a List:
        raise Error
    for path in include_paths:
        if path is String:  # Empty destination path
            path = os.path.abspath(path)
            src, dst = path, path
        else if path is Tuple:
            # Both src and dst path provided as a tuple
            src, dst = path
        else raise Error
        processed_paths.append((src, dst))

    return processed_paths
```

Figure 4.4: *Path processing to produce valid tuples for the tar.add operation*

4.2.3 DockerImage Class

Before, we referred to a *DockerImage* object. The *DockerImage* class represents a Docker Image in an unpacked format. A typical structure of a Docker image is:

```
image/
├── <configuration_digest> <- Configuration JSON file
```

```
├── <layer1_digest>
├── <layer2_digest>
    .
    .
├── <layerN_digest> <- tar.gz file
├── manifest.json
└── version
```

As we already know, the manifest file is the signature of the image containing metadata (size and digest) about the configuration file and the layers of the image. The configuration file is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime.

This class acts like a mirror for a Docker image in an unpacked format

1. It loads vital image's components
   (a) Manifest file
   (b) Configuration file
2. It retrieves various image's attributes:
   (a) The nth layer's digest, length and diff_id
   (b) The image's digest
3. It appends a new layer on top of the image

Some of the most important class' methods are:

- *read_manifest_file() and read_config_file()* which return the raw data of each file respectively. Both make use of the *open()* [26] Python function.
- *get_layer_digest() and get_layer_len()* which return the length and size of the nth layer. These metadata are written in the manifest and configuration file. The function won't do more than opening the files and retrieving the relevant values from the corresponding fields.
- *append_layer()* which appends a new layer on top of the image.

4.2.4 Append Layer

The way to append a layer on top of an image is to edit the manifest and the configuration file. This is something we realized in the previous chapters. Specifically, the manifest file keeps track of a list of the compressed layers' sizes and digests and the config's size and digest. The configuration file, on the other hand, records a list of root filesystem changes, that is, the digests of uncompressed tar archived layers. Note that the configuration and the layer file names must be the digest of their own contents.

Consequently, this function receives a path to a tar archive file which represents the new layer. Then, *append_layer(tar_file)* does the following:

1. Loads the configuration file using the read_config_file() function. Then, it computes the digest of the tar archived layer with the *hashlib* Python library [27] and adds it to the configuration file.
2. Computes the updated configuration's length and digest. The os.path.getsize function helps us compute the file's size.
3. Zips the layer using *gzip* [28] and gets its digest and length.
4. Finally, it updates the manifest file and does the required file renaming

```
def append_layer(self, tar_file):
    configuration = self.read_config_file()
    # Append new diff_id under diff_ids field in configuration object
    configuration["rootfs"]["diff_ids"].append(
        compute_file_digest(tar_file))
    config_digest = compute_sha256_digest(configuration)

    # Zip layer and compute its metadata (size, digest)
    zip_layer_digest = self.zip_layer(tar_file)
    zip_layer_len = os.path.getsize(zipped_layer)


    self.update_manifest(configuration_len, config_digest,
                         zip_layer_len, zip_layer_digest)

    save_config_file()
```

```
        # Rename the compressed tar archive to its own digest
        # to make it part of the image
        rename_layer()
```

update_manifest() adds and edits metadata like the length and the digest in the layers and the config field respectively. These additions indicate that the image has changed. Let's see the arguments in more detail:

- *config_len*: Length of the config object
- *config_digest*: Digest of the config object
- *zip_layer_digest*: Digest of the tar zipped layer object
- *zip_layer_len*: Length of the tar zipped layer object

We present in brief the steps followed to update the manifest file:

1. Open and read the manifest file using the *read_manifest_file()* function.
2. Adjust appropriately the *manifest["config"]*
3. Append the new layer's metadata to the *manifest["layers"]* list.

4.2.5 Skopeo Local Directory vs OCI Storage

Skopeo is a tool for moving container images between different types of container storages. One choice is dir:path which represents an existing local directory path where we can store images files in all the supported skopeo formats (like oci format or Docker's v2 schema 2 format). We thought that this storage option would be a great choice and suitable for our use case. Unfortunately, we realized that it is not fully maintained and it is only recommended for testing purposes. One of the major disadvantages of this backend that we bumped into was that they haven't implemented a blob caching mechanism.

```
$ skopeo copy docker://busybox:latest dir:./base_image
Getting image source signatures
Copying blob 5cc84ad355aa done
Copying config beae173cca done
Writing manifest to image destination
Storing signatures
pangiann@WhiteRose:~$ skopeo copy docker://busybox:latest dir:./base_image
Getting image source signatures
Copying blob 5cc84ad355aa done
Copying config beae173cca done
Writing manifest to image destination
Storing signatures
```

As you may have noticed, all of the image's layers and files are downloaded two times which makes it time inefficient. Consider that users will want to continuously iterate on their models, thus building again and again images for their components. Therefore, we decided that we need to support the oci:path storage. For this to happen, we need to handle images in the OCI format in order to be accepted from the relevant storage. With 'handle' we mean to expand our build image mechanism and support extending images with OCI format apart from the Docker ones.

4.2.6 Docker Registry API vs Skopeo

The Docker Registry HTTP API is the protocol to facilitate the distribution of images to the docker engine. It interacts with instances of the docker registry, which is a service to manage information about docker images and enable their distribution.

Docker uses the v2_2 format, which we know on a great scale, to store images in Docker registries. This is also the format (along with the OCI image) we use to manipulate, edit and extend an image. In fact, we bring the image to the local filesystem from a registry in a v2_2 or OCI format and then we extend the image by editing the manifest and the configuration file. This method has some implications, because most of the container runtimes cannot run a container using this image format. Docker uses a storage driver to store image layers. The same thing holds for Podman. The most common and preferred storage driver is overlay2. The storage driver controls how images and containers are stored and managed on the host system.

OverlayFS layers two directories on a single Linux host and presents them as a single

directory. These directories are called layers and the unification process is referred to as a union mount. OverlayFS refers to the lower directory as lowerdir and the upper directory a upperdir. The unified view is exposed through its own directory called merged.

After downloading a five-layer image using docker pull ubuntu we notice six directories under /var/lib/docker/overlay2. The five of them represent an overset of the image's layers. The lowest layer contains a file called link, which contains the name of the shortened identifier, and a directory called diff which contains the layer's contents. Note that layers are not stored as compressed tar archives like we described in the previous chapters. The second-lowest layer, and each higher layer, contain a file called lower, which denotes its parent, and a directory called diff which contains its contents. It also contains a merged directory, which contains the unified contents of its parent layer and itself, and a work directory which is used internally by OverlayFS.

As you can see, container engines and docker daemon use a completely different way to store and access on disk the container images, which doesn't share any similarities with the v2_2 or OCI image format which we exploit to modify an image. The last two formats are beneficial for the network bandwidth, since layers are compressed and the size is significantly reduced. Nevertheless, container engines manage to decompress layers and store them accordingly on disk efficiently, allowing later container runtimes to access them.

Eventually, to complete the procedure of building an image, after bringing it to the local fs using v2_2 / OCI format and extending it by modifying the corresponding files, we need to push the new extended image to a registry, either local or remote. And local refers to a container-storage as the one we depicted above. For this purpose, until now in our mechanism we used Skopeo which copies an image from a source to a destination. Skopeo supports a lot of container image's formats and it is responsible for the corresponding conversion. For example, we can pass our extended image to the docker daemon and then use it with the docker command. Notice that we are forced to use sudo to communicate with the docker daemon which is one of the Docker's major disadvantages.

```
$ sudo skopeo copy oci:busybox/ docker-daemon:extended-busybox:latest
[sudo] password for pangiann:
Getting image source signatures
Copying blob 50e8d59317eb done
```

```
Copying blob 82d377b6dfbf done
Copying config 7b59ae3ca3 done
Writing manifest to image destination
Storing signatures
$ sudo docker images  | grep extended-busybox
extended-busybox   latest   7b59ae3ca33c   4 weeks ago    1.24MB
```

Although Skopeo seems to be the ideal tool for performing copy operations and work with remote image registries, it hasn't been created for our use case, which is to extend images, thus revealing some of its drawbacks. Specifically, Skopeo forces us to download the full contents of an image, that is, all the compressed layers, the manifest and the configuration file. In the figure 3.8 you can notice that to append a new layer we only need:

- The manifest file
- The configuration file
- The new layer

Imagine downloading an image with plenty of layers and massive size. How time, space consuming and inefficient is it? Taking that into consideration, an adequate alternative would be to interact with a Docker registry using the Docker Registry API. We'll start to use this mechanism alongside Skopeo. That is:

1. Detect if the target registry supports the Docker API
2. If yes, use the API to efficiently extend an image
3. If no, use Skopeo

A minimal endpoint, mounted at '/v2/' will provide version support information based on its response statuses[29]. The request format is as follows:

```
GET /v2/
```

- *If a 200 OK* response is returned, the registry implements the V2(.1) registry API and the client may proceed safely with other V2 operations.

- *If 404 Not Found* response status, or other unexpected status, is returned, the client should proceed with the assumption that the registry does not implement V2 of the API.

These are the steps of the algorithm that bring a container image to the local filesystem, extend it and push it to a registry using the Docker Registry API.

1. GET the authentication token for the registry communication
2. GET the manifest file of the base image
3. Extend the image using our Python SDK
4. POST the new tar gzipped layer to a registry of our choice
5. PUT the updated configuration file
6. PUT the updated manifest file
7. Confirm that we have successfully built a new image

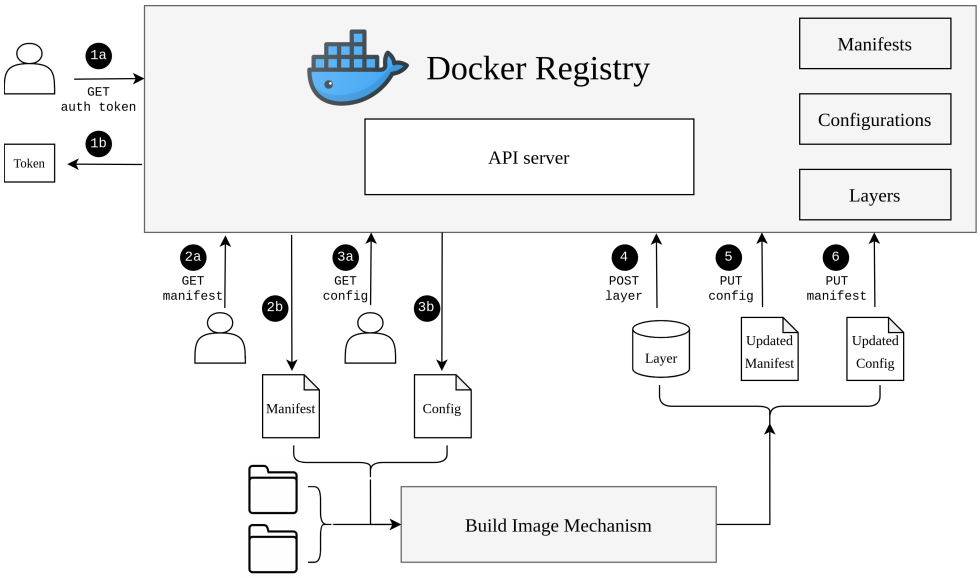


Figure 4.5: *Communication with the Docker Registry API server to extend an image*

As a base image we will use busybox:latest. The documentation about requesting a Manifest file informs us:

```
GET /v2/<name>/manifests/<reference>
Host: <registry host>
Authorization: <scheme> <token>
```

Despite the fact that we observe an Authorization Header, we presumably assume that it will probably be needed in private repositories. As brave as a Software Engineer could be, we ignore it and continue with the request. Unfortunately, we immediately realize the need for authorization given the *401 Not Authorized* response we received. In the response, the field *WWW-Authenticate* header indicates how to authenticate ourselves to the repository.

```
Bearer realm="https://auth.docker.io/token",service="registry.docker.io",scope
="repository:library/busybox:pull",error="invalid_token"
```

GET authorization token

Following that, we understand that a GET request is required to the endpoint *https://auth.docker.io/token* with QUERY Parameters:

- service: registry.docker.io
- scope: repository:library/busybox:pull

Our HTTP GET request for the access token will be:

```
GET https://auth.docker.io/token?service=registry.docker.io&scope=repository:
library/busybox:pull
```

GET manifest file

Now, we try again to pull busybox's Manifest file including in the request the new token. A small reminder: Docker includes two schema versions for a container image representation. They created the version 2 (V2) schema 1 of the image manifest back in 2016 and therefore deprecated version 1. After multiple iterations of various image format improvement approaches, schema 2 has been released to supersede the existing schema 1 in 2017. Therefore, we explicitly request the v2_2 manifest [63].

Our new GET Request will be:

```
GET https://index.docker.io/v2/library/busybox/manifests/latest
```

The *Authorization* field will contain the new token to authorize us to pull the image. We also fill the Accept field with *application/vnd.docker.distribution.manifest.v2+json* to explicitly ask for the new image Manifest format.

GET configuration file

The Docker Registry API provides us with the below GET request to retrieve the configuration file.

```
GET /v2/<name>/blobs/<digest>
Host: <registry host>
Authorization: <scheme> <token>
```

In fact, this request fetches a blob from the registry identified by its digest. We already know how to discover the config's digest. It lives in the manifest's config field. Hence, our request will have the form below:

```
GET /v2/library/busybox/blobs/sha256:<digest>
```

Likewise, the Authorization field contains the token we received previously. Now, the Accept field has the *MediaType* of the configuration file which is: *application/vnd.docker.container.image.v1+json*.

Extend Image

We will definitely leverage our build image mechanism to extend the base image with some random files.

```
extend_image("/home/user/busybox/",
             [("/home/user/src", "/home/user/data")])
```

This means that now a new compressed tarred layer with the updated manifest and configuration file exists in our filesystem.

Upload the new layer

All layer uploads use two steps to manage the upload process.

- The first step initiates the upload process in the registry service, returning a url to carry out the second step
- The second step uses the upload url to transfer the actual data. Note that we can upload a blob in chunks or we can choose a monolithic upload, pushing them all at once.

Initiate Resumbale Blob Upload

Uploads are started with a POST request which returns a url that can be used to push data and check upload status:

```
POST /v2/<name>/blobs/uploads/
Host: <registry host>
Authorization: <scheme> <token>
Content-Length: 0
```

This endpoint helps us initiate a new location where the communication between the client and the registry will be held. This allows us to transfer data in chunks. On success we should receive as a response a 202 Accepted status alongside the location.

```
202 Accepted
Content-Length: 0
Location: /v2/<name>/blobs/uploads/<uuid>
Range: 0-0
Docker-Upload-UUID: <uuid>
```

We will push the new extended image to a public repository of ours living in docker hub.

```
POST /v2/pangiann/docker-image/blobs/upload
```

The response is indeed 202 Accepted. The location is a UUID and can be found in the corresponding field in the response headers.

Monolithic Upload

To upload in a monolithic way our new layer, a PUT Request is required. In case we want to send the contents in chunks the same endpoint will be used but with a PATCH Request. The final chunk must be always sent with a PUT Request.

```
PUT /v2/<name>/blobs/uploads/<uuid>?digest=<digest>
Host: <registry host>
Authorization: <scheme> <token>
Content-Length: <length of data>
Content-Type: application/octet-stream

<binary data>
```

As a response, we should receive a 201 Created status.

Upload the updated Configuration file

The configuration file is just another blob for our registry. Therefore, we don't deviate from the procedure we followed when uploading our new layer.

1. Initiate a blob upload:

   ```
   POST /v2/pangiann/docker-image/blobs/uploads/
   ```

This request returns to us a location (UUID) to push our blob

2. Upload the configuration file:

```
PUT /v2/pangiann/docker-image/blobs/uploads/UUID
```

This request includes the digest of the configuration file and the actual data in the request's body.

Upload the updated Manifest file

The last step to complete the process of building a new image is to upload the updated manifest file to the registry. This requires all the layers and the configuration file to be already uploaded. Remember that the manifest file includes metadata for both of them. Thus, a registry will try to validate the information that the manifest carries. If any of the layers or the configuration file is not uploaded with their respective digests, then the registry will not accept the request. Consequently, even though layers will be there, the image will not be alive. The manifest is the file that brings to life an image and makes it accessible and visible. The reason is that it defines which layers and which configuration corresponds to which image name and tag. The request is:

```
PUT /v2/<name>/manifests/<reference>
Host: <registry host>
Authorization: <scheme> <token>
Content-Type: <media type of manifest>

<manifest data>
```

What does the reference mean and represent? The **tag** of the image. Tag is tightly coupled with the manifest. Imagine having 100 layers. One manifest could point to 3 of them and be tagged as latest. Another manifest could point to 10 of them with a v2 tag. This figure (δειξε φιγκιουρ που εφτιαξες) portrays this idea. On success we should wait for the below response:

```
201 Created
Location: <url>
Content-Length: 0
Docker-Content-Digest: <digest>
```

Our request will be:

```
PUT https://registry-1.docker.io/v2/pangiann/docker-image/manifests/v2
```

Notice that we used the v2 tag for our extended image.

Verify Section

As a verification step, we will pull from the registry our new extended image and run it using docker.

```
$ docker pull docker.io/pangiann/docker-image:v2
v8: Pulling from pangiann/docker-image
5cc84ad355aa: Pull complete
f47e99253df1: Pull complete
Digest: sha256:f1d47858e65dcc968139f61342086a89f3eb6e13b6522188a724d9a97677572d
Status: Downloaded newer image for pangiann/docker-image:v2
docker.io/pangiann/docker-image:v2
# docker run -it pangiann/docker-image:v2 sh
/ # ls /home/user
src data
```

4.3 The Secrets of Building Kale Components

A common practice today is to deploy ML jobs as containers managed by the Kubernetes orchestrator. Kubeflow is a machine learning toolkit for Kubernetes strived to make deployments of machine learning (ML) workflows on Kubernetes simple, portable and scalable. With Kubeflow, data scientists and engineers are now able to develop a complete pipeline composed of segmented steps. These segmented steps in Kubeflow are loosely coupled components of an ML pipeline, allowing pipelines to become easily reusable and modifiable for other jobs. Unfortunately, Kubeflow's way to create reusable components consists of multiple manually executed steps making the process a cumbersome experience for Data Scientists both in the development and production stage. The main issue is located around building images in a transparent and automated way, since Kubeflow is not able to disengage ML engineers from such operations.

Let's remind ourselves of the KFP component definition: A Kubeflow Pipelines component is a containerized application - self-contained set of code - that performs one step in an ML workflow. A pipeline component is composed of:

- The component code, which implements the logic needed to perform a step in the ML workflow

- A component specification which defines the following:
  - The component's metadata, its name and description
  - The component's interface, the component's inputs and outputs
  - The component's implementation, the Docker container image to run, how to pass inputs to your component code, and how to get the component's outputs.

4.3.1 Save Component

The thesis ultimate goal is to abstract the transition from a Python function to a KFP component definition. Our dream is to allow users to identify a Python function with a KFP component. That is exactly save_component()'s purpose.

```
def save_component(save_path: str = None,
                   **build_kwargs):
```

The function's arguments in detail are:

- *save_path*: Path to a file where the component definition will be saved. If empty, the file is saved under CWD/<component.name>.kale.yaml
- ***build_kwargs*: Build arguments related to advanced build image settings (i.e. contents, name, registry)

save_component() returns the updated step instance including the new Docker image.

Before moving on to the inner levels of the function it is important to highlight that we extended Kale's CLI in order for users to leverage our mechanism directly from there. Kale's Python CLI is nothing more than an advanced **argument parser** [64]. For the shake of completeness we show how we extended the CLI for our build image mechanism using the *--build-image* flag.

```
main_group.add_argument("--build-image", dest="BUILD_IMAGE",
                        action="store_true",
                        help="Build the image that the compiled pipeline"
```

```
                    " will use to run its steps. --no-snapshot action"
                    " is implied. The base image is the container's"
                    " image where Kale is running. If no "
                    " '--build-path' is specified, the following paths"
                    " are included: ./<pipeline_script>.py, ./data,"
                    " ./src, ~/.local")
```

As you can see we configure the name of the argument, the help message that will be displayed and what will be the variable where the value of the argument will be stored to.

Back to the implementation, There are 3 stages to move from a function to a component definition lying in the local filesystem.

1. The first stage with paramount importance is to package the component's code into a Docker container image. The build image library will be responsible for this.

2. The second stage encompasses the metamorphoses of the step decorated function into a KFP *ComponentSpec* object.

3. Finally, at the third stage, the *ComponentSpec* object must be converted into a yaml specification file that will reside in the local filesystem making it shareable and transferable.

Stage1: Package Component into a Docker Image

The first step towards building reusable and reproducible KPF components is to package the component's code and its various dependencies into an image. This means that we'll need to import our new build image library and use its API. The component's image will also include the user's working environment, that is, the Jupyter Notebook server they're working on.

```
base_docker_image = get_docker_base_image()
step.docker_image = build_image_component(base_docker_image, image_settings)
```

get_docker_base_image() gets the current container's docker image. In Kubeflow, a Notebook server runs in a container on a K8s Pod. In other words, this function returns to us

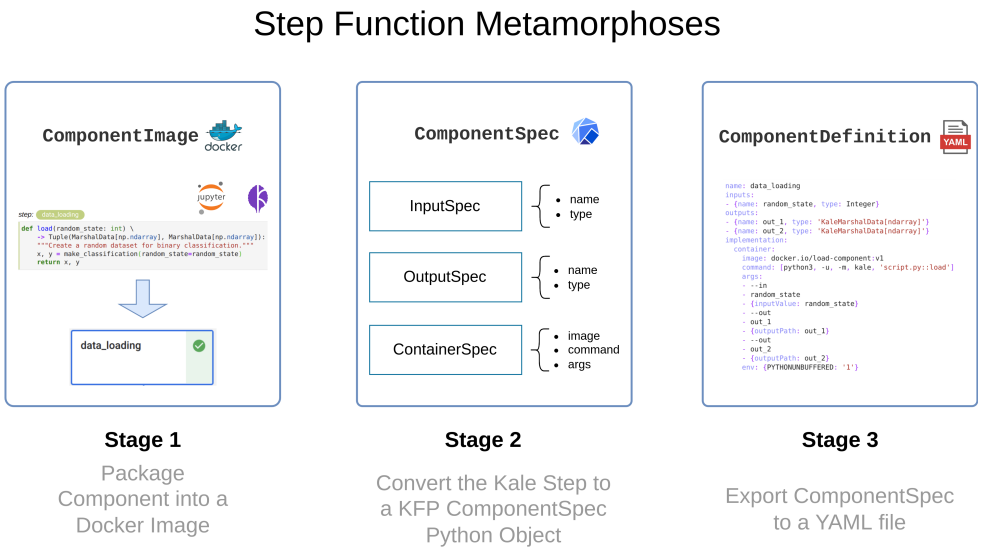


Figure 4.6: *From a step decorated function to a component definition*

the user's environment. The Kubernetes Python SDK is used for this job by providing a relevant API. Eventually, save_component() will end up calling our imageutils.build_-image(). Let's remind ourselves the API's signature:

```
def build_image(base_image: str,
                include_paths: List[Union[str, Tuple[str, str]]],
                dst_image: str,
                src_auth_file: str = None,
                src_cert_dir: str = None,
                dst_auth_file: str = None,
                dst_cert_dir: str = None) -> str:
```

build_image() accepts a valid base image located in either a private or a public registry and a set of files and directories that will be added to the filesystem of the new image. It also accepts the appropriate credentials and certificates for authentication, when required by the repository. If executed successfully, it returns the name of the new image.

It goes without saying that there are a lot of decisions that need to be made regarding the build_image's arguments. build_image_component() aims to implement the required business logic.

```
def build_image_component(base_docker_image: str,
                          build_paths: List[Union[str,Tuple[str, str]]] = None,
                          build_image_name: str = None,
```

```
                                registry_host: str = None,
                                registry_auth_file: str = None,
                                registry_cert_file: str = None) -> str:
```

build_image_component() as the name suggests builds the component's Docker image starting from the user's working environment, that is, the Jupyter notebook's base. Essentially, this method accepts the notebook's base Docker image and extends it with the minimum component's requirements. The main purpose of it is to define the priority regarding advanced build image settings among function's argument, CLI arguments and Environment variables. The priority is:

- Function arguments
- CLI arguments
- Environment variables

In brief, if not defined in the function, the CLI input takes precedence about the image's settings like contents, name, registry. In case they are both empty, the values are retrieved from the respective ENV vars. Regarding the image's contents the default behavior is to include the following paths: *./<pipeline_script.py*, *./data*, *./src*, */.local*. The idea behind this choice is to comprise the user's source code where they develop their pipeline and all the Python installed libraries living under /.local. Then hopefully we can involve user's source and data dependencies with the convention that they live under ./data and ./src. In all cases, the most demanding users can specify the image's contents based on their use case.

```
>>> from kale.common import componentutils
>>> image = componentutils.build_image_component(
>>>     base_docker_image="gcr.io/arrikto/kale-py38",
>>>     build_paths=["./component_dep"],
>>>     build_image_name="my-component:v1",
>>>     registry_host="docker.io/my-registry"
>>> )
>>> image
'docker.io/my-registry/my-component:v1@<digest>'
```

Stage2: Convert a Kale Step to a KFP ComponentSpec

At this stage, our step instance is completed and ready to be reshaped into a KFP ComponentSpec, that will allow us to produce a valid yaml definition. Why do we need this intermediate stage? Why not move from the step object directly to the yaml file? In fact, this can happen, but it would require a lot of code already written from KFP. Our intention is **not** to bypass KFP but to expand it and make it easy to use. Back to the implementation, the step's name is used as the component name. Argument and return annotations are used as component input/output types. Kale's typing system takes care of this. Specifically, the *inspect* [65] Python library provides to us the required API to retrieve a function's signature and in turn the input names and typing annotations. The new Docker image created in Stage 1 is placed in the container's implementation.

ComponentSpec

What is a ComponentSpec? A ComponentSpec is a KFP class object that describes the metadata (name, description, labels), the interface (inputs and outputs) and the implementation of the component. In particular, it consists of the following key attributes:

- name: Optional[str]
- description: Optional[str]
- metadata: Optional[MetadataSpec]
- inputs: Optional[List[InputSpec]],
- outputs: Optional[List[OutputSpec]],
- implementation: Optional[implementationType]

We can already discern the similarities between the ComponentSpec and the corresponding yaml component definition and how the former could easily evolve and be transformed into the latter.

InputSpec & OutputSpec

The InputSpec class describes the component input specification. The attributes that compose the spec are:

- name

- type
- description
- default
- optional

Equally, the *OutputSpec* class describes the component output specification, although it can't be optional or have a default value. The first 3 arguments stay the same:

- name
- type
- description

ContainerSpec

The *ImplementationType* ends up being a *ContainerSpec* which in turn describes the container component implementation.

- image
- command
- args
- env

From a Step to a ComponentSpec

The conversion of a step to a KFP ComponentSpec mostly includes the appropriate initialization of all of the above objects.

- At first, for each step input we create a new InputSpec object with the relevant name type and default value, all acquired from the step function signature. The same holds for the OutputSpec.
- After that, we instantiate and fill with the required information the ContainerSpec.

```
    @property
    def component(self) -> kfp_structs.ComponentSpec:
        """Create a KFP ComponentSpec."""
        for input in self.ins:
            InputSpec(name=input.name, type=input.type, default=input.default_value)

        for output in step.outs:
            OutputSpec(name=output.name, type=output.type)

        container=ContainerSpec(
            image=self.docker_image,
            command=self.entrypoint,
            args=self.cli_args,
        )
```

Recall that the step's entrypoint is the source of truth for a container's execution and thus a step's execution. Note that we don't directly execute the user's code. On the contrary, we point to Kale the script and the function step. Kale will load the module, find the step decorated function and run it in a more advanced way wrapped with Kale's marshalling logic for data passing between steps. Concerning the CLI arguments, they are essential for Kale to know the input and output values of the component.

Stage3: Export ComponentSpec to a YAML file

At this stage, we have all the required information for the component to run gathered in a Python object. Naturally, we cannot share an object to others or ship it in other clusters and environments. The most convenient way to make an object shareable is to save it in the local filesystem as a file. In our case, this file will have a yaml format, as it happens lately for most of the object specifications. YAML is a human-friendly data serialization standard [66]. It is a language-independent specification that enables data interoperability across all programming languages. YAML enables us to describe real-world objects in an easy-to-read format with extreme capabilities when it comes to creating complex data structures.

Needless to say, there is a Python library called yaml to cover and handle most of the operations around yaml files. It is a two-step process to save the ComponentSpec into a yaml file:

1. Convert the object to a dictionary

2. Dump the dictionary to a file in a yaml format

There are a lot of different ways to produce a dictionary from a Python object. Kubeflow has created their sophisticated way, implementing at the same time their business logic. The referred-to function is called to_dict(). It converts an object to structure (usually a dict). Its specialty is that it serializes all properties that do not start with underscores. If the type of some property is a class that has *.to_dict* class method, that method is used for conversion.

Regarding dumping the dictionary to a yaml file, the relevant Python SDK provides us with a *yaml.dump()* function. Still, Kubeflow has numerous reasons to not fully trust it, since each use case has different expectations. For example, see more here [67].

Therefore, Kubeflow carefully wraps this function for a safe for them dump of the dictionary into a yaml. It is clear now why Kale doesn't want to throw into the bin Kubeflow's work, but only clean their mess and save users from their boilerplate.

In the aggregate, we present the implementation of our way to produce reusable and shareable KFP components:

```
def save_component(...):

    # Build component's image based on the working environment's image
    base_docker_image = get_docker_base_image()
    step.docker_image = build_image_component(base_docker_image,
                                              **build_kwargs)

    # Convert the step instance into a ComponentSpec
    component = self.component

    # Save ComponentSpec into a YAML file
    with open(output_path) as f:
        f.write(dump_yaml(component.to_dict()))
```

Figure 4.7: *From a Python function to a component yaml definition*

4.3.2 Load Component

It stands to reason that loading a component follows almost the reversed procedure to that of creating a component. Of course, we cannot reverse building a Docker image for the component, thus it comes without saying that will omit this stage. The two remaining stages are:

- Load component definition from a YAML file to a Python ComponentSpec object
- Convert the ComponentSpec into a Step instance that Kale can process, control, register to a Pipeline and execute it

Figure 4.8: *The conversion of a YAML file to a Kale step*

For this to happen, we introduce the below function:

```
def load_component(component_path: str) -> ExternalStep:
```

Stage1: From a YAML file to a ComponentSpec

Naturally, KFP provides us with the corresponding API. Let's repeat that we don't replace KFP, we keep its benefits and enhance it to make it more convenient for the end user.

At first we need to open and read the component file's contents. Of course, what we have is a stream of bytes completely rigid and non-manipulable. Therefore, it is necessary to convert them into a dictionary object. KFP offers us the respective to *dump_yaml()* function called *load_yaml()*. It takes as input a stream of bytes that form a yaml specification file and produces a dictionary object. Note that there is a *yaml* library with a *load* attribute, however KFP wraps that function with the aim to overcome some of its issues.

Second, the dictionary must be converted into a KFP *ComponentSpec* object. As it happened with *to_dict()*, KFP has the respective *from_dict()* function that does the opposite operation. That is, it receives a Dictionary Python object and turns it into a ComponentSpec.

Stage2: From a ComponentSpec to a Step instance
Now we have in our hands a *ComponentSpec* that includes the component's name, inputs outputs and the container's implementation.

We need to instantiate a new Step object. But there are many reasons why the classic Kale's step class is not adequate for our case:

1. First and foremost a Step class wraps a Python function (an actual callable) with marshalling logic in order to run it later. But in our case, we don't have the actual function implementation. It lives in the component's docker image.

2. The Step class has the function's implementation as its centre of interest. Why? It does infer the component's inputs and outputs by inspecting the function's signature. However, regarding external components inputs and outputs will be retrieved from the component's specification, not the function's signature. Remember we don't have access to the component's function!

3. Finally, a new entrypoint is crafted that instructs Kale to run the corresponding step. This is not right cause for external plain KFP components we don't want to tamper with its entrypoint and change it. Same thing holds for the container's arguments.

For all of the above reasons we extend Kale's step implementation with a new Class called *ExternalStep* that will inherit the *BaseStep* class.

```
class ExternalStep(BaseStep):
    """Run an external KFP or Kale Component."""
```

Arguments:

- *component_spec* (kfp_structs.ComponentSpec): The component specification
- **args*: Positional arguments passed to BaseStep

- **kwargs*: Keyword arguments passed to BaseStep

We can instantiate this class with input parameter a component specification and use the returned object as a Kale step in a *@pipeline* function. Input and output dependencies (*ins* and *outs* for short) are retrieved from the respective ComponentSpec.inputs and ComponentSpec.outputs fields of the relevant KFP object.

The *entrypoint* is identified with the command field of the component's container implementation. In case of a Kale component we expand the entrypoint with the marshal path to wrap the step with marshalling logic. For KFP components we don't interfere with the container's implementation.

args is a list of string arguments for the entrypoint command of the container.

```
def load_component(...) -> Step:

    # Load ComponentSpec from YAML
    with open(component_path) as component:
        component_dict = load_yaml(component)
        component_spec = from_dict(component_dict)

    # Return the new step instance
    return ExternalStep(component_spec)
```

Figure 4.9: *From a YAML component definition to a Kale Step*

4.4 Testing

4.4.1 Unit Testing

Unit testing is a level of software testing where individual components of a software are tested. The purpose is to validate that each unit of the software performs as designed. A unit is the smallest testable part of any software. It usually has one or a few inputs and usually a single output. A testing script is included in Kubeflow Pipelines, which runs a bunch of unit tests on the DSL entities, using the Unittest [68] testing framework. We had to extend these by adding some corresponding tests on our newly introduced classes and methods.

4.4.2 End-to-End Testing

End-to-end testing is a technique used to test whether the flow of an application right from start to finish is behaving as expected. The purpose of performing end-to-end testing is to identify system dependencies and to ensure that the data integrity is maintained between various system components and systems. Kubeflow Pipelines project owns an infrastructure dedicated to performing end-to-end tests. This procedure includes compiling all components of the project, performing unit and functional testing on them and, lastly, compiling and running some pipelines expecting their success. These end-to-end tests now also run pipelines containing a subset of the features we introduced, those able to run on this specific infrastructure.

5 Conclusion

In this final chapter, we present a brief synopsis of our work assessing some principal points of the design process. Following that, we conclude by mentioning a few possible extensions and improvements that could be developed in the future.

5.1 Concluding Remarks

Our goal has been to study and experiment with container images and their properties and investigate how to exploit them to transparently reproduce an environment on Kubeflow. It is our firm belief that in the modern world Machine Learning Operations will thrive. Development, packaging and, of course, deployment undeniably can benefit from the advantages of containers and the convenience from the features provided by orchestration tools, like Kubernetes. Still, there is a big automation and reproducibility gap that doesn't enable the MLOps potential.

We developed a library responsible for building images on the fly. Then, with our mechanism in hand we introduced a seamless way to package the whole user's environment into a docker image, super fast within an unprivileged environment. Finally, we coupled this with producing Kubeflow Pipeline components. In the end, we provide one simple command for the complete metamorphosis of a Machine Learning Workflow step into a KFP standalone, reusable and reproducible component. Therefore, we may now conclude that our objective was met. Despite that, there was one more underlying goal: to effectively cooperate with developers from Arrikto and Kubeflow from all over the world as well as contribute to an open-source project of that scale. As far as

the second goal is concerned, the collaboration, it was a first time experience including constructive stress. Nevertheless, it was edifying and wonderful.

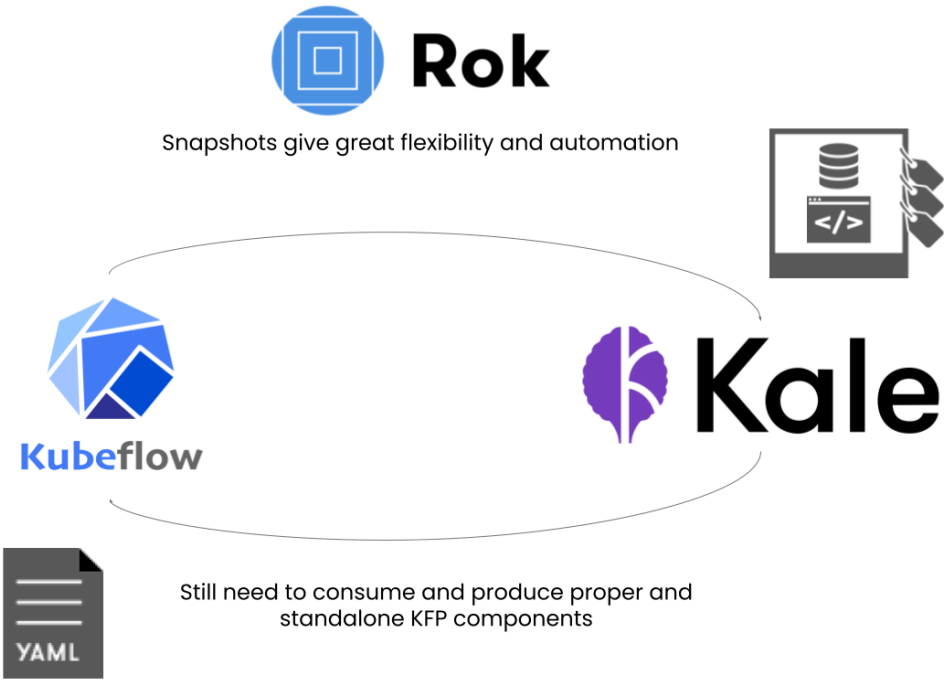


Figure 5.1: *Time to close the loop*

5.2 Future Work

Pretty much as expected, our build image mechanism along with our way to produce standalone KFP component has room for more development.

- **Build Image Mechanism**
  Currently we are forced to bring the base image's contents into the user's local filesystem. This means that we waste the user's space even temporarily. On the same page, the build of the new layer requires two operations: 1) Tar archive creation and 2) Compression of the tar archive. The first one literally copies the contents of the files we want to add in our archive and pastes into a separate file that will be represent our tar archive. Obviously, this means that we have a duplicate of the same files in our filesystem. The same happens with the compression of the archive. During the compress operation and for safety reasons (at least until it is

over) copies of the same files exist in the filesystem. Therefore, in the future and based on the customers' input we will consider provisioning an external volume with the required space to bring the base image and execute the whole procedure of the build image mechanism.

- **User Interface & Notebooks**

  For now users can only produce shareable and reproducible KFP components and run pipelines with our build image mechanism only from the Kale's CLI. One of our next steps is to integrate the whole mechanism in Jupyter Notebooks. This will include changes to the Kale's Lab extension in order to enable users configure their build image advanced settings and the contents of their image.

- **Component Marketplace**

  As we mentioned multiple times in this thesis we strive to create an ML community where data scientists and ML teams will be able to interact with each other and share their ideas with the click of a button. Undeniably, it is not that easy to share a yaml file (component's definition), thus our intention is to build a component registry, similar to pipeline definitions registry, that will use Kubeflow's database to store the definitions. They will show up in the Kubeflow's central dashboard and they will be accessed and used from the whole team immediately.

Bibliography

[1] Wikipedia – The Free Encyclopedia. Container. https://en.wikipedia.org/wiki/Container. accessed on the May 5th, 2022.

[2] RedHat. What's a linux container. https://www.redhat.com/en/topics/containers/whats-a-linux-container. accessed on the May 5th, 2022.

[3] Wikipedia – The Free Encyclopedia. Εικονικοποίηση - Βικιπαίδεια. https://el.wikipedia.org/wiki/Εικονικοποίηση. (Accessed on 06/30/2022).

[4] Linux manual pages. namespaces(7). https://man7.org/linux/man-pages/man7/namespaces.7.html. accessed on the May 5th, 2022.

[5] Docker Documentation. About storage drivers. https://docs.docker.com/storage/storagedriver/. accessed on the May 6th, 2022.

[6] Kirill Shirinkin. The tool that really runs your containers: deep dive into runc and oci specifications. https://mkdev.me/posts/the-tool-that-really-runs-your-containers-deep-dive-into-runc-and-oci-specifications. accessed on the May 6th, 2022.

[7] Runc. runc - open container initiative runtime. https://github.com/opencontainers/runc/. accessed on the May 6th, 2022.

[8] OCI. Conversion to oci runtime configuration. https://github.com/opencontainers/image-spec/blob/main/conversion.md. accessed on the May 6th, 2022.

[9] Bin Chen. Open container initiative (oci) specifications. https://www.alibabacloud.com/blog/open-container-initiative-oci-specifications_594397. accessed on the May 6th, 2022.

[10] Docker Documentation. Docker overview. https://docs.docker.com/get-started/overview/. accessed on the May 6th, 2022.

[11] Kubernetes Documentation Concepts. What is kubernetes? https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. accessed on the May 10th, 2022.

[12] Kubeflow Documentation. An introduction to kubeflow. https://www.kubeflow.org/docs/started/introduction/. accessed on the May 20th, 2022.

[13] Kale Documentation. Kale. https://github.com/kubeflow-kale/kale. accessed on the May 20th, 2022.

[14] Stefano Fioravanzo. Automating jupyter notebook deployments to kubeflow pipelines with kale. https://medium.com/kubeflow/automating-jupyter-notebook-deployments-to-kubeflow-pipelines-with-kale-a4ede38bea1f. accessed on the May 20th, 2022.

[15] Arrikto. Rok data management platform. https://www.arrikto.com/rok-data-management-platform/. accessed on the May 20th, 2022.

[16] Open Container Initiative. Oci image format specification. https://github.com/opencontainers/image-spec/blob/main/spec.md. accessed on the May 22th, 2022.

[17] Open Container Initiative. Oci image index specification. https://github.com/opencontainers/image-spec/blob/main/image-index.md. accessed on the May 22th, 2022.

[18] Open Container Initiative. Oci image index specification. https://github.com/opencontainers/image-spec/blob/main/image-index.md. accessed on the May 22th, 2022.

[19] Open Container Initiative. Oci image configuration specification. https://github.com/opencontainers/image-spec/blob/main/config.md. accessed on the May 22th, 2022.

[20] Kubeflow Documentation. Build a pipeline. https://www.kubeflow.org/docs/components/pipelines/sdk/build-pipeline/. accessed on the May 26th, 2022.

[21] Kubeflow Documentation. Building components. https://www.kubeflow.org/docs/components/pipelines/sdk/component-development/. accessed on the May 26th, 2022.

[22] Kale Documentation. Kale. https://docs.arrikto.com/user/kale/index.html. accessed on the May 26th, 2022.

[23] Python Documentation. os.path - common pathname manipulations. https://docs.python.org/3/library/os.path.html. accessed on the May 26th, 2022.

[24] Python Documentation. tarfile — read and write tar archive files. https://docs.python.org/3/library/tarfile.html#tarfile.open. accessed on the May 26th, 2022.

[25] Python Documentation. tarfile — read and write tar archive files. https://docs.python.org/3/library/tarfile.html#tarfile.TarFile.add. accessed on the May 26th, 2022.

[26] Python Documentation. Python built-in functions. https://docs.python.org/3/library/functions.html#open. accessed on the May 26th, 2022.

[27] Python Documentation. hashlib — secure hashes and message digests. https://docs.python.org/3/library/hashlib.html. accessed on the May 26th, 2022.

[28] Python Documentation. gzip - support for gzip files. https://docs.python.org/3/library/gzip.html. accessed on the May 26th, 2022.

[29] Dcoker Documentation. Docker registry http api v2. https://docs.docker.com/registry/spec/api/. accessed on the June 18th, 2022.

[30] Linux manual pages. chroot(1). https://man7.org/linux/man-pages/man1/chroot.1.html. accessed on the May 5th, 2022.

[31] Linux manual pages. chroot(2). https://man7.org/linux/man-pages/man2/chroot.2.html. accessed on the May 5th, 2022.

[32] Sascha Grunert. Demystifying containers - part i: Kernel space. https://medium.com/@saschagrunert/demystifying-containers-part-i-kernel-space-2c53d6979504. accessed on the May 5th, 2022.

[33] Linux manual pages. clone(2). https://man7.org/linux/man-pages/man2/clone.2.html. accessed on the May 5th, 2022.

[34] Linux manual pages. unshare(2). https://man7.org/linux/man-pages/man2/unshare.2.html. accessed on the May 5th, 2022.

[35] Linux manual pages. setns(2). https://man7.org/linux/man-pages/man2/setns.2.html. accessed on the May 5th, 2022.

[36] Linux manual pages. nsenter(1). https://man7.org/linux/man-pages/man1/nsenter.1.html. accessed on the May 5th, 2022.

[37] Michael Kerrisk. Mount namespaces and shared subtrees. https://lwn.net/Articles/689856/. accessed on the May 5th, 2022.

[38] Michael Kerrisk. Namespaces in operation, part 1: namespaces overview. https://lwn.net/Articles/531114/#series_index. accessed on the May 5th, 2022.

[39] Linux manual pages. veth(4). https://man7.org/linux/man-pages/man4/veth.4.html. accessed on the May 5th, 2022.

[40] Ivan Velichko. Container networking is simple! https://iximiuz.com/en/posts/container-networking-is-simple/. accessed on the May 5th, 2022.

[41] Linux manual pages. cgroups(7). https://man7.org/linux/man-pages/man7/cgroups.7.html. accessed on the May 5th, 2022.

[42] Linux manual pages. capabilities(7). https://man7.org/linux/man-pages/man7/capabilities.7.html. accessed on the May 6th, 2022.

[43] Kubernetes Documentation Concepts. What is kubernetes? https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. accessed on the May 10th, 2022.

[44] Kubernetes Documentation Concepts. Understanding kubernetes objects. https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/. accessed on the May 10th, 2022.

[45] Kubernetes Documentation Concepts. Kubernetes controllers. https://kubernetes.io/docs/concepts/architecture/controller/. accessed on the May 10th, 2022.

[46] Kevin Sookocheff. A guide to the kubernetes networking model. https://sookocheff.com/post/kubernetes/understanding-kubernetes-networking-model/. accessed on the May 15th, 2022.

[47] Kubernetes Documentation Concepts. Services, load balancing, and networking. https://kubernetes.io/docs/concepts/services-networking/. accessed on the May 15th, 2022.

[48] Kubernetes Documentation Concepts. Kubernetes service. https://kubernetes.io/docs/concepts/services-networking/service/. accessed on the May 15th, 2022.

[49] Kubernetes Documentation Concepts. Kubernetes ingress. https://kubernetes.io/docs/concepts/services-networking/ingress/. accessed on the May 15th, 2022.

[50] Kubernetes Documentation Concepts. Dns for services and pods. https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/. accessed on the May 15th, 2022.

[51] Kubeflow Documentation. An overview of kubeflow's architecture. https://www.kubeflow.org/docs/started/architecture/. accessed on the May 20th, 2022.

[52] Linux manual pages. jq(1). https://www.systutorials.com/docs/linux/man/1-jq/. accessed on the May 22th, 2022.

[53] Open Container Initiative. Oci image layer filesystem changeset. https://github.com/opencontainers/image-spec/blob/main/layer.md. accessed on the May 22th, 2022.

[54] Podman. Manage pods, containers, and container images. https://podman.io/. accessed on the June 5th, 2022.

[55] Linux manual pages. tar(1). https://man7.org/linux/man-pages/man1/tar.1.html. accessed on the May 22th, 2022.

[56] Linux manual pages. ls(1). https://man7.org/linux/man-pages/man1/ls.1.html. accessed on the May 22th, 2022.

[57] Linux manual pages. mv(1). https://man7.org/linux/man-pages/man1/mv.1.html. accessed on the May 22th, 2022.

[58] Linux manual pages. tar(1). https://man7.org/linux/man-pages/man1/tar.1.html. accessed on the May 22th, 2022.

[59] Argo Workflow. Argo workflows - the workflow engine for kubernetes. https://argoproj.github.io/argo-workflows/. accessed on the June 5th, 2022.

[60] NetworkX Documentation. Digraph—directed graphs with self loops. https://networkx.org/documentation/stable/reference/classes/digraph.html. accessed on the May 26th, 2022.

[61] Chiradeep BasuMallick. Rootless containers are generating a buzz in the industry. https://www.spiceworks.com/it-security/cloud-security/articles/rootless-containers-are-generating-a-buzz-in-the-industry-heres-why/. (Accessed on 06/30/2022).

[62] The Kubeflow Authors. Poddefault. https://github.com/kubeflow/kubeflow/blob/master/components/admission-webhook/README.md. (Accessed on 06/30/2022).

[63] Dcoker Documentation. Image manifest v2, schema 2. https://docs.docker.com/registry/spec/manifest-v2-2/. accessed on the May 26th, 2022.

[64] Python Documentation. argparse — parser for command-line options, arguments and sub-commands. `https://docs.python.org/3/library/argparse.html`. accessed on the June 18th, 2022.

[65] Python Documentation. `https://docs.python.org/3/library/inspect.html`. accessed on the June 18th, 2022.

[66] YAML Documentation. Yaml ain't markup language (yaml™) version 1.2. `https://yaml.org/spec/1.2.2/`. accessed on the May 26th, 2022.

[67] Stack Overflow. In python, how can you load yaml mappings as ordered-dicts? `https://stackoverflow.com/questions/5121931/in-python-how-can-you-load-yaml-mappings-as-ordereddicts/21912744#21912744`. accessed on the June 18th, 2022.

[68] Python Documentation. unittest — unit testing framework. `https://docs.python.org/3/library/unittest.html`. accessed on the June 18th, 2022.