Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων
Και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης Και Υπολογιστων

Σχεδίαση και Υλοποίηση Μηχανισμού Διαχείρισης Δεδομένων για Ροές Εργασίας Μηχανικής Μάθησης στον Κυβερνήτη

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ηλίας Α. Κατσακιώρης

Εργαστήριο Υπολογιστικών Συστημάτων
Αθήνα, Ιούλιος 2019

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

Σχεδίαση και Υλοποίηση Μηχανισμού Διαχείρισης Δεδομένων για Ροές Εργασίας Μηχανικής Μάθησης στον Κυβερνήτη

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ηλίας Α. Κατσακιώρης

Επιβλέπων Καθηγητής: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 11η Ιουλίου 2019.

. . . . .	. . . . .	. . . . .
Νεκτάριος Κοζύρης	Νικόλαος Παπασπύρου	Γεώργιος Γκούμας
Καθηγητής ΕΜΠ	Αν. Καθηγητής ΕΜΠ	Επ. Καθηγητής ΕΜΠ

Εργαστήριο Υπολογιστικών Συστημάτων
Αθήνα, Ιούλιος 2019

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

Design and Implementation of Advanced Data Management for Machine Learning Workflows on Kubernetes

DIPLOMA THESIS

Ilias A. Katsakioris

Computing Systems Laboratory

Athens, July 2019

. . . . . .

Ηλίας Α. Κατσακιώρης
Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

Περίληψη

Η μηχανική μάθηση κερδίζει μέρα με τη μέρα όλο και περισσότερο έδαφος, αφού λύνει περίπλοκα και κρίσιμα προβλήματα της καθημερινής ζωής μας εκμεταλλευόμενη σχετικά δεδομένα. Το Kubernetes είναι ένας ενορχηστρωτής κατανεμημένου φόρτου εργασίας που τρέχει σε συστοιχίες υπολογιστών και κέντρα δεδομένων. Το Kubeflow έρχεται και συγχωνεύει αυτές τις δύο έννοιες ως η καθιερωμένη πλατφόρμα για πραγματοποίηση μηχανικής μάθησης στο Kubernetes. Μια ροή εργασίας μηχανικής μάθησης είναι μια αυτοματοποιημένη και επαναληπτική ροή εργασίας πολλαπλών βημάτων που κάνει τη μηχανική μάθηση δυνατή. Το Kubeflow διαχειρίζεται τέτοιες ροές εργασίας μέσω του Kubeflow Pipelines.

Κατά τη διάρκεια αυτής της διπλωματικής εργασίας γίναμε μέλη της κοινότητας του Kubeflow και ήρθαμε σε επαφή με ερευνητές, που αξιοποιούν το πεδίο της μηχανικής μάθησης, μέσω του Slack του Kubeflow και των συναντήσεων της κοινότητας. Διερευνώντας τις απαιτήσεις του πεδίου ανακαλύψαμε πως η προηγμένη διαχείριση δεδομένων εκλείπει από το Kubeflow Pipelines. Η εκτέλεση κάποιου Kubeflow Pipelines τοπικά ήταν δύσκολη ενώ η αναπαραγωγιμότητα ήταν απλώς μια χίμαιρα. Έτσι, σχεδιάσαμε μια επέκταση για να αντιμετωπίσουμε αυτό το εμπόδιο. Αρχικά, δημοσιεύσαμε ένα κείμενο τεχνικών προδιαγραφών [1] και συλλέξαμε τις απόψεις των προγραμματιστών αλλά και της υπόλοιπης κοινότητας. Εν συνεχεία, υλοποιήσαμε τον προκύπτον σχεδιασμό και κάναμε μια σχετική επίδειξη στη συνάντηση της κοινότητας την 26η Φεβρουαρίου, 2019 [2]. Έπειτα, υποβάλαμε μια αίτηση προσθήκης της επέκτασης στο επίσημο πρότζεκτ του Kubeflow Pipelines [3]. Ως εκ τούτου, διευκολύνθηκε η κατανόηση του σχεδιασμού και της υλοποίησής μας, ενώ η κοινότητα είχε τη δυνατότητα να τη δοκιμάσει. Τελικά, μετά από ένα σύνολο αναθεωρήσεων, επανασχεδιασμών και βελτιώσεων η υλοποίηση του τελικού σχεδιασμού μας ενσωματώθηκε και είναι πλέον μέρος του επίσημου πρότζεκτ.

Συνολικά, η επέκτασή μας χρησιμοποιείται ευρέως από οργανισμούς, όπως IBM και Seldon, αλλά και από ιδιωτικούς χρήστες, και λαμβάνει όλο και περισσότερη προσοχή καθημερινά. Ο στόχος που πετύχαμε σε αυτήν την εργασία δεν ήταν μόνο να λύσουμε ένα πρόβλημα, αλλά να το κάνουμε με τέτοιο τρόπο ώστε οι τελικοί χρήστες να μπορούν να ωφεληθούν από αυτό.

Λέξεις-Κλειδιά

advanced data management, data-intensive computing, machine learning, workflows, pipelines, compiler, containers, Kubernetes, Kubeflow, Kubeflow Pipelines, Argo

[1] `https://github.com/kubeflow/pipelines/issues/801`

[2] `https://drive.google.com/a/kubeflow.org/file/d/1PUy3BGrFJ43inyPK9QAV9PCT9pwZfE1D`

[3] `https://github.com/kubeflow/pipelines/pull/926`

Abstract

Machine learning (ML) is increasingly gaining traction day by day, since it gets to solve complex and critical problems in our everyday lives by exploiting related data. Kubernetes is a distributed workload orchestrator running over clusters and data centers. Kubeflow comes to merge these two concepts by being the de facto used platform for running machine learning on Kubernetes. A machine learning workflow is a multi-step, automated and iterative workflow rendering ML possible. Kubeflow manages such workflows as instances of Kubeflow Pipelines.

During this thesis we became part of the Kubeflow community and we got in touch with data scientists through Kubeflow's Slack workspace and Community Meetings. By performing requirements gathering we found out that Kubeflow Pipelines lack advanced data management. Running Kubeflow Pipelines on-prem was challenging and reproducibility was only a pipe dream. Therefore, we designed an extension to tackle that hindrance. Initially, we published a design document [4] and collected feedback from developers and the rest of the community. Subsequently, we implemented the resulting design and demonstrated it during the Community Meeting of February 26, 2019 [5]. Then, we created a pull request adding the extension to the official Kubeflow Pipelines project[6]. Hence, the understanding of our design and implementation was easier, while the community was also able to try it out. Finally, after a number of reviews, refactoring and enhancements our final design and implementation was merged and is now part of the official repository.

All in all, the feature extension is widely used by organizations, such as IBM and Seldon, as well as other private users, and is getting more and more attention every day. The met objective of this thesis was to not only solve a problem, but also do it in such manner that end-users can actually benefit from it.

Keywords

advanced data management, data-intensive computing, machine learning, workflows, pipelines, compiler, containers, Kubernetes, Kubeflow, Kubeflow Pipelines, Argo

[4] https://github.com/kubeflow/pipelines/issues/801
[5] https://drive.google.com/a/kubeflow.org/file/d/1PUy3BGrFJ43inyPK9QAV9PCT9pwZfE1D
[6] https://github.com/kubeflow/pipelines/pull/926

Πρόλογος

Στο σημείο αυτό, θα ήθελα να ευχαριστήσω τους ανθρώπους που έπαιξαν καθοριστικό ρόλο στην εκπόνηση της διπλωματικής, αλλά και ευρύτερα στην πορεία μου σε ακαδημαϊκό επίπεδο. Πιο συγκεκριμένα, τον επιβλέποντα της διπλωματικής μου, Καθηγητή Νεκτάριο Κοζύρη, ο οποίος με εισήγαγε στον κόσμο των υπολογιστικών συστημάτων και μου μετέφερε το ενδιαφέρον του σε αυτόν μέσω των διαλέξεών του.

Εν συνεχεία, 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. Θέλω να ευχαριστήσω, επίσης, τον Ιωάννη Ανδρουλιδάκη για τον χρόνο που μου αφιέρωσε ώστε να ανταλλάξουμε σκέψεις και ιδέες σχετικά με ζητήματα που ανέκυψαν κατά τη διάρκεια αυτής της διπλωματικής.

Οφείλω, ακόμα, ένα μεγάλο ευχαριστώ στον αδελφό και συνάδελφό μου Χρήστο, ο οποίος με το πάθος και τις γνώσεις του με ωθούσε συνεχώς προς την εξέλιξή μου, τόσο ως μηχανικός αλλά και ως άτομο.

Τέλος, θα ήθελα να ευχαριστήσω από τα βάθη της καρδιάς μου τους γονείς μου για την αγάπη και τη στήριξη που μου παρέχουν, καθώς και όλους τους κοντινούς μου ανθρώπους που ομορφαίνουν την καθημερινότητά μου.

Ηλίας Κατσακιώρης
Ιούλιος 2019

Contents

List of figures

Ελληνικό Κείμενο

0.1 Εισαγωγή

0.1.1 Διατύπωση Προβλήματος

Οι διεργασίες μηχανικής μάθησης (ΜΜ) [1] καταναλώνουν και παράγουν μεγάλο όγκο δεδομένων. Για την ακρίβεια, ανάλογα το πρόβλημα, όσο περισσότερα είναι τα δεδομένα τόσο καλύτερα είναι τα αποτελέσματα. Για παράδειγμα, για να αντιμετωπίσουμε κάποια από τα "δύσκολα" προβλήματα, όπως αυτά με τα οποία ασχολείται η βαθειά μάθηση [7], μπορεί να χρειαστεί να αναλυθούν εκατομμύρια ή ακόμη και δεκάδες εκατομμύρια παραδείγματα. Η διαχείριση αυτών των δεδομένων (δηλαδή η αποθήκευση, η λήψη εκδόσεών τους, κ.α.) είναι πρόκληση για τους ερευνητές και τους παρόχους πλατφορμών. Παρόλα αυτά, είναι μια δυσκολία που πρέπει να υπερνικήσουμε με κάθε τρόπο.

Ένα διάσημο εργαλείο μηχανικής μάθησης, αφοσιωμένο να κάνει το deployment[8] ροών εργασίας μηχανικής μάθησης στο Kubernetes απλό, φορητό και κλιμακώσιμο είναι το Kubeflow[2]. Το Kubeflow Pipelines (KFP) [3] είναι εκείνο το συστατικό του Kubeflow που είναι υπεύθυνο για τη συνολική ενορχήστρωση διοχετεύσεων[9]. Μια διοχέτευση είναι η περιγραφή μιας ροής εργασίας ΜΜ, συμπεριλαμβανομένων όλων των συστατικών στη ροή καθώς και τον τρόπο με τον οποίο σχετίζονται σε μορφή γράφου.

Στο Kubeflow Pipelines δεν υπάρχει τρόπος για πραγματοποίηση εντατικής διαχεί-

[7] Με τον ελληνικό όρο "βαθειά μάθηση" αναφερόμαστε στον αγγλικό όρο "deep learning"

[8] Ελλείψει κατάλληλου όρου, γίνεται χρήση του αγγλικού "deployment" αντί του εν προκειμένω αδόκιμου ελληνικού όρου "παράταξη"

[9] Με τον ελληνικό όρο "διοχέτευση" αναφερόμαστε στον αγγλικό όρο "pipeline"

ρισης δεδομένων [10]. Για να είμαστε ακριβείς, η διαθέσιμη διαχείριση δεδομένων στο χρήστη είναι πολύ περιοριστική σε ό,τι αφορά τόσο το μέγεθος όσο και τον τύπο τους, ενώ δεν υποστηρίζονται εκδόσεις δεδομένων [11].

Θα αναλύσουμε βαθύτερα την τρέχουσα κατάσταση σε επόμενο σημείο του κειμένου.

0.1.2 Κίνητρα

Στο [4] μπορούμε να βρούμε έναν ακριβή ορισμό της μηχανικής μάθησης:

> Η μηχανική μάθηση είναι μια εφαρμογή της τεχνητής νοημοσύνης (ΤΝ) που παρέχει σε συστήματα τη δυνατότητα να μαθαίνουν αυτόματα και να βελτιώνονται από την εμπειρία χωρίς να είναι ρητά προγραμματισμένα. Η μηχανική μάθηση εστιάζει στην ανάπτυξη προγραμμάτων που έχοντας πρόσβαση σε δεδομένα και τα χρησιμοποιούν για να μάθουν.
>
> Η διαδικασία της εκμάθησης ξεκινά με παρατηρήσεις ή δεδομένα, όπως παραδείγματα, άμεσες εμπειρίες, ή εντολή, προκειμένου να αναζητηθούν πρότυπα[12] στα δεδομένα και να λαμβάνονται καλύτερες αποφάσεις στο μέλλον βασιζόμενες σε παραδείγματα που παρέχουμε. Ο πρωταρχικός στόχος είναι να επιτραπεί στους υπολογιστές να μαθαίνουν αυτόματα χωρίς την ανθρώπινη παρέμβαση ή βοήθεια και να προσαρμόζουν τις ενέργειές τους αναλόγως.

Οι αλγόριθμοι ΜΜ παράγουν μοντέλα που είναι άρρηκτα συνδεδεμένα με μια συνάρτηση κόστους. Αυτή η συνάρτηση περιγράφει την ακρίβεια του μοντέλου, την αποτελεσματικότητά του, και έτσι, τέτοιοι αλγόριθμοι αναζητούν ένα μοντέλο που ελαχιστοποιεί βέλτιστα τη συνάρτηση κόστους.

Δυστυχώς, η κωδικοποίηση κάποιων διεργασιών μπορεί να γίνει πολύ περίπλοκη για τους ανθρώπους. Η ανάλυσή τους και η λύση τους μπορεί να είναι μη πρακτική ή ακόμα και αδύνατη. Οι αλγόριθμοι μηχανικής μάθησης μπορούν να σώσουν τέτοιες περιπτώσεις. Αυτοί οι αλγόριθμοι μπορούν να τροφοδοτηθούν με πολλά δεδομένα,

[10] Με τον ελληνικό όρο "εντατική διαχείριση δεδομένων" αναφερόμαστε στον αγγλικό όρο "intensive data management"

[11] Με τον ελληνικό όρο "εκδόσεις δεδομένων" αναφερόμαστε στον αγγλικό όρο "data versioning"

[12] Με τον ελληνικό όρο "πρότυπο" αναφερόμαστε στον αγγλικό όρο "pattern"

να τα εξερευνήσουν και να αναζητήσουν ένα μοντέλο που καταφέρνει οτιδήποτε έχει προγραμματίσει ο ερευνητής.

Η μηχανική μάθηση έχει πάρα πολλές εφαρμογές και υπάρχει στην καθημερινότητά μας περισσότερο από όσο μπορεί να πιστεύουμε. Η κατηγοριοποίηση των μηνυμάτων ηλεκτρονικού ταχυδρομείου, τα νέα στο Facebook, συστήματα φωνή-σε-κείμενο καθώς και όλοι οι τύποι διαδικτυακών διαφημίσεων είναι απλώς μερικές από αυτές.

Τα αυτοκίνητα ή τα ελικόπτερα που οδηγούν και πετούν μόνα τους είναι κάποια πολύ πιο εντυπωσιακά επιτεύγματα της μηχανικής μάθησης. Όμως, είναι ταυτόχρονα και παραδείγματα που έχουν αποτύχει σε κρίσιμες καταστάσεις, όπως στα [5], και αναδεικνύουν την ανάγκη για περαιτέρω βελτίωση των αποτελεσμάτων αυτού του πεδίου της τεχνητής νοημοσύνης.

Τα μοντέλα μηχανικής μάθησης δεν είναι τίποτα παραπάνω από μαθηματικές συναρτήσεις που παίρνουν χαρακτηριστικά ως είσοδο, παράγουν προβλέψεις σαν έξοδο και μαθαίνουν πως να ταιριάξουν καλύτερα προβλέψεις σε πρότυπα που παρατηρούνται από τα δεδομένα εκμάθησης. Επομένως, ένα καίριο βήμα κατά την ανάπτυξη μιας εφαρμογής ΜΜ είναι να καταλάβουμε τα απαιτούμενα δεδομένα, καθώς και την ποσότητά τους.

Όπως έχουμε ήδη αναφέρει, τα μοντέλα δεν είναι 100% ορθά, αλλά είναι μάλλον "καλύτερες εικασίες" δεδομένου του μεγέθους των δεδομένων που έχει δει το μοντέλο. Το σύνολο των γνωστών εισόδων και εξόδων που ο αναλυτής δεδομένων [13] χρησιμοποιεί για να "εκπαιδεύσει" το μοντέλο καλείται **"σύνολο εκπαίδευσης"**[14]. Το σύνολο εκπαίδευσης μπορεί να χρησιμοποιηθεί σαν είσοδος σε πολλούς αλγορίθμους, ή ακόμα και στον ίδιο αλγόριθμο με διαφορετικές ρυθμίσεις, για την παραγωγή πολλαπλών μοντέλων.

Το επόμενο βήμα είναι ο έλεγχος της απόδοσης των προαναφερθέντων "εκπαιδευμένων" μοντέλων. Αυτό επιτυγχάνεται τροφοδοτώντας κάθε μοντέλο με ένα νέο σύνολο δεδομένων [15], το επονομαζόμενο **"σύνολο επικύρωσης"**[16]. Τα αποτελέσματα της επικύρωσης οδηγούν στην εκλογή νικητή: την επιλογή του ακριβέστερου μοντέλου.

Στο τέλος, είναι η ώρα να αξιοποιήσουμε το καλύτερο μοντέλο. Δεν χρειάζεται ένας

[13] Με τον ελληνικό όρο "αναλυτής δεδομένων" αναφερόμαστε στον αγγλικό όρο "data analyst"
[14] Με τον ελληνικό όρο "σύνολο εκπαίδευσης" αναφερόμαστε στον αγγλικό όρο "training set"
[15] Με τον ελληνικό όρο "σύνολο δεδομένων" αναφερόμαστε στον αγγλικό όρο "dataset"
[16] Με τον ελληνικό όρο "σύνολο επικύρωσης" αναφερόμαστε στον αγγλικό όρο "validation set"

αλγόριθμος ΜΜ για να προβλέψουμε πως υπάρχει ένα σύνολο δεδομένων που θέλουμε να εξερευνήσουμε. Ονομάζεται **"σύνολο εξέτασης"**[17]. Όλη η παραπάνω ροή εργασίας πραγματοποιείται για αυτό ακριβώς το σημείο: το **"σερβίρισμα"**[18]. Κατά τη διάρκεια αυτής της φάσης, πραγματοποιείται η ουσιαστική ανάλυση δεδομένων.

Λαμβάνοντας όλα τα παραπάνω υπόψιν, ένα πόρισμα εξάγεται: οι ροές εργασίας μηχανικής μάθησης έχουν έντονη την παρουσία μεγάλου όγκου δεδομένων και, ως εκ τούτου, απαιτούν την προηγμένη διαχείριση αυτών.

Είναι γεγονός πως τα δεδομένα αυξάνονται συνεχώς και μαζί με αυτά αυξάνονται και οι απαιτήσεις στις υποδομές για υπολογισμούς. Αυτό έχει ως αποτέλεσμα πολλοί χρήστες να στρέφονται σε λύσεις όπως Υποδομή-ως-Υπηρεσία[19], Πλατφόρμα-ως-Υπηρεσία[20], Λογισμικό-ως-Υπηρεσία[21] ή οποιονδήποτε άλλο τύπο υπηρεσιών στο Cloud[7]. Φαίνεται πως ζούμε την αυγή της εποχής του Όλα-ως-Υπηρεσία[22].

Αυτές οι υπηρεσίας εκτελούνται σε κατανεμημένα συστήματα. Αναπόφευκτα, εμπλέκεται πολλή πολυπλοκότητα. Η ενορχήστρωση υπηρεσιών και διεργασιών, η διαχείριση του storage[23] και η αντιμετώπιση του προβλήματος του statefulness[24] εν γένει μπορεί να είναι μεγάλη πρόκληση.

Ο συγχρονισμός των δεδομένων, η ασφάλεια, η διαϋπηρεσιακή επικοινωνία, ο διαμοιρασμός του φόρτου [25], η υψηλή διαθεσιμότητα και η ανοχή σε σφάλματα είναι απλώς κάποια από τα μεγάλα προβλήματα από τα οποία πάσχουν σύγχρονα κατανεμημένα συστήματα και εφαρμογές.

Λύσεις για τέτοια προβλήματα πρώτα εγείρονται μέσα από τη βιομηχανία λογισμικού. Τέσσερα χρόνια πριν, η Google σχεδίασε και υλοποίησε ένα ανοιχτού κώδικα σύστημα

[17] Με τον ελληνικό όρο "σύνολο εξέτασης" αναφερόμαστε στον αγγλικό όρο "test set"

[18] Με τον ελληνικό όρο "σερβίρισμα" αναφερόμαστε στον αγγλικό όρο "serving"

[19] Με τον ελληνικό όρο "Υποδομή-ως-Υπηρεσία" αναφερόμαστε στον αγγλικό όρο "Infrastructure-as-a-Service"

[20] Με τον ελληνικό όρο "Πλατφόρμα-ως-Υπηρεσία" αναφερόμαστε στον αγγλικό όρο "Platform-as-a-Service"

[21] Με τον ελληνικό όρο "Λογισμικό-ως-Υπηρεσία" αναφερόμαστε στον αγγλικό όρο "Software-as-a-Service"

[22] Με τον ελληνικό όρο "Όλα-ως-Υπηρεσία" αναφερόμαστε στον αγγλικό όρο "Everything-as-a-Service"

[23] Ελλείψει κατάλληλου όρου, γίνεται χρήση του αγγλικού "storage" αντί του εν προκειμένω αδόκιμου ελληνικού όρου "αποθήκευση"

[24] Ελλείψει κατάλληλου όρου, γίνεται χρήση του αγγλικού "statefulness" αντί του εν προκειμένω αδόκιμου ελληνικού όρου "κρατικότητα"

[25] Με τον ελληνικό όρο "διαμοιρασμός φόρτου" αναφερόμαστε στον αγγλικό όρο "load-balancing"

για την ενορχήστρωση περιεκτών[26] που ονομάζεται Kubernetes[8]. Το πρότζεκτ υποστηρίζεται πλέον από το Cloud Native Computing Foundation (CNCF) [9], και στοχεύει στο να κάνει το deployment των περιεκτικοποιημένων [27] μικροϋπηρεσιών[10] ευκολότερο. Παρόλα αυτά, οι περιπλοκότητες των κατανεμημένων συστημάτων χρειάζονται αρκετό χρόνο ακόμα για να αντιμετωπιστούν πλήρως.

Συνεπώς, πολλοί οργανισμοί αναπτύσσουν τις πλατφόρμες τους, ή επεκτείνουν τα πρότζεκτ τους κατάλληλα, ώστε να πραγματοποιούν μηχανική μάθηση στο Kubernetes. Κατά πλειοψηφία, αυτό σημαίνει να επεκτείνουν το ίδιο το Kubernetes [11]. Θα αναλύσουμε τι είναι το Kubernetes και πώς λειτουργεί αργότερα σε αυτό το κείμενο.

Σε αυτή τη διπλωματική εργασία εστιάζουμε στην κατάσταση των δεδομένων των ροών εργασίας μηχανικής μάθησης που εκτελούνται στο Kubernetes, καθώς επίσης και στη διαχείριση και τη λήψη εκδόσεών τους.

0.1.3 Υπάρχουσες Λύσεις

Έχουν πραγματοποιηθεί πολλαπλές προσπάθειες από πρότζεκτ και προϊόντα να αντιμετωπίσουν το πρόβλημα του deployment ροών εργασίας μηχανικής μάθησης στον Κυβερνήτη. Παρά το στόχο τους να κάνουν απλή και εύκολη τη χρήση τους, δεν καταφέρνουν ελκυστικό αποτέλεσμα για τους επιστήμονες δεδομένων [28]. Συνεπώς, ακόμα και απλές λειτουργίες απαιτούν αρκετή μελέτη άλλων πεδίων από το χρήστη.

Εδώ θα παρουσιάσουμε συνοπτικά τις πιο αξιοσημείωτες λύσεις.

- **Pachyderm**

  Το Pachyderm[12] είναι σχεδιασμένο να επιτρέπει βιώσιμες ροές εργασίας επιστήμης δεδομένων [29] μέσω ενός συστήματος ανεξάρτητου από τη γλώσσα [30] για λήψη εκδόσεων δεδομένων και διοχέτευση αυτών. Επιτρέπει στο χρήστη

[26] Ελλείψει αντίστοιχου όρου στην ελληνική βιβλιογραφία, χρησιμοποιείται ο ελληνικός όρος "περιέκτης" για να εκφράσει την έννοια που εκφράζεται με τον αγγλικό όρος "container".

[27] Ελλείψει αντίστοιχου ελληνικού όρου, χρησιμοποιείται ο ενδεχομένως αδόκιμος ελληνικός όρος "περιεκτικοποιημένος" για να εκφράσει την έννοια του αγγλικού όρου "containerized"

[28] Με τον ελληνικό όρο "επιστήμονας δεδομένων" αναφερόμαστε στον αγγλικό όρο "data scientist"

[29] Με τον ελληνικό όρο "επιστήμη δεδομένων" αναφερόμαστε στον αγγλικό όρο "data science"

[30] Με τον ελληνικό όρο "σύστημα ανεξάρτητου από τη γλώσσα" αναφερόμαστε στον αγγλικό όρο "language-agnostic system"

να διαχειρίζεται διοχετεύσεις δεδομένων πολλών σταδίων ενώ διατηρεί πλήρη αναπαραγωγή[31].

Σχετικά με ένα μέρος του στόχου μας, τη λήψη στιγμιοτύπων δεδομένων, το Pachyderm χρησιμοποιεί ένα εργαλείο που μοιάζει με το Git[13] ως σύστημα ελέγχου εκδόσεων [32] για αυτό το σκοπό. Ο χρήστης μπορεί να κοιτάξει πίσω στο χρόνο και να δει πώς έμοιαζαν τα δεδομένα εκπαίδευσης όταν ένα συγκεκριμένο μοντέλο δημιουργήθηκε, ή πως τα αποτελέσματα μεταβάλλονται με την πάροδο του χρόνου.

Το Pachyderm, όμως, έχει και μειονεκτήματα. Η χρονική καθυστέρηση [33], καθώς εξαρτάται από εξωτερικές αποθήκες αντικειμένων [34] που το κάνουν να εξαρτάται από την ταχύτητα του δικτύου, και η σύνταξη που απαιτεί, αφού χρειάζεται πολλές γραμμές κώδικα ακόμα και για μικρές λειτουργίες. Ακόμη, χρειάζεται πολλή προσπάθεια από τον ερευνητή ώστε να μάθει ένα νέο εργαλείο και τη χρήση του.

- **Argo**

  Το Argo[16] είναι μια μηχανή ροών εργασίας [35]. Είναι μια επέκταση της Kubernetes συστοιχίας[36] που κάνει δυνατή την εκτέλεση ροών εργασίας. Ο χρήστης υποβάλλει έναν ορισμό της ροής εργασίας σε αρχείο YAML και έπειτα το Argo είναι υπεύθυνο να εκκινήσει τις διεργασίες σε κατάλληλη σειρά και να περιμένει την ολοκλήρωσή τους. Παρέχει, ακόμη, μια διεπαφή γραμμής εντολών [37] καθώς επίσης και μια βασική διεπαφή χρήστη [38] για την απεικόνιση των ροών εργασίας.

  Το Kubeflow Pipelines χρησιμοποιεί το Argo ως τη μηχανή ροών εργασίας του. Το Κιτ Ανάπτυξης Λογισμικού (ΚΑΛ) [39] μεταγλωττίζει τον πηγαίο κώδικα του χρήστης σε Argo δήλωση.

---

[31] Με τον ελληνικό όρο "αναπαραγωγή" αναφερόμαστε στον αγγλικό όρο "reproducibility"

[32] Με τον ελληνικό όρο "σύστημα ελέγχου εκδόσεων" αναφερόμαστε στον αγγλικό όρο "version control system"

[33] Με τον ελληνικό όρο "χρονική καθυστέρηση" αναφερόμαστε στον αγγλικό όρο "latency"

[34] Με τον ελληνικό όρο "αποθήκη αντικειμένων" αναφερόμαστε στον αγγλικό όρο "object store"

[35] Με τον ελληνικό όρο "μηχανή ροών εργασίας" αναφερόμαστε στον αγγλικό όρο "workflow engine"

[36] Με τον ελληνικό όρο "συστοιχία" αναφερόμαστε στον αγγλικό όρο "cluster"

[37] Με τον ελληνικό όρο "διεπαφή γραμμής εντολών" αναφερόμαστε στον αγγλικό όρο "command line interface"

[38] Με τον ελληνικό όρο "διεπαφή χρήστη" αναφερόμαστε στον αγγλικό όρο "user interface"

[39] Με τον ελληνικό όρο "Κιτ Ανάπτυξης Λογισμικού" ή "ΚΑΛ" αναφερόμαστε στον αγγλικό όρο "Software Development Kit" ή "SDK"

Το Argo έχει ευρεία λειτουργικότητα, την οποία αξιοποιούμε σε αυτήν την εργασία. Μολαταύτα, οφείλουμε να σημειώσουμε μερικά μειονεκτήματα. Απαιτείται από το χρήστη να έχει πρόσβαση στην Kubernetes συστοιχία, που σημαίνει πως ο χρήστης θα πρέπει να γνωρίζει τόσο Argo όσο και Kubernetes. Θεωρούμε πως ένας μηχανικός δεδομένων [40] θα έπρεπε να εργάζεται με αφηρημένο τρόπο από τα Kubernetes και Argo.

- **Η Κατάσταση στο Kubeflow Pipelines**

  Το πρότζεκτ Kubeflow Pipelines είναι σε πολύ πρώιμη κατάσταση, κι όμως έχει απήχηση σε πολλούς αναλυτές δεδομένων. Αυτό είναι σημαντικό κίνητρο για τα μέλη του οργανισμού του Kubeflow και όσους συνεισφέρουν στο πρότζεκτ να συνεχίσουν την επέκταση και την ανάπτυξη λειτουργιών.

  Όπως έχει ήδη σημειωθεί, το ΚΑΛ του KFP μεταγλωττίζει τον κώδικα σε Argo δήλωση. Μια Γλώσσα Ειδικού Σκοπού (ΓΕΣ) [17] βασισμένη σε Python παρέχεται στο χρήστη, φέρνοντας έτσι το Argo κοντά στα στάνταρ ενός επιστήμονα δεδομένων ελαχιστοποιώντας την πολυπλοκότητα που διαγράφουν οι YAML δηλώσεις.

  Αφού πρόκειται για νέο έργο, είναι αναμενόμενο πως θα είναι ατελές. Ωστόσο, χάρις στις δεκάδες των συνεργατών, το πεδίο που καλύπτει του αυξάνεται ραγδαία. Κατά τη συγγραφή αυτής της εργασίας, η ΓΕΛ δεν υποστηρίζει όλες τις δυνατότητες του Argo. Όσον αφορά τη μεταφορά δεδομένων μεταξύ των διεργασιών, η ΓΕΛ υποστηρίζει μόνο μία συγκεκριμένη μέθοδο του Argo, η οποία είναι περιοριστική στο μέγεθος και τον τύπο των δεδομένων που μπορούν να μεταφερθούν.

  Συμπληρωματικά, η δυναμική δημιουργία πόρων (όπως για παράδειγμα σχετικών με το persistent storage [41] και τη λήψη στιγμιοτύπων των δεδομένων) κατά τη διάρκεια της διοχέτευσης δεν υποστηρίζεται, ενώ η χρήση ήδη υπαρχόντων απαιτεί πολλές γραμμές κώδικα.

[40] Με τον ελληνικό όρο "μηχανικός δεδομένων" αναφερόμαστε στον αγγλικό όρο "data engineer"

[41] Ελλείψει κατάλληλου όρου, γίνεται χρήση του αγγλικού "persistent storage" αντί του εν προκειμένω αδόκιμου ελληνικού όρου "επίμονη αποθήκευση"

0.1.4 Προτεινόμενη Λύση

Ο σκοπός και το επίτευγμα αυτής της διπλωματικής εργασίας είναι να καταστήσει δυνατή τη δυναμική δημιουργία persistent storage και στιγμιοτύπων στους επιστήμονες που χρησιμοποιούν το Kubeflow Pipelines, αλλά να παρέχει και έναν εύκολο τρόπο για τη χρήση αυτών των χαρακτηριστικών. Στην πραγματικότητα, αναδεικνύει τους τόμους αποθήκευσης [42] σε "πολίτες πρώτης κατηγορίας" του ΚΑΛ.

Για να το κάνουμε αυτό, πρώτα επεκτείνουμε τη ΓΕΣ με εύκολες στη χρήση οντότητες και στη συνέχεια επεκτείνουμε τον μεταγλωττιστή ώστε να παράγει την αντίστοιχη Argo δήλωση.

Πρώτα, εισάγουμε μια νέα αντιστοίχηση στη λειτουργία διαχείρισης πόρων του Argo. Αυτό είναι ένας βασικός τύπος και μπορεί να χρησιμοποιηθεί για τη διαχείριση οποιουδήποτε πόρου του Kubernetes. Οι χρήστες μπορούν να πραγματοποιήσουν οποιαδήποτε πράξη σε ένα πόρο και, προαιρετικά, να παρέχουν συνθήκες που θα καθορίσουν την επιτυχία ή την αποτυχία του βήματος που αναλαμβάνει αυτήν την πράξη. Αυτό απέχει πόρρω από το να είναι φιλικό προς το χρήστη, ενώ είναι θεμελιώδες για τη δημιουργία πόρων.

Η ανύψωση μιας οντότητας σε πολίτη πρώτης κατηγορίας προϋποθέτει να είναι βολικό για τον μέσο χρήστη. Για το λόγο αυτό, παρέχουμε δύο υποτύπους που αντλούνται από προαναφερθέν. Ο πρώτος αφορά τη διαχείριση persistent storage, ενώ ο δεύτερος τα στιγμιότυπά του.

Αυτές οι κλάσεις αποσκοπούν στο να κάνουν τη συνήθη περίπτωση γρήγορη. Εκθέτουν μια Διεπαφή Προγραμματισμού Εφαρμογών (ΔΠΕ) [18] [43] που απλοποιεί σημαντικά τον ορισμό των εκάστοτε πόρων, συγκριτικά με τη ΔΠΕ του Kubernetes.

Όλα τα παραπάνω λύνουν μόνο τη δυναμική δημιουργία των πόρων. Η χρήση του persistent storage ακόμη χρειάζεται πολλές γραμμές κώδικα. Για να υπερκεράσουμε αυτό το ζήτημα, επεκτείνουμε τη ΔΠΕ των συστατικών της διοχέτευσης με έναν εύκολο, κομψό και πλήρη τρόπο να προσδένουμε οποιονδήποτε τύπο τόμου[44].

Είναι σημαντικό να τονίσουμε πως ενώ καταφέραμε μια μεγάλη αλλαγή στο πρότζεκτ,

[42] Με τον ελληνικό όρο "τόμος αποθήκευσης" αναφερόμαστε στον αγγλικό όρο "storage volume"

[43] Με τον ελληνικό όρο "Διεπαφή Προγραμματισμού Εφαρμογών" ή "ΔΠΕ" αναφερόμαστε στον αντίστοιχο αγγλικό "Application Programming Interface" ή "API"

[44] Με τον ελληνικό όρο "τόμος" αναφερόμαστε στον αντίστοιχο αγγλικό "volume"

δεν χαλάσαμε την προϋπάρχουσα ΔΠΕ, διατηρώντας έτσι προς τα πίσω συμβατότητα [45].

0.1.5 Συνεργασίες

Κατά τη διάρκεια της διπλωματικής εργασίας συνεργαστήκαμε διεξοδικά με τους προγραμματιστές δύο πρότζεκτ και διασυνδεθήκαμε με τις κοινότητές τους.

- **Kubeflow Organization**

  Αφού αυτή η εργασία επεκτείνει το KFP, το οποίο διαχειρίζεται ο οργανισμός Kubeflow [19], το να επικοινωνήσουμε το σχεδιασμό μας με τους τελικούς χρήστες και τους προγραμματιστές ήταν απαραίτητο. Αυτό είχε πολλά θετικά, όπως το ότι θα ανασκοπούνταν από ένα σύνολο κόσμου έμπειρο σε αυτόν τον τομέα. Ακόμη, ο σχεδιασμός θα έπρεπε να εγκριθεί από χρήστες οι οποίοι υποστηρίζουν, χρησιμοποιούν και δοκιμάζουν αυτή τη λειτουργία. Τέλος, είχαμε διαθέσιμη δομή δοκιμών [46] που μας επέτρεψε να προσθέσουμε επιπρόσθετες εξετάσεις [47] Η κοινότητα του Kubeflow, και ειδικά οι προγραμματιστές και οι χρήστες του Kubeflow Pipelines, υποδέχθηκαν εξαρχής με πολύ θετική διάθεση την προσπάθειά μας και μας παρείχαν βοήθεια και καθοδήγηση αδιάλειπτα.

  Στο τέλος, έγινα μέλος του οργανισμού Kubeflow.

- **Argoproj Organization**

  Συμπληρωματικά στο Kubeflow, κατά την υλοποίηση της εργασίας βρεθήκαμε σε ανάγκη λειτουργιών που δεν ήταν διαθέσιμες ή δεν λειτουργούσαν σωστά στο Argo, τη μηχανή ροών εργασίας πίσω από το Kubeflow Pipelines. Αυτό είχε σαν αποτέλεσμα να εργαστούμε πάνω σε μερικές μικρές διορθώσεις και επεκτάσεις που ενσωματώθηκαν στο επίσημο πρότζεκτ.

  Έτσι, ήρθα σε επαφή με τους ιδιοκτήτες του πρότζεκτ, ανταλλάσσοντας απόψεις και ιδέες. Αυτό με κατέστησε έμπιστο συνεργάτη του οργανισμού Argoproj.

[45] Με τον ελληνικό όρο “προς τα πίσω συμβατότητα” αναφερόμαστε στον αντίστοιχο αγγλικό “backwards compatibility”

[46] Με τον ελληνικό όρο “δομή δοκιμών” αναφερόμαστε στον αντίστοιχο αγγλικό “testing framework”

[47] Με τον ελληνικό όρο “εξέταση” αναφερόμαστε στον αντίστοιχο αγγλικό “tests”

0.2 Σχεδιασμός

0.2.1 Αρχές Σχεδιασμού και Στόχοι

Όπως ήδη επιχειρηματολογήσαμε στην Εισαγωγή, η μηχανική μάθηση βασίζεται στα δεδομένα. Αυτό σημαίνει πως τα βήματα μιας διοχέτευσης θα έπρεπε να μπορούν να ανταλλάσσουν πολλά GB δεδομένων πραγματοποιώντας στάνταρ πράξεις εισόδου-/εξόδου (Ε/Ε) σε προσαρτημένα `Persistent Volumes` (PVs), χωρίς να οφείλουν να κατεβάζουν δεδομένα από εξωτερικές αποθήκες αντικειμένων, ούτε να μεταφορτώνουν σε αυτές.

Για τη διαχείριση τόμων αποθήκευσης, χρησιμοποιούμε στάνταρ, ουδέτερες από τον πωλητή [48] αρχές του Kubernetes (αναφορικά `PersistentVolumeClaim` και `Volume Snapshot` αντικείμενα), αλλά χωρίς να επιβάλλεται στο χρήστη να διαχειρίζεται αυτά τα αντικείμενα του Kubernetes. Αντί αυτού, ο στόχος είναι ο χρήστης να δηλώνει τον τρόπο που τα βήματα χρησιμοποιούν τους τόμους ως πόρους της διοχέτευσης για ανταλλαγή δεδομένων, με μια ενστικτώδη σύνταξη στη ΓΕΣ, όμοια με τους υπόλοιπους πόρους της διοχέτευσης.

Επεκτείνοντας τη ΓΕΣ ώστε να υποστηρίζει τη χρήση PVs για ανταλλαγή δεδομένων θα κάνει την τοπική χρήση του Kubeflow Pipelines πολύ πιο εύκολη, και θα ενεργοποιήσει τη χρήση προηγμένης λειτουργικότητας αποθήκευσης [49] που προσφέρεται από τις σύγχρονες μεθόδους αποθήκευσης, δηλαδή τα στιγμιότυπα, ως τρόπο να υπάρχει επίγνωση σχετικά με τον τρόπο που μια διοχέτευση πολλών GB εκτελείται σε κάθε βήμα.

Υποστηρίζοντας τόμους εγγενώς στη ΓΕΣ τα δεδομένα σε ένα PV είναι ένα ακόμα κομμάτι πληροφορίας που περνάει από βήμα σε βήμα, και μπορεί να χρησιμοποιηθεί σαν είσοδος από τον μεταγλωττιστή του Pipelines ώστε να εξάγει πληροφορίες εξάρτησης μεταξύ διεργασιών.

Συνολικά, ο στόχος είναι να αναδείξουμε τα `PersistentVolumes` και `VolumeSnapshots` σε κορυφαίου επιπέδου αντικείμενα στη ΓΕΣ του KFP, επιτρέποντας πλήρη προγραμ-

[48] Με τον ελληνικό όρο "ουδέτερο από τον πωλητή" αναφερόμαστε στον αγγλικό όρο "vendor-neutral"

[49] Με τον ελληνικό όρο "προηγμένη λειτουργικότητα αποθήκευσης" αναφερόμαστε στον αντίστοιχο αγγλικό "advances storage functionality"

ματιστική διαχείριση και κρύβοντας τις λεπτομέρειες χαμηλού επιπέδου για τη διαχείριση των αντικειμένων άμεσα μέσω της ΔΠΕ του Kubernetes.

0.2.2 Αρχικό Κείμενο Τεχνικών Προδιαγραφών

Για να πραγματοποιήσουμε όλα τα παραπάνω, δημιουργούμε δύο νέες οντότητες: τα `PipelineVolumes` και τα `PipelineVolumeSnapshots`. Ακόμη, απλοποιούμε τη χρήση τους από βήματα της ροής εργασίας εκθέτοντας ένα μηχανισμό και επεκτείνοντας τον ορισμό του `ContainerOp`, ώστε να τα προσαρτά με ευκολία.

Ακολούθως περιγράφουμε αυτές τις αφαιρέσεις και τη χρήση τους.

- **PipelineVolumes**

  Το `PipelineVolume` είναι μια οντότητα της ΓΕΣ που είτε αναφέρεται σε ένα υπάρχον `PersistentVolumeClaim` είτε είναι υπεύθυνη για τη δημιουργίας ενός νέου. Εκθέτει έναν εύκολο τρόπο ώστε να τεθεί ένα υποσύνολο των χαρακτηριστικών ενός PVC, κάνοντας έτσι τη συνήθη περίπτωση γρήγορη, αλλά επιτρέποντας επίσης στο χρήστη να παρέχει ένα πλήρως προσαρμοσμένο στις ανάγκες αντικείμενο μέσω των μοντέλων του πελάτη[50] του Kubernetes, γεγονός που με τη σειρά του επιτρέπει το πλήρες φάσμα προδιαγραφών.

  Εφόσον αυτοί οι τόμοι περιέχουν πληροφορίες παραγόμενες από κάποιες διεργασίες της ροής εργασίας, η χρήση ενός περιστατικού[51] μιας τέτοιας οντότητας από ένα βήμα υπονοεί πως η εκτέλεσή του πρέπει να ξεκινήσει μετά από εκείνες τις διεργασίες. Το καταφέρνουμε αυτό δένοντας κάθε περιστατικό με οποιαδήποτε απαιτούμενη εξάρτηση, ενόσω παρέχεται ένας εκφραστικός, συναρτησιακός τρόπος επέκτασης αυτών των εξαρτήσεων.

- **PipelineVolumeSnapshots**

  Ομοίως, το `PipelineVolumeSnapshot` είναι μια οντότητα της ΓΕΣ που είτε αναφέρεται σε ήδη υπάρχον `VolumeSnapshot` είτε είναι αίτηση για τη δημιουργία ενός τέτοιου. Η λογική είναι ίδια με προηγουμένως: ένα συχνά χρησιμοποιούμενο υποσύνολο των χαρακτηριστικών του πόρου εκτίθεται μέσω μιας

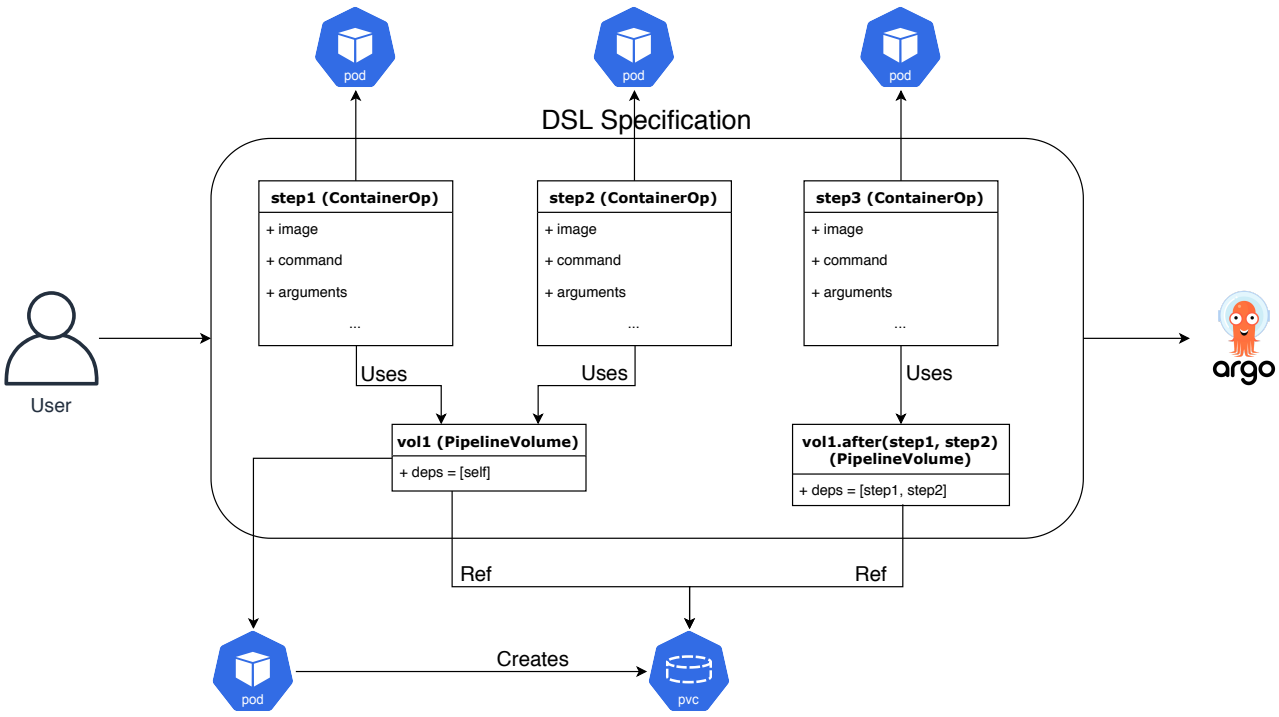
[50] Με τον ελληνικό όρο "πελάτης" αναφερόμαστε στον αγγλικό όρο "client"
[51] Με τον ελληνικό όρο "περιστατικού" αναφερόμαστε στον αγγλικό όρο "instance"

απλής ΔΠΕ, αλλά οι χρήστες έχουν τη δυνατότητα να περάσουν ένα προσαρμοσμένο αντικείμενο, επίσης.

Σε αντίθεση με τα `PersistentVolumeClaims`, τα `VolumeSnapshots` είναι αμετάβλητα. Τα περιεχόμενα δεδομένα δεν μπορούν να μεταβληθούν από κανένα άλλο βήμα. Λαμβάνοντας αυτό υπόψιν, δεν υπάρχει ανάγκη για πολλαπλά περιστατικά που να αναφέρονται στον ίδιο πόρο και να έχουν διαφορετικές εξαρτήσεις. Ένα `VolumeSnapshot` που αναφέρεται από ένα περιστατικό στον προσδιορισμό της διοχέτευσης μέσω της ΓΕΣ είναι δεμένο με συγκεκριμένες αμετάβλητες εξαρτήσεις.

- **Usage**

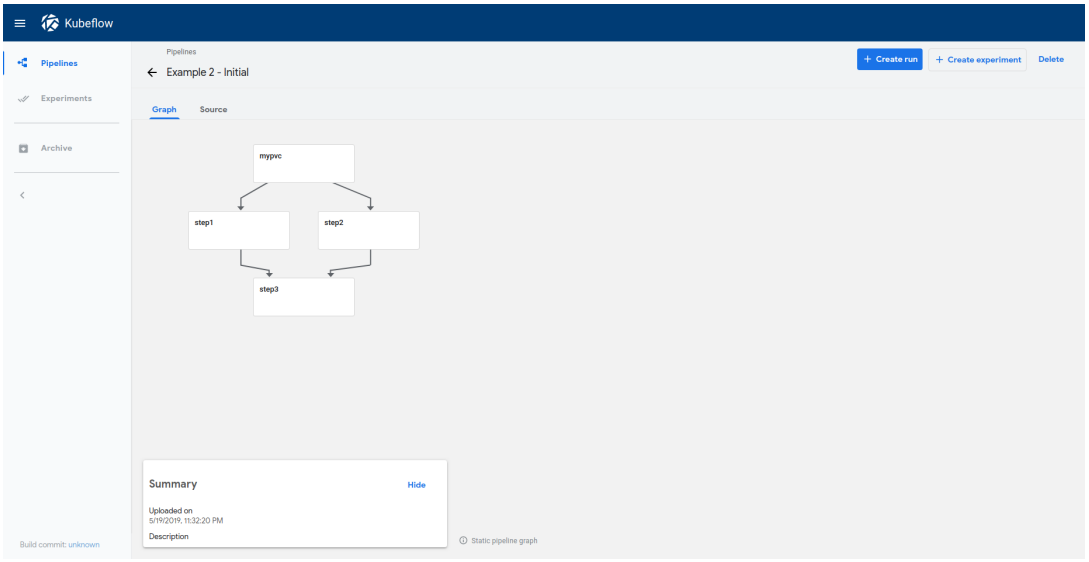
  Για τη χρήση αυτού του σχεδιασμού παραπέμπουμε στο αγγλικό κείμενο και συγκεκριμένα στην υποπαράγραφο 3.2.3 Usage.



Σχήμα 1: *Αρχικό Κείμενο Τεχνικών Προδιαγραφών - Οπτικοποίηση Παραδείγματος*

0.2.3 Ανασκοπήσεις

Το αρχικό κείμενο τεχνικών προδιαγραφών αντανακλούσε τις ανάγκες των χρηστών και τις ικανοποιούσε σε σημαντικό βαθμό. Παρόλα αυτά, έχοντας διαθέσιμο τον κώ-



Σχήμα 2: *Αρχικό Κείμενο Τεχνικών Προδιαγραφών - Γράφος Παραδείγματος*

δικα υλοποίησης ώστε να εξετασθεί σε πολλών ειδών σενάρια καθίσταται πιο ξεκάθαρος ο ίδιος ο σχεδιασμός. Έτσι, μόλις υποβάλαμε στο GitHub[28] το αίτημα ενσωμάτωσης στο επίσημο πρότζεκτ, οι συντηρητές του εξέτασαν την υλοποίηση και επέστρεψαν με ιδέες.

Η πρότασή τους αφορούσε την ύπαρξη κάποιας αφαίρεσης για τη διαχείριση οποιωνδήποτε πόρων του Kubernetes. Τότε οι οντότητες υπεύθυνες για τη δημιουργία των `PersistentVolumeClaims` ή των `VolumeSnapshots` θα ήταν υποπεριπτώσεις.

Αυτό είχε ως συνέπεια ειδικός και περιορισμένος σκοπός να εξυπηρετείται από κάθε οντότητα. Για παράδειγμα, στον αρχικό σχεδιασμό, ένα `PipelineVolume` μπορούσε να σημαίνει είτε την αναφορά σε υπάρχον PVC ή τη δημιουργία νέου. Θα δείξουμε, όμως, πως στο τελικό σχέδιο χρησιμοποιείται για την αναφορά και μόνο.

Μια τέτοια βελτίωση απλοποιεί τον κώδικα, διευκολύνει τη συντήρησή του και ανοίγει το έδαφος για περαιτέρω επεκτάσεις. Οπότε, αφού αναλύσαμε αυτές τις ιδέες, δημοσιοποιήσαμε έναν πλήρως αναθεωρημένο σχεδιασμό που περιγράφεται στη συνέχεια.

Μεταξύ άλλων, αλλάζουμε το όρισμα `volumes` και τα χαρακτηριστικά `volume` και `volumes` σε `pvolumes` και `pvolume`, αντίστοιχα, προκειμένου να μην διατηρήσουμε την προς τα πίσω συμβατότητα. Τελικώς, μεταβάλλουμε τον τρόπο με τον οποίο εκφράζονται κάποιες εξαρτήσεις, χωρίς πλέον να αφαιρούμε τις “περιττές” εξαρτήσεις που προέκυπταν από τη χρήση του δημιουργηθέντος PVC.

0.2.4 Τελικό Κείμενο Τεχνικών Προδιαγραφών

Στο τελικό κείμενο τεχνικών προδιαγραφών επεκτείνουμε τη ΓΕΣ με περισσότερες οντότητες, ενώ πειράζουμε ή ακόμη αφαιρούμε αυτές που περιγράφηκαν στον αρχικό σχεδιασμό. Ακολουθεί μια αναλυτική σκιαγράφηση αυτών.

- **PipelineVolume: Απλοιποιώντας τη Χρήση PVC**

  Ένα `PipelineVolume` κληρονομεί από το `Volume`[30], τη θεμελιώδη οντότητα του Kubernetes. Στον πυρήνα του, ένας τόμος είναι απλά ένας κατάλογος[52], πιθανά με δεδομένα στο εσωτερικό του, ο οποίος είναι προσιτός στους περιέκτες ενός `Pod`. Το πώς είναι αυτός ο κατάλογος, το μέσο που το υποστηρίζει και τα περιεχόμενά του καθορίζονται από τον εκάστοτε τύπο τόμου που χρησιμοποιείται.

  Υπάρχουν διάφοροι τύποι τόμων, όμως θα επικεντρωθούμε στην κυρίαρχη χρήση μας η οποία είναι οι τόμοι που αναφέρονται σε PVCs.

  Η διαφορά ενός `PipelineVolume` από ένα `Volume` είναι το γεγονός ότι το πρώτο κρατάει εξαρτήσεις. Σε ό,τι αφορά αυτό, υιοθετούμε τις ίδιες αρχές που περιγράφηκαν στο αρχικό κείμενο τεχνικών προδιαγραφών.

  Ωστόσο, αυτή η οντότητα δεν είναι υπεύθυνη για τη διαχείριση `Persistent VolumeClaim` πόρων.

- **PipelineVolumeSnapshot: Το Τέλος μιας Εποχής**

  Αυτή η οντότητα αφαιρείται πλήρως.

- **ResourceOp: Ενεργοποιώντας μια Νέα Λειτουργία του Argo**

  Αυτή είναι η βασική οντότητα που χρησιμοποιείται για τη διαχείριση πόρων του Kubernetes. Είναι μια κλάση της ΓΕΣ που μοντελοποιεί αυτή τη λειτουργία του Argo.

  Ο χρήστης μπορεί να ορίσει κάποια δράση που επιθυμεί να πραγματοποιηθεί πάνω σε έναν πόρο του Kubernetes (η προκαθορισμένη είναι η δημιουργία) και ένα αντικείμενο με τον προσδιορισμό του πόρου. Μπορεί επίσης να προσδιορίσει συνθήκες που θα σημάνουν την επιτυχία ή την αποτυχία του βήματος.

[52] Με τον ελληνικό όρο “κατάλογος” αναφερόμαστε στον αγγλικό όρο “directory”

Μια τελευταία και πολύ σημαντική λειτουργία είναι η δυνατότητα να εξάγουμε ως παράμετρο οποιοδήποτε χαρακτηριστικό του πόρου, που μετά μπορεί να χρησιμοποιηθεί από οποιοδήποτε βήμα της διοχέτευσης. Επιλέγουμε την εξαγωγή του ονόματος αλλά και ολόκληρου του προσδιορισμού του πόρου σε κάθε περίπτωση, όμως ο χρήστης μπορεί να ορίσει περισσότερες εξόδους.

Σχεδιάζοντας αυτήν την οντότητα, ανακαλύψαμε πολλά κοινά σημεία με τα `ContainerOp`s, όπως ήταν αναμενόμενο. Έτσι, δημιουργήσαμε την κλάση `BaseOp`, που είναι γονέας των δύο προαναφερθέντων και εμπεριέχει όλα τα κοινά χαρακτηριστικά και τις μεθόδους που έχουν νόημα και για τις δύο κλάσεις.

- **VolumeOp - VolumeSnapshotOp: Απλοποιώντας τη Δημιουργία των Persistent VolumeClaims και VolumeSnapshots**

  Οι `VolumeOp` και `VolumeSnapshotOp` είναι υποκλάσεις[53] της `ResourceOp`. Η πρώτη στοχεύει στο να κάνει τη δημιουργία των `PersistentVolumeClaim`s πιο εύκολη, ενώ η δεύτερη τη δημιουργία των `VolumeSnapshot`s.

  Αυτές οι κλάσεις είναι υπεύθυνες μόνο για τη διαχείριση των πόρων. Και οι δύο εκθέτουν μια ΔΠΕ ώστε να κάνουν τη συνήθη περίπτωση γρήγορη, όπως έκαναν στην αρχική σχεδίαση τα `PipelineVolume` και `PipelineVolumeSnapshot`.

  Ειδικότερα για τα `VolumeOp`s, εκθέτουμε ένα χαρακτηριστικό "`volume`", το οποίο είναι ένα `PipelineVolume` που αναφέρεται στο δημιουργηθέν PVC, ώστε να το προσαρτούμε με ευκολία σε ένα `ContainerOp`.

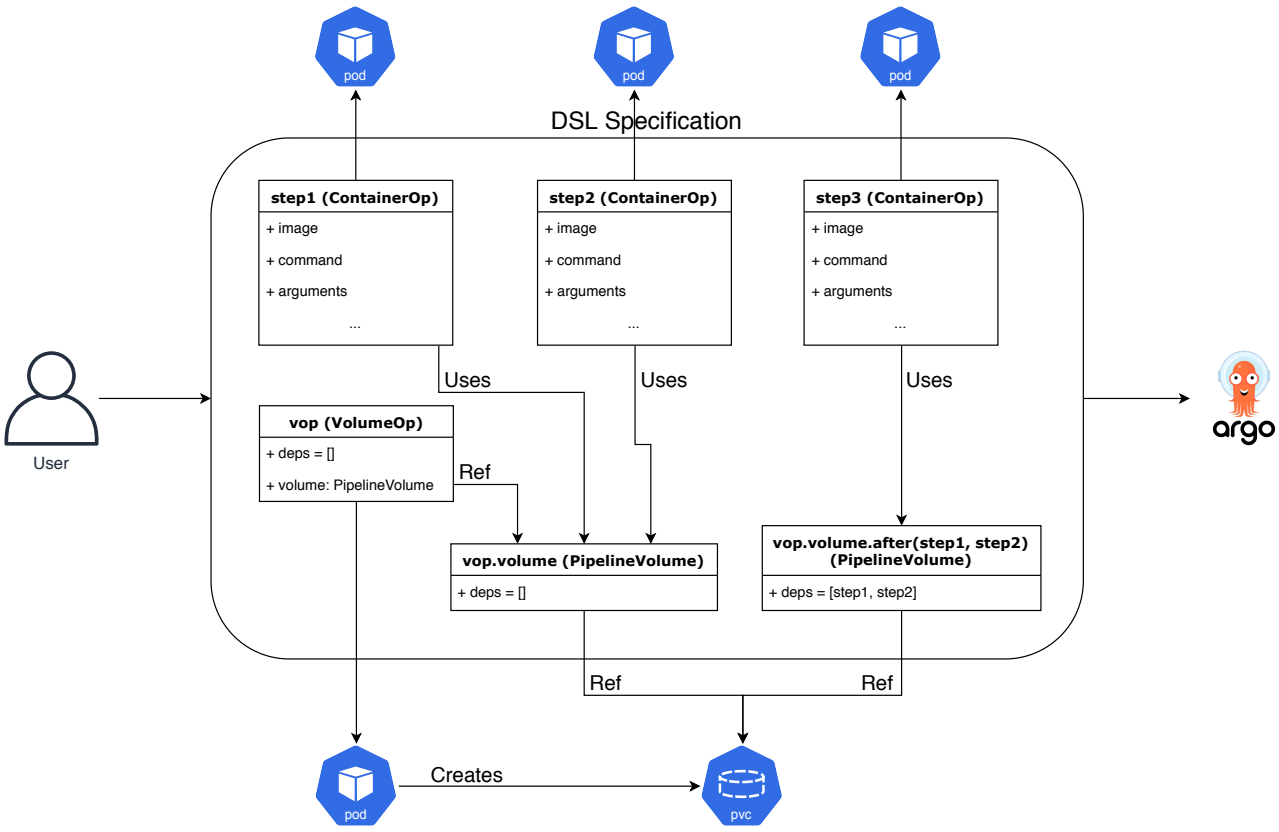
- **Usage**

  Για τη χρήση αυτού του σχεδιασμού παραπέμπουμε στο αγγλικό κείμενο και συγκεκριμένα στην υποπαράγραφο 3.4.5 Usage.

0.3 Συμπεράσματα

Συμπερασματικά, ο πρωταρχικός στόχος αυτής της διπλωματικής εργασίας ήταν η υλοποίηση μιας σχεδίασης που καλύπτει τη λογική μας όπως την έχουμε περιγράψει.

[53] Με τον ελληνικό όρο "υποκλάση" αναφερόμαστε στον αγγλικό όρο "subclass"

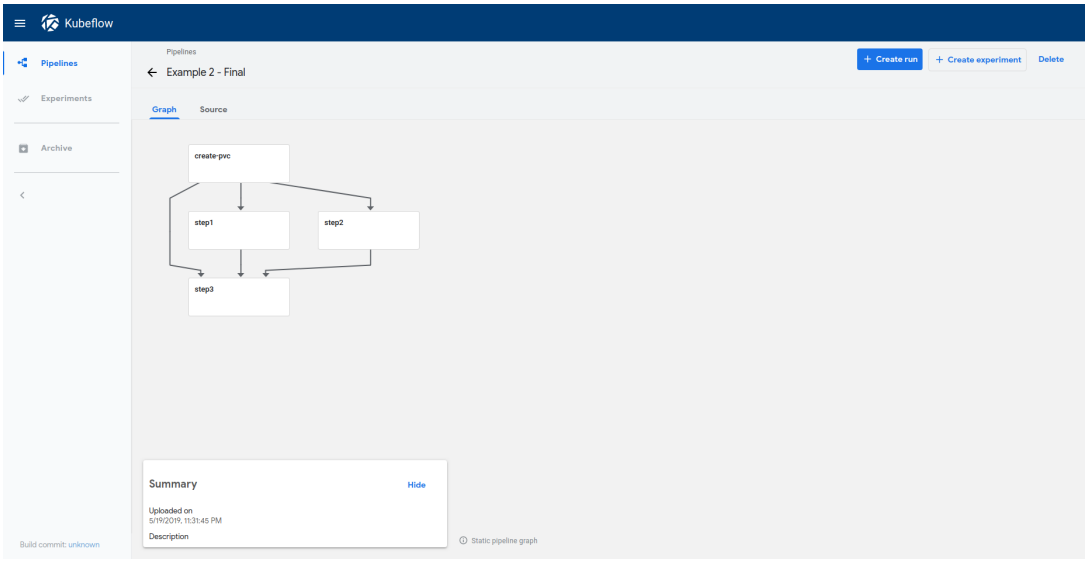


Σχήμα 3: *Τελικό Κείμενο Τεχνικών Προδιαγραφών - Οπτικοποίηση Παραδείγματος*

Εκτός αυτού, υπήρχε και ένας ακόμη βαθύτερος στόχος: να συνεργαστούμε αποτελεσματικά με προγραμματιστές και χρήστες από όλον τον κόσμο και να συνεισφέρουμε σε ένα πρότζεκτ ανοιχτού κώδικα αυτού του βεληνεκούς. Μπορούμε, πλέον, να καταλήξουμε στο γεγονός ότι φέραμε εις πέρας και τους δύο στόχους.

Πιο συγκεκριμένα, αναφορικά με τον βασικό μας στόχο, την επέκταση της ΓΕΣ, καταφέραμε μια μεγάλη αλλαγή: το Kubeflow Pipelines μπορεί πλέον να εκτελεστεί σε τοπικές συστοιχίες με την υποστήριξη τόμων, επιτρέποντας επίσης την αναπαραγωγιμότητα των ροών εργασίας. Η απόδειξη ορθότητας του σχεδιασμού δεν μπορεί παρά να είναι το πλήθος των οργανισμών που χρησιμοποιούν την υλοποίησή μας, μεταξύ των οποίων η Cisco, η IBM και η Seldon, αλλά και ιδιωτικοί χρήστες. Καθώς επίσης, μέρα με τη μέρα το χρησιμοποιεί όλο και περισσότερος κόσμος. Αυτό ήταν αναπόφευκτο, αφού συμφωνήσαμε σε ένα κείμενο τεχνικών προδιαγραφών που υποδέχτηκε θερμά ολόκληρη η κοινότητα του Kubeflow.

Σε ό,τι αφορά το δεύτερο στόχο, τη συνεργασία, επρόκειτο για μία πρωτόγνωρη και γόνιμα αγχωτική εμπειρία, που όμως ήταν έντονα επιμορφωτική και συνολικά υπέ-



Σχήμα 4: *Τελικό Κείμενο Τεχνικών Προδιαγραφών - Γράφος Παραδείγματος*

ροχη.

0.3.1 Μελλοντικές Επεκτάσεις

Παρόλο που έχουμε υλοποιήσει μερικές βελτιώσεις στη Διεπαφή Χρήστη (ΔΧ) [54], βελτιώνοντας την Εμπειρία Χρήστη (ΕΧ) [55], υπάρχουν περιθώρια για περαιτέρω ανάπτυξη.

Συμπληρωματικά, μπορούμε να δημιουργήσουμε περισσότερες κλάσεις, που θα είναι υποκλάσεις της `ResourceOp`, για τη διευκόλυνση της δημιουργίας άλλων πόρων του Kubernetes που είναι χρήσιμοι στους επιστήμονες δεδομένων. Παραδείγματος χάριν, ένα `TFJob` χρησιμοποιείται ευρέως για την εκτέλεση λειτουργιών εκμάθησης στο Kubernetes. Θα μπορούσαμε να υλοποιήσουμε την κλάση `TFJobOp` ακολουθώντας την ίδια αρχή, που είναι το να είναι η συνήθης περίπτωση γρήγορη, αλλά και να καθίσταται δυνατός ο πλήρης προσδιορισμός του πόρου.

Επιπρόσθετα, θα μπορούσαμε να αναζητήσουμε κάποια ενοποίηση με την υπηρεσία Metadata του Kubeflow [33]. Αυτή αναλαμβάνει, μεταξύ άλλων, να παρακολουθεί εκτελέσεις ροών εργασίας διατηρώντας εγγραφές για το ποιες ήταν οι είσοδοι και έξο-

[54] Με τον ελληνικό όρο “Διεπαφή Χρήστη” ή “ΔΧ” αναφερόμαστε στον αντίστοιχο αγγλικό “User Interface” ή “UI”

[55] Με τον ελληνικό όρο “Εμπειρία Χρήστη” ή “ΕΧ” αναφερόμαστε στον αντίστοιχο αγγλικό “User Experience” ή “UX”

δοί τους. Αυτές οι είσοδοι και έξοδοι θα μπορούσαν να είναι αναφορές σε στιγμιότυπα τόμων, τα οποία μετά να είναι διαθέσιμα για εξερεύνηση με ένα κλικ.

Τέλος, η λογική πίσω από το σχεδιασμό μας ανοίγει το δρόμο για περαιτέρω ανάπτυξη με σκοπό την κάλυψη επιπλέον λειτουργιών του Argo. Για παράδειγμα, το Argo παρέχει τρόπο για την εκτέλεση μιας δέσμης ενεργειών [56] και την άμεση συλλογή των αποτελεσμάτων, χωρίς να χρειάζεται η δημιουργίας ενός στιγμιοτύπου[57] που να περιέχει τη δέσμη ενεργειών ακολουθούμενη από την αίτηση ενός περιέκτη που θα το εκτελεί. Αυτό φαντάζει πολύ χρήσιμο για επιστήμονες στο πεδίο της μηχανικής μάθησης. Φτάνει μόνο να επιβεβαιώσουμε αυτήν την εικασία και να σχεδιάσουμε μια τέτοια υποστήριξη με τη βοήθειά τους.

[56] Με τον ελληνικό όρο “δέσμη ενεργειών” αναφερόμαστε στον αντίστοιχο αγγλικό “script”

[57] Με τον ελληνικό όρο “στιγμιότυπο” αναφερόμαστε στον αντίστοιχο αγγλικό “image”, και πιο συγκεκριμένα “container image”

1 Introduction

To begin with, we outline the scope of our work. First, we provide a quick overview of the problem we are trying to solve and argue about its importance. Next, we move on to describe some of the existing solutions in brief, examining parameters that might be making some of them rather cumbersome to work with, and some others to potentially thrive. After this, we adumbrate the flow of our reasoning on confronting the problem, along with our respective design attempts. Finally, we present how various concepts, as well as design and implementation issues are structured within this thesis.

1.1 Problem Statement

Machine Learning (ML) [1] tasks may consume or produce large quantities of data. Actually, depending on the problem, the more the data the better the results. For instance, to address some of the "hard" problems, like those tackled by deep learning, millions or even tens-of-millions of examples may need to be analyzed. The management of such data (i.e. storing, versioning, etc) is challenging for data scientists and the Platform-as-a-Service providers. Nevertheless, it is a difficulty to subdue howsoever.

A popular machine learning toolkit dedicated to making deployment of ML workflows on Kubernetes simple, portable and scalable is Kubeflow[2]. Kubeflow Pipelines (KFP) [3] 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.

In Kubeflow Pipelines there is no way to perform intensive data management. To be precise, the data management available to the user is quite restrictive with respect to both their size and their type, while no data versioning is supported.

We will make a deeper analysis of the current state later in this chapter.

1.2 Motivation

1.2.1 What is Machine Learning?

In [4] we may find a precise definition of Machine Learning:

> Machine learning is an application of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from experience without being explicitly programmed. Machine learning focuses on the development of computer programs that can access data and use it to learn for themselves.
>
> The process of learning begins with observations or data, such as examples, direct experience, or instruction, in order to look for patterns in data and make better decisions in the future based on the examples that we provide. The primary aim is to allow the computers learn automatically without human intervention or assistance and adjust actions accordingly.

Machine learning algorithms produce models which are tightly coupled to a cost function. This function describes the accuracy of the model, its effectiveness, therefore such algorithms seek a model that optimally minimizes the cost function.

1.2.2 Why is it Required?

Unfortunately, the coding of some tasks can get too complex for humans. Their analysis and their solution may render impractical or even impossible. In such cases, machine learning algorithms come to save the day. Those algorithms can be provided with a large amount of data, explore them, and search for a model that achieves anything the scientist programs them to.

Machine learning has far too many applications and exists in our everyday life more than we might believe it does. Email categorization, Facebook's News Feed, Voice-to-Text as well as all kinds of online advertisements are just some of them.

Self-driving cars or self-flying helicopters are some much more impressive achievements of machine learning. However, these are also examples which have failed in critical situations, such as [5], and point out the necessity to further improve the results of the specific AI field.

1.2.3 What are the Requirements?

Machine learning models are nothing more than mathematical functions that take features as inputs, produce predictions as outputs, and learn how to best match predictions to patterns observed from the training data. Therefore, a crucial step during the development of a ML application is to figure out the required data, as well as their quantity.

As already mentioned, models are not 100% correct, but are rather "best guesses" given the amount of data the model has seen. The set of known inputs and outputs that the data analyst uses to "train" a model is called **"training set"**. The training set may be used as input in numerous algorithms, or even in the same algorithm with different options, to generate multiple models.

The next step is to check the performance of the aforementioned "trained" models. This is achieved by feeding each model with a new dataset, called the **"validation set"**. The results of the validation lead to a winner election: the selection of the most accurate model.

Finally, it is time to put the best model to actual use. We do not need an ML algorithm to guess that there is a dataset which we want to explore. It is called the **"test set"**. All the previous workflow is performed for this exact point: the **"serving"**. During this phase, the actual data analysis is accomplished.

Taking all of the above into consideration, one corollary stands out: machine learning workflows are **data intensive** and, therefore, require **advanced data management**.

1.2.4 Why is Deploying Machine Learning Hard?

It is a fact that data get bigger and bigger and that infrastructure requirements for the computations are getting greater. As a result, many users turn to solutions like Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), Software-as-a-Service (SaaS) or any other type of Cloud[7] services. It appears like we live the dawn of the Everything-as-a-Service (EaaS/XaaS/*aaS) era.

These services run over distributed systems. Unavoidably, there is quite some complexity involved. Orchestrating services and tasks, handling storage and tackling the problem of statefulness in general can be really challenging.

Data synchronization (consensus), **security**, **inter-services communication**, **load-balancing**, **high availability** and **fault-tolerance** are just some of the major problems that ail modern distributed systems and applications.

Solutions for such problems first arise from within the software industry. Four years ago, Google designed and implemented an open-source container orchestration system called **Kubernetes**[8] (also referred as "K8s", for short). The project is now maintained by Cloud Native Computing Foundation (CNCF) [9], and aims to make the deployment of containerized microservices[10] easier. Nevertheless, the intricacies of distributed systems are yet to be fully confronted.

As a result, a number of organizations have been developing platforms or extending their projects appropriately in order to perform machine learning on Kubernetes. In the majority of cases, this means to extend Kubernetes itself [11]. We will elaborate on what Kubernetes is and how it actually works later in this document.

In this thesis, we focus on the **state** of the machine learning workflows run on Kubernetes, as well as **managing** it and **versioning** it.

1.3 Existing Solutions

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 oper-

ations require a lot of irrelevant studying by the user.

In this section, we briefly present the most notable existing solutions.

1.3.1 Pachyderm

Pachyderm[12] is designed to enable sustainable data science workflows via a language-agnostic system for **data versioning** with **data pipelining**. It lets the user deploy and manage multi-stage data pipelines[1] while maintaining complete reproducibility and provenance.

Regarding a part of our goal, that is taking snapshots of the data, Pachyderm uses a Git[13]-like tool as a version control system for this purpose. The user may look back to see what the training data looked like when a particular model was created or how their results have changed over time.

Nevertheless, Pachyderm has some drawbacks:

- **Latency**: Deploying Pachyderm on a cluster requires a (Simple Storage Service) S3 [14] compatible backing object store. This makes the whole data management depend on network bandwidth and, as a result, it can be really slow.
- **Syntax**: Defining a step of the pipeline requires writing a lot of boilerplate code to accomplish only minor functionality.
- **Multi-step pipelines**: Every single task of the workflow requires its own file and its own submission separately.
- **Effort**[15]: Just as a data scientist needs to be able to work with Git, they also need to know how to work with this "Git for data". While this kind of tools are common and accepted as productivity enhancers within the data engineering world, it is often less accepted by mainstream machine learning scientists, unfortunately. This restricts the adoption rate among those without strong engineering skills.

[1] Pachyderm calls a *pipeline* what KFP call a task. We will always be using the KFP definition of the word, i.e. a multi-task workflow

1.3.2 Argo

Argo[16] is a workflow engine. It is an extension to the Kubernetes cluster[2] that makes the execution of workflows possible. The user submits a YAML definition of a workflow and then Argo is responsible to initiate tasks in proper order and wait for them to complete. It also provides a Command Line Interface (CLI) tool as well as a basic User Interface (UI) for the virtual representation of the workflows.

Kubeflow Pipelines uses Argo as their workflow engine. The Software Development Kit (SDK) compiles the user's source code into an Argo manifest.

Argo has wide functionality, which we leverage in this thesis. However, we ought to note a couple of drawbacks:

- **Access to the cluster**: Data scientists must have access to the cluster in order to use Argo. This is not the desired case. It would be best if there was a level of abstraction between data analysts and the infrastructure they may use.
- **Learn Kubernetes**: Additionally, users have to construct the Argo YAML file and submit it. Admittedly, such manifests are quite cumbersome and require effort to learn and write. A data scientist should also be abstracted from most of the Kubernetes specifics and focus on the AI part of the problem.

1.3.3 The State of Kubeflow Pipelines

Kubeflow Pipelines project is in a very early state, yet it has a resonance for lots of data analysts. That is quite a motive for the members of the Kubeflow organization and the contributors of the project to continue extending and developing new features.

As already mentioned, the KFP SDK compiles the code into an Argo manifest. A Domain Specific Language (DSL) [17] based on Python is provided to the user, thus bringing Argo closer to the data scientist's standards by minimizing the complexity looming over the YAML definitions.

Since it is a fresh project, it is also incomplete. Nonetheless, thanks to the tens of contributors to the project, its coverage is rapidly increasing. During the writing of

[2] Such Kubernetes cluster extensions are called *Operators*.

this thesis, the DSL does not support the full potential of Argo. As far as the data-passing between tasks is concerned, the DSL only supports a specific Argo method, which does not allow passing large quantities of data.

Moreover, the dynamic creation of resources (such as resources for persistent storage) during the pipeline is not supported, while the usage of existing ones requires boiler-plate code.

The above can be summarized as the following drawbacks:

- **Boilerplate code**: The data scientist has to write boilerplate code to use persistent storage for their pipeline task.
- **No dynamic persistent storage**: The data analyst cannot request persistent storage during the execution of the pipeline.
- **No dynamic snapshotting of data**: Since snapshotting requires the creation of specific Kubernetes resources, machine learning engineers cannot take immutable backups of their data dynamically during the execution of the pipeline. That makes it hard or even impossible for them to have insight into the data and operations taking place. As a consequence, there is no way to easily and effectively debug the pipeline by viewing the data (either consumed or produced) and then reproduce the whole job or a part of it.

1.4 Proposed Solution

The purpose and achievement of this thesis is to make the **dynamic** creation of persistent storage and snapshots available to the scientists using Kubeflow Pipelines, but also to provide an **easy** way to make use of these features. It, actually, leverages **storage volumes** into **first-class citizens** of the SDK.

To make this possible we first extend the DSL with easy-to-use entities, and then we extend the compiler to produce the corresponding Argo manifests.

Initially, we introduce a **new mapping** to Argo's resource manipulation feature. This is a base type and can be used to manage any type of Kubernetes resource. Users may perform any action to a resource (i.e. `get`, `create`, `apply`, `delete` or `replace`) and,

optionally, provide conditions to denote the success or failure of the workflow step undertaking the action. This is far from user-friendly, albeit fundamental to the creation of resources.

Raising something to a first-class citizen requires making it **convenient** for the mainstream user. Therefore, we also provide two subtypes which derive from the aforementioned one. The first is dedicated to persistent storage manipulation, while the second handles the snapshotting.

These classes aim to make the common case fast. They expose an Application Programming Interface (API) [18] that significantly simplifies the definition of the corresponding resources, compared to the Kubernetes API.

All of the above solve only the dynamic creation of resources. The usage of persistent storage itself still needs a lot of boilerplate. To overcome this issue, we extend the pipeline operators' API with an **easy, neat and thorough way to mount any type of volume**.

Before proceeding to the rest of the thesis, it is of great importance to point out that while achieving a **breaking change** to the project, we do not break the existing API thus retaining **backwards compatibility**.

1.5 Collaborations

During this thesis, we thoroughly collaborated with the developers of two projects and interfaced with their communities.

1.5.1 Kubeflow Organization

Since this thesis extends KFP, maintained by the Kubeflow Organization [19], communicating our thoughts and design with the end-users as well as other developers was indispensable. This offered a number of advantages:

- Our design and implementation was peer reviewed by industry experts.
- A healthy community and userbase would approve the design and embrace, use and test this feature.

- End-to-end testing framework with Argo integration, which helped us easily add tests.

The Kubeflow community, and especially the developers and users of Kubeflow Pipelines, welcomed the effort from the beginning and have constantly provided help and guidance.

In the end, I was also acknowledged as a **member** of the Kubeflow Organization.

1.5.2 Argoproj Organization

In addition to Kubeflow, during the implementation of this thesis we found ourselves in need of functionality that was not available or did not function properly in Argo, the workflow engine behind Kubeflow Pipelines. As a result, we worked on certain patches and bug fixes which were merged in the original upstream project.

All in all, I got in touch with the owners of the project and exchanged thoughts and ideas. This rendered me trusted **contributor** to the Argoproj Organization [20].

1.6 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 our work. Initially, we describe general principles of domain-specific languages. We move on to explain some fundamentals about containers, an implementation of OS-level virtualization. In the next section, we discuss about Kubernetes and some of its facets. Last but not least, we make a brief presentation of the two projects we mostly worked on: Argo and Kubeflow.

2.1 Domain-Specific Language

According to [21], a domain-specific language (DSL) is a computer language specialized to a particular application domain, in contrast to a general-purpose language (GPL), which is broadly applicable across domains.

The design and use of appropriate DSLs is a key part of domain engineering, by using a language suitable to the domain at hand – this may consist of using an existing DSL or GPL, or developing a new DSL. Language-oriented programming considers the creation of special-purpose languages for expressing problems as standard part of the problem-solving process. Creating a domain-specific language (with software to support it), rather than reusing an existing language, can be worthwhile if the language allows a particular type of problem or solution to be expressed more clearly than an existing language would allow and the type of problem in question reappears sufficiently often. Pragmatically, a DSL may be specialized to a particular problem domain, a

particular problem representation technique, a particular solution technique, or other aspects of a domain.

A domain-specific language is somewhere between a tiny programming language and a scripting language, and is often used in a way analogous to a programming library. The boundaries between these concepts are quite blurry, much like the boundary between scripting languages and general-purpose languages.

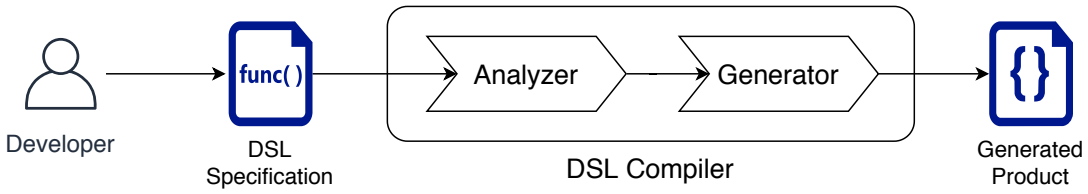


Figure 2.1: *Typical DSL Compiler*

2.2 Containers

A great overview of containers can be found in [22], where it is mentioned that containers offer a logical packaging mechanism in which applications can be abstracted from the environment in which they actually run. This decoupling allows container-based applications to be deployed easily and consistently, regardless of whether the target environment is a private data center, the public cloud, or even a developer's personal laptop. Containerization provides a clean separation of concerns, as developers focus on their application logic and dependencies, while IT operations teams can focus on deployment and management without bothering with application details such as specific software versions and configurations specific to the app.

For those coming from virtualized environments, containers are often compared with virtual machines (VMs). You might already be familiar with VMs: a guest operating system such as Linux or Windows runs on top of a host operating system with virtualized access to the underlying hardware. Like virtual machines, containers allow you to package your application together with libraries and other dependencies, providing isolated environments for running your software services. As you'll see below however, the similarities end here as containers offer a far more lightweight unit for developers and IT Ops teams to work with, carrying a myriad of benefits.

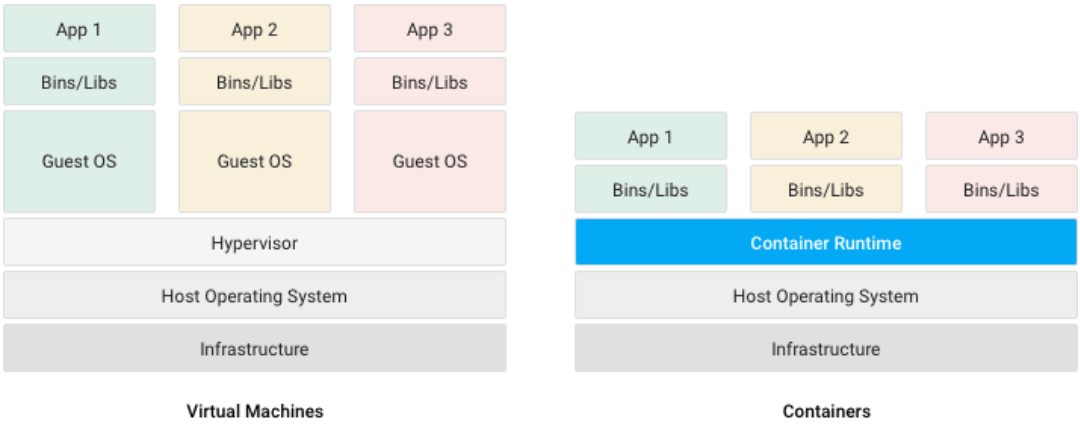


Figure 2.2: *Comparing containers and virtual machines*

Instead of virtualizing the hardware stack as with the virtual machines approach, containers virtualize at the operating system level, with multiple containers running atop the OS kernel directly. This means that containers are far more lightweight: they share the OS kernel, start much faster, and use a fraction of the memory compared to booting an entire OS.

2.3 Kubernetes

The official Kubernetes documentation provides a thorough introduction to its concepts and principles. The basic information mentioned in this section is obtained from [23].

> Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem.

In a production environment, the management of containers that run the applications is required, along with ensuring that there is no downtime. Kubernetes solves that problem by providing a framework to run distributed systems resiliently. It takes care of scaling requirements, failover, deployment patterns, and more.

Among others, Kubernetes provides service discovery and load balancing, self-healingness[1], and storage orchestration.

[1] Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to user-defined health check, and doesn't advertise them to clients until they are ready to serve.

Finally, the biggest advantage Kubernetes offers is its extensibility. This means that the cluster administrator may define custom resources and implement the logic backing their management. These are called *Custom Resource Definitions* (CRD), the rationale is called *Controller Pattern* while the component implementing the pattern is the *Controller*. All these compose what we call an *Operator*.

2.3.1 Container Storage Interface

A nice introductory to the Container Storage Interface (CSI) can be accessed at [24]. The aim of CSI is to unify the storage interface of Container Orchestrator Systems (COs) like Kubernetes, Mesos, Docker swarm, cloud foundry, etc. combined with storage vendors like Ceph, Portworx, NetApp, etc. This means, implementing a single CSI for a storage vendor is guaranteed to work with all COs.

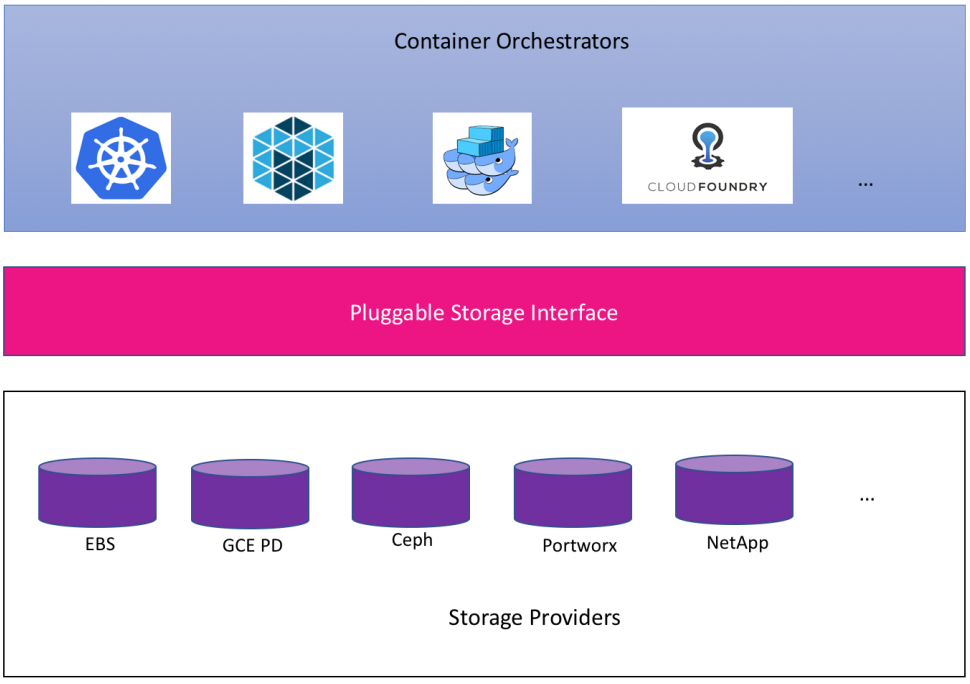


Figure 2.3: *Container Storage Interface diagram*

Container Storage Interface replaced the custom volume plugins that served all the storage needs for container workloads in case of Kubernetes. They have to be part of the Kubernetes core, as shown in figure 2.4, so they come with numerous drawbacks. Their development was tightly coupled and dependent on Kubernetes releases, Kuber-

netes developers and community were responsible for testing and maintaining them, while these plugins could crash even critical Kubernetes components. These are just some of their disadvantages.

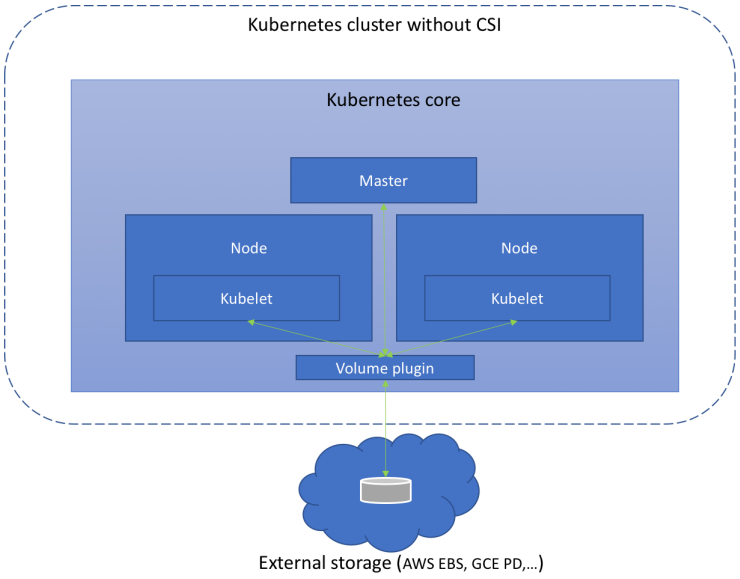


Figure 2.4: *Volume plugins diagram*

Persistent Volumes

Extracted from the documentation[25], the `PersistentVolume` subsystem provides an API for users and administrators that abstracts details of how storage is provided from how it is consumed. To do this, two new API resources are introduced: `PersistentVolume` and `PersistentVolumeClaim`.

A `PersistentVolume` (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes. It is a resource in the cluster just like a node is a cluster resource. PVs are volume plugins like Volumes, but have a lifecycle independent of any individual pod that uses the PV. This API object captures the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system.

A `PersistentVolumeClaim` (PVC) is a request for storage by a user. It is similar to a pod. Pods consume node resources and PVCs consume PV resources. Pods can request specific levels of resources (CPU and Memory). Claims can request specific size and access modes (e.g., can be mounted once read/write or many times read-only).

Volume Snapshots

Similarly, we may find the official documentation for Volume Snapshots in [26], where it is stated that the same way API resources `PersistentVolume` and `PersistentVolumeClaim` are used to provision volumes for users and administrators, `VolumeSnapshotContent` and `VolumeSnapshot` API resources are provided to create volume snapshots for users and administrators.

A `VolumeSnapshotContent` is a snapshot taken from a volume in the cluster that has been provisioned by an administrator. Similar to `PersistentVolume`, it is a resource to the cluster.

A `VolumeSnapshot` is a request for snapshot of a volume by a user. It is equivalent to a `PersistentVolumeClaim`.

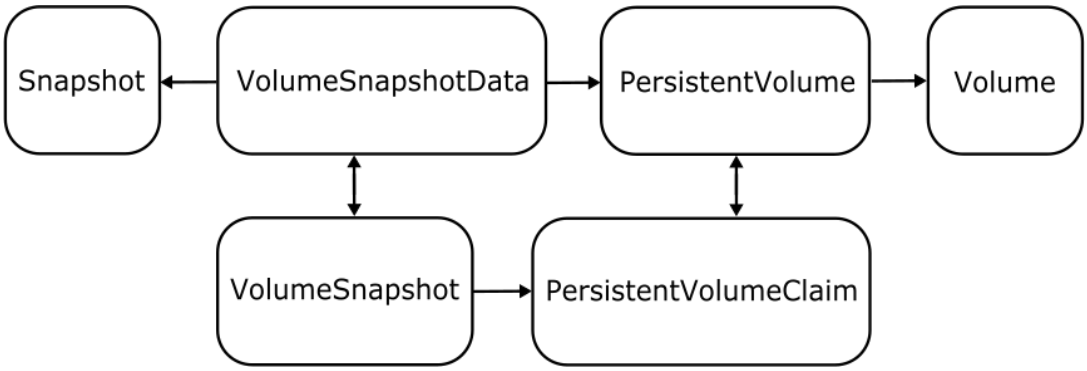


Figure 2.5: *Persistent Volumes & Volume Snapshots relationship diagram*

2.4 Argo

The official website of Argo is [27] and contains detailed descriptions, documentation and other useful related links. In this thesis we focus on Argo Workflows, so by mentioning Argo or Argo Workflows we will be referring to the same thing.

Argo Workflows is an open source container-native workflow engine for orchestrating parallel jobs on Kubernetes. Argo Workflows is implemented as a Kubernetes CRD (Custom Resource Definition). It is, essentially, a Kubernetes Operator responsible for the deployment of workflows.

Users may easily define workflows where each step in the workflow is a container and

model multi-step workflows as a sequence of tasks or capture the dependencies between tasks using a graph (DAG). Furthermore, they can easily run compute intensive jobs for machine learning or data processing in a fraction of the time using Argo Workflows on Kubernetes. Last but not least, running CI/CD pipelines natively on Kubernetes without configuring complex software development products is possible through this operator.

Argo is designed from the ground up for containers without the overhead and limitations of legacy VM and server-based environments, while at the same time it is cloud agnostic and can run on any Kubernetes cluster. Lastly, the user may easily orchestrate highly parallel jobs on Kubernetes.

2.5 Kubeflow

> Kubeflow is the machine learning toolkit for Kubernetes.

The Kubeflow project is dedicated to making deployments of machine learning workflows on Kubernetes simple, portable and scalable. The main goal is not to recreate other services, but to provide a straightforward way to deploy best-of-breed open-source systems for ML to diverse infrastructures.

Kubeflow started as an open sourcing of the way Google ran TensorFlow internally, based on a pipeline called TensorFlow Extended. It began as just a simpler way to run TensorFlow jobs on Kubernetes, but has since expanded to be a multi-architecture, multi-cloud framework for running entire machine learning pipelines.

The objective is to make scaling ML models and deploying them to production as simple as possible, by letting Kubernetes do what it's great at. That is, easy, repeatable and portable deployments on a diverse infrastructure. For instance, starting from a laptop locally, then being transferred to an ML rig and a training cluster to finally move to the production cluster. Kubernetes can easily deploy and manage loosely-coupled microservices and scale based on demand.

2.5.1 Kubeflow Pipelines

Kubeflow Pipelines (KFP) is a platform for building and deploying portable, scalable machine learning workflows based on Docker containers and is one of the Kubeflow core components. It's automatically deployed during Kubeflow deployment.

The Kubeflow Pipelines platform consists of a user interface (UI) for managing and tracking experiments, jobs, and runs, along with an engine for scheduling multi-step ML workflows. It also comes with an SDK for defining and manipulating pipelines and components. Besides that, there are notebooks for interacting with the system using the SDK.

KFP provides end-to-end orchestration, enabling and simplifying the orchestration of machine learning pipelines. On top of that, it is easy for users to try numerous ideas and techniques and manage various trials/experiments. In addition, it enables re-using components and pipelines to quickly create end-to-end solutions without having to rebuild each time.

3 Design

This chapter is vital to the comprehension of both the purpose and the rationale of the thesis. Here, we unravel the flow of the reasoning that led us to the implementation of the feature.

Then, we present the first holistic iteration of the design, the document which was publicly put to discussion. We thoroughly analyze the fundamental design decisions that we had to make during this process.

Next, we move on to describe the final version of the design document. This, totally refurbished version, was a result of the community feedback as well as the discussions with the developers. With this collective process, we achieved a design which is generalized and can be extended even further to cover any similar need.

This chapter is heavily based on the design document iterations discussed publicly at the KFP project[1].

3.1 Design Rationale and Goals

As we have already elaborated during the Introduction of this thesis, machine learning is all about data. This means that the steps of a pipeline should be able to exchange multi-GB of data by performing standard file input/output (I/O) on mounted Persistent Volumes (PVs) without having to download data from external object stores, nor

[1] Extend the DSL with support for Persistent Volumes and Snapshots: `https://github.com/kubeflow/pipelines/issues/801`

to upload to them.

To manipulate storage volumes, we use standard, vendor-neutral Kubernetes primitives (namely `PersistentVolumeClaim`, and `VolumeSnapshot` API objects), but without having the user to manipulate those Kubernetes objects manually. Instead, the goal is for the user to declare the way steps use volumes as pipeline resources for data exchange, with an **intuitive DSL syntax**, similarly to what they do for the rest of their pipeline resources.

Extending the DSL to support the use of PVs for data exchange will make the use of Kubeflow Pipelines on-premises (on-prem) much easier, and will also enable the use of **advanced storage functionality** offered by modern storage, that is snapshots, as a way to **gain insight** on how a multi-GB pipeline executes at every step.

Supporting volumes natively in the DSL makes data in a `PersistentVolume` just another piece of information being passed from step to step, and can be used as input by the Pipelines compiler to **extract dependency information** between individual tasks.

All in all, the goal is to make `PersistentVolumes` and Snapshots of `PersistentVolumes` **top-level objects** in the Pipelines DSL, to enable full programmatic manipulation, and to hide the low-level details of managing Kubernetes API objects directly.

3.2 Initial Design Document

To make all of the above possible, we create two new entities: `PipelineVolumes` and `PipelineVolumeSnapshots`. We also simplify their usage by the steps of the pipeline by exposing a mechanism and extending `ContainerOp`'s definition, so as to easily mount them.

In the following subsections, we describe these abstractions as well as their usage.

3.2.1 PipelineVolumes

`PipelineVolume` is an entity of the DSL which either refers to an existing `PersistentVolumeClaim` or is responsible for the creation of a new one. It exposes an easy way to set a subset of a PVC's attributes, thus making the common case fast, but also letting

the user provide a custom object using the Kubernetes client models, which in turn makes the **full range of specifications available**.

Since these volumes contain information produced by some tasks of the pipeline, the usage of an instance of this entity by a step implies that its execution should start **after** these tasks. We achieve this by binding each instance with any necessary dependency, while also providing an expressive, functional way to extend these dependencies.

3.2.2 PipelineVolumeSnapshots

Similarly, `PipelineVolumeSnapshot` is an entity of the DSL which is either a reference to an existing `VolumeSnapshot` or a request to create one. The logic is the same as before: a commonly used subset of the resource's attributes is exposed via a simple API, but users are able to pass a custom object, too.

In contrast to `PersistentVolumeClaim`s, `VolumeSnapshot`s are **immutable**. The data contained cannot be changed by any other step. Taking that into consideration, there is no need for multiple instances referencing the same resource but having different dependencies. A `VolumeSnapshot` referenced by some instance of the DSL specification is bound to certain **immutable dependencies**.

3.2.3 Usage

In this subsection, we present some examples showing the basic usage of this design. There are also comments explaining the directed acyclic graphs (DAG) resulted from the specification, as well as the demonstrated semantics.

```
# Example 1 - Single PV: step1 // step2 // step3
#
# Note the PV is created dynamically, by the workflow engine,
# mounted at different mount points in each step,
# and that using the PV imposes no dependencies between steps.
from kfp import dsl


@dsl.pipeline(
    name="Example 1"
)
def example1():
    vol_common = dsl.PipelineVolume(
        name="mypvc",
        size="10Gi"
    )
    step1 = dsl.ContainerOp(
        name="step1",
        image="library/bash:4.4.23",
        command=["cat", "/mnt/file1"],
        volumes={"/mnt": vol_common}
    )
    step2 = dsl.ContainerOp(
        name="step2",
        image="library/bash:4.4.23",
        command=["cat", "/common/file2"],
        volumes={"/common": vol_common}
    )
    step3 = dsl.ContainerOp(
        name="step2",
        image="library/bash:4.4.23",
        command=["cat", "/mnt3/file3"],
        volumes={"/mnt3": vol_common}
    )
```

```
# Example 2 - Multiple PVs, with dependencies between steps:
# step1 // step2 --> step3
#
# step1 and step2 run in parallel, output to the same volume,
# and only when they are done with the volume, can step3 start
# to run.
# Note this example introduces dependencies based on volume use
# by itself, in addition to any dependencies on file_outputs.
# Note the notation `vol1.after(step1, step2)` returns an object that points
# to the same underlying Persistent Volume Claim as vol1, but with dependencies
# on step1 and step2, i.e., after step1 and step2 are done using it.
from kfp import dsl
@dsl.pipeline(
    name="Example 2"
)
def example2():
    vol1 = dsl.PipelineVolume(name="mypvc",
                              size="10Gi")
    step1 = dsl.ContainerOp(
        name="step1",
        image="library/bash:4.4.23",
        command=["sh", "-c"],
        arguments=["echo 1 | tee /mnt/file1"],
        volumes={"/mnt": vol1}
    )
    step2 = dsl.ContainerOp(
        name="step2",
        image="library/bash:4.4.23",
        command=["sh", "-c"],
        arguments="echo 2|tee /mnt2/file2",
        volumes={"/mnt2": vol1}
    )
    step3 = dsl.ContainerOp(
        name="step3",
        image="library/bash:4.4.23",
        command=["sh", "-c"],
        arguments=["cat /mnt/file1 /mnt/file2"],
        volumes={"/mnt": vol1.after(step1, step2)}
    )
```

```
# Example 3 - Single PV, used by consecutive steps:
# step1 --> step2 --> step3
#
# Note use of `stepX.volume` to refer to the single volume
# in stepX.volumes.
# Note how referring to stepX.volumes gets a
# reference to the same underlying volume but *after* stepX
# is complete. This is equivalent to the `vol1.after(step1)` syntax
# used in the previous example.
from kfp import dsl
@dsl.pipeline(
    name="Example 3"
)
def example3():
    vol1 = dsl.PipelineVolume(
        name="mypvc",
        size="10Gi"
    )

    step1 = dsl.ContainerOp(
        name="step1",
        image="library/bash:4.4.23",
        command=["sh", "-c"],
        arguments=["echo 1|tee /data/file1"],
        volumes={"/data": vol1}
    )
    step2 = dsl.ContainerOp(
        name="step2",
        image="library/bash:4.4.23",
        command=["sh", "-c"],
        arguments=["cp /data/file1 /data/file2"],
        volumes={"/data": step1.volume}
    )
    step3 = dsl.ContainerOp(
        name="step3",
        image="library/bash:4.4.23",
        command=["cat", "/mnt/file1", "/mnt/file2"],
        volumes={"/data": step2.volume}
    )
```

```
# Example 4 - Multiple PVs, with dynamic snapshots
#             and clones:
# step1 --> step2 --> step3
#
# Note creation of dsl.PipelineVolumeSnapshot`objects from
# dsl.PipelineVolume objects.
# Note dynamic creation of dsl.PipelineVolume objects from
# snapshots, when we pass instances of dsl.PipelineVolumeSnapshot to
# the `volumes` named argument of ContainerOp instances.
from kfp import dsl


@dsl.pipeline(
    name="Example 4"
)
def example4(url):
    vol1 = dsl.PipelineVolume(
        name="mypvc",
        size="10Gi"
    )
    step1 = dsl.ContainerOp(
        name="ingest data as a gzipped file",
        image="google/cloud-sdk:216.0.0",
        command=["sh", "-c"],
        arguments=["gsutil cat %s | tee /data/file1.gz" % url],
        volumes={"/data": vol1}
    )
    step1_snap = dsl.PipelineVolumeSnapshot(step1.volume)
    # Or: step1_snap = step1.volume.snapshot()
    # Or: step1_snap = dsl.PipelineVolumeSnapshot(vol1.after(step1))
    # Or: step1_snap = vol1.after(step1).snapshot()

    step2 = dsl.ContainerOp(
        name="gunzip",
        image="library/bash:4.4.23",
        command=["gunzip", "/data/file1.gz"],
        volumes={"/data": step1_snap}
    )
    step2_snap = dsl.PipelineVolumeSnapshot(step2.volume)
```

```
step3 = dsl.ContainerOp(
    name="output data to stdout",
    image="library/bash:4.4.23",
    command=["cat", "/data/file1"],
    volumes={"/data": step2_snap}
)
step3_snap = dsl.PipelineVolumeSnapshot(step3.volume)
```

In figure 3.1 we visualize the design, the way DSL entities interact as well as their correspondence between those and Kubernetes primitives. Finally, figure 3.2 is a pipeline graph as shown by the Kubeflow Pipelines UI.

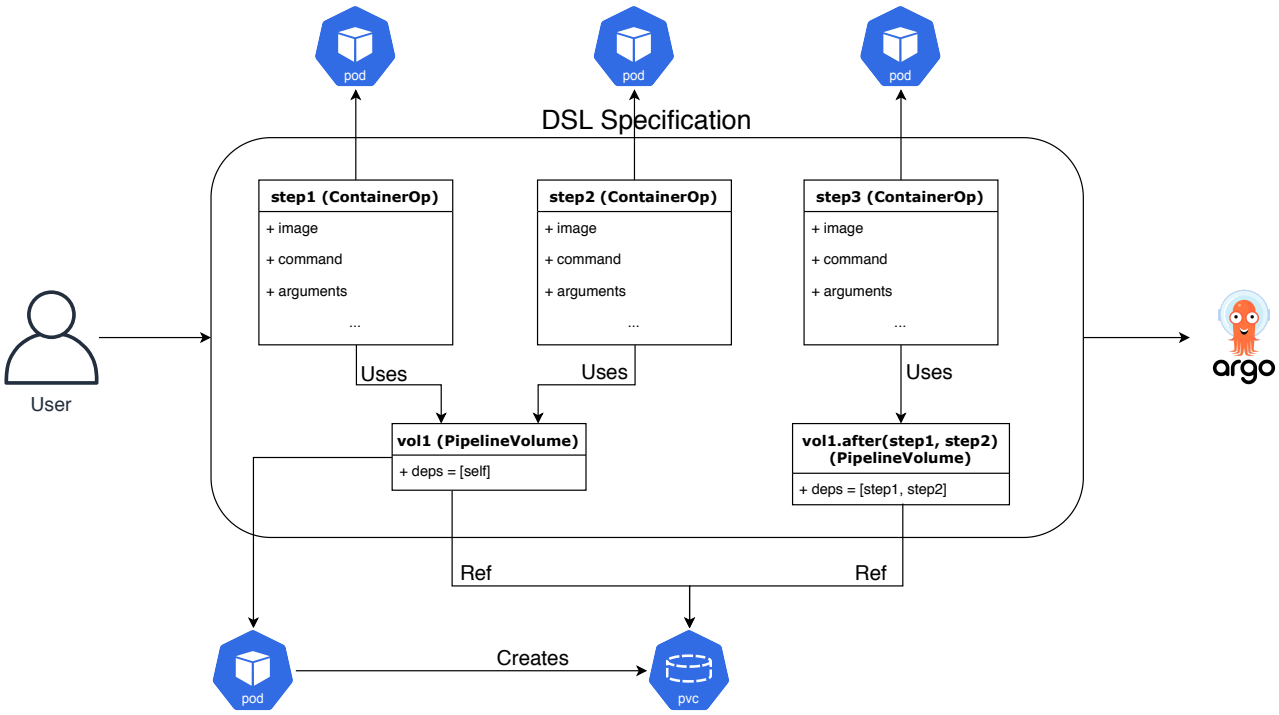


Figure 3.1: *Initial Design Document - Visualization of Example 2*

3.3 Reviews

The initial design document had a huge reflection on users' needs, and fulfilled them greatly. However, having the source code available and people testing it, everything can be clearer, as far as the design is concerned. Thus, once the GitHub[28] pull request (PR) [29] was submitted, the maintainers of the project reviewed the implementation and came back with several ideas.

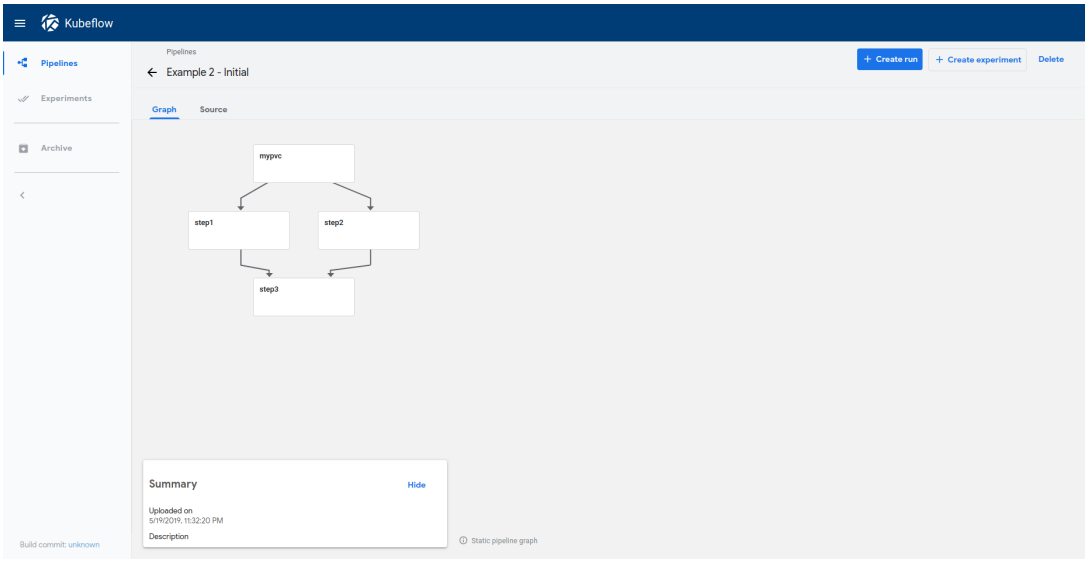


Figure 3.2: *Initial Design Document - Graph of Example 2*

Their proposal was to have an abstraction for general Kubernetes resources manipulation, and to have the entities responsible for the creation of `PersistentVolumeClaim`s and `VolumeSnapshot`s derive from it.

As a consequence, a specific and restricted purpose is served by each entity. For instance, in the initial design, a `PipelineVolume` could either mean a reference to or a creation of a `PersistentVolumeClaim`. But we will show that in the final version it is used just for the referencing part.

Such enhancement simplifies the code, makes its maintenance easier and breaks ground for further extensions. Therefore, after putting some thought on it, we came up with a totally refurbished design described in the following section.

We change the `volumes` argument and `volumes` and `volume` attributes to `pvolumes` and `pvolume` respectively, to not break the existing API and the backwards compatibility. Finally, we change the way some dependencies are expressed by not removing the redundant dependencies derived from the usage of a created PVC.

3.4 Final Design Document

In the final design document, we extend the DSL with more entities, while also modifying or even deleting the ones referred earlier in the Initial Design Document. A

detailed outline of them is contained in following subsections.

3.4.1 PipelineVolume: Simplifying the Consumption of Persistent VolumeClaims

A `PipelineVolume` inherits from the Kubernetes primitive `Volume` [30]. At its core, a volume is just a directory, possibly with some data in it, which is accessible to the Containers in a Pod. How that directory comes to be, the medium that backs it, and the contents of it are determined by the particular volume type used.

There are several types of volumes, although we focus on our main usage, that is the references to PVCs.

What is different about `PipelineVolume`s is that they carry dependencies. As far as this is concerned, we adopt the same principles described in the first design document.

However, this entity is no longer responsible for the `PersistentVolumeClaim` resource manipulation.

3.4.2 PipelineVolumeSnapshot: The End of an Era

This entity is totally removed.

3.4.3 ResourceOp: Rendering a New Argo Template Available

This is the basic entity used for the manipulation of arbitrary Kubernetes resources. It is the corresponding DSL class which models the Argo resource-type templates.

The user may pass some action to perform on a Kubernetes resource (defaults to `create`) and an object with the resource's specification. They may also provide conditions regarding the success or the failure of the step.

One last and really important functionality is the ability to export any of the resource's attributes as output parameters of this task, which can then be used by any step of pipeline. By default, we export the name and the whole specification of the resource.

While sketching this entity, we discovered many similarities to the `ContainerOp`, which models the Argo container-type template, as it was expected. Hence, we introduce a new base class, called `BaseOp`, that is the parent of the aforementioned two classes and enfolds all the common attributes and methods.

3.4.4 VolumeOp - VolumeSnapshotOp: Simplifying the Creation of PersistentVolumeClaims and VolumeSnapshots

`VolumeOp` and `VolumeSnapshotOp` are two subclasses of `ResourceOp`. The first aims to render the creation of `PersistentVolumeClaim`s easier, while the second simplifies the creation of `VolumeSnapshot`s.

Those classes are only responsible for the manipulation of the resources. They both expose an easy API to make the common case fast, in the same way `PipelineVolume` and `PipelineVolumeSnapshot` did in the initial design document.

Especially for the `VolumeOp`s, we expose a `volume` attribute, which is a `PipelineVolume`, so as to seamlessly mount this PVC on `ContainerOp`.

3.4.5 Usage

In this subsection, we show the examples presented in subsection 3.2.3 refactored according to the final design document.

```
# Example 1 - Single PV: step1 // step2 // step3
#
# Note the PV is created dynamically, by the workflow engine,
# mounted at different mount points in each step,
# and that using the PV imposes no dependencies between steps.
from kfp import dsl


@dsl.pipeline(
    name="Example 1"
)
def example1():
    vol_common = dsl.VolumeOp(
        name="create-mypvc",
        resource_name="mypvc",
        size="10Gi"
    )
    step1 = dsl.ContainerOp(
        name="step1",
        image="library/bash:4.4.23",
        command=["cat", "/mnt/file1"],
        pvolumes={"/mnt": vol_common.volume}
    )
    step2 = dsl.ContainerOp(
        name="step2",
        image="library/bash:4.4.23",
        command=["cat", "/common/file2"],
        pvolumes={"/common": vol_common.volume}
    )
    step3 = dsl.ContainerOp(
        name="step2",
        image="library/bash:4.4.23",
        command=["cat", "/mnt3/file3"],
        pvolumes={"/mnt3": vol_common.volume}
    )
```

```
# Example 2 - Multiple PVs, with dependencies between steps:
# step1 // step2 --> step3
#
# step1 and step2 run in parallel, output to the same volume,
# and only when they are done with the volume, can step3 start
# to run.
# Note this example introduces dependencies based on volume use
# by itself, in addition to any dependencies on file_outputs.
# Note the notation `vop.after(step1, step2)` returns an object that points
# to the same underlying Persistent Volume Claim as vop.volume, but with
# dependencies on step1 and step2: after step1 and step2 are done using it.

from kfp import dsl


@dsl.pipeline(
    name="Example 2"
)
def example2():
    vop = dsl.VolumeOp(
        name="create-mypvc",
        resource_name="mypvc",
        size="10Gi"
    )
    step1 = dsl.ContainerOp(
        name="step1",
        image="library/bash:4.4.23",
        command=["sh", "-c"],
        arguments=["echo 1 | tee /mnt/file1"],
        pvolumes={"/mnt": vop.volume}
    )
    step2 = dsl.ContainerOp(
        name="step2",
        image="library/bash:4.4.23",
        command=["sh", "-c"],
        arguments="echo 2|tee /mnt2/file2",
        pvolumes={"/mnt2": vop.volume}
    )
```

```
    step3 = dsl.ContainerOp(
        name="step3",
        image="library/bash:4.4.23",
        command=["sh", "-c"],
        arguments=["cat /mnt/file1 /mnt/file2"],
        pvolumes={"/mnt": vop.volume.after(step1, step2)}
    )
```

```
# Example 3 - Single PV, used by consecutive steps:
# step1 --> step2 --> step3
#
# Note use of `stepX.pvolume` to refer to the single volume
# in stepX.pvolumes.
# Note how referring to stepX.pvolumes gets a
# reference to the same underlying volume but *after* stepX
# is complete. This is equivalent to the `vop.volume.after(step1)` syntax
# used in the previous example.
from kfp import dsl
@dsl.pipeline(
    name="Example 3"
)
def example3():
    vop = dsl.VolumeOp(name="create-mypvc",
                               resource_name="mypvc",
                               size="10Gi")

    step1 = dsl.ContainerOp(
        name="step1",
        image="library/bash:4.4.23",
        command=["sh", "-c"],
        arguments=["echo 1|tee /data/file1"],
        pvolumes={"/data": vop.volume}
    )
    step2 = dsl.ContainerOp(
        name="step2",
        image="library/bash:4.4.23",
        command=["sh", "-c"],
        arguments=["cp /data/file1 /data/file2"],
        pvolumes={"/data": step1.pvolume}
    )
    step3 = dsl.ContainerOp(
        name="step3",
        image="library/bash:4.4.23",
        command=["cat", "/mnt/file1", "/mnt/file2"],
        pvolumes={"/data": step2.pvolume}
    )
```

```
# Example 4 - Multiple PVs, with dynamic snapshots
# step1 --> step2 --> step3
#
# Note creation of dsl.VolumeSnapshotOp`objects from
# dsl.PipelineVolume objects.

from kfp import dsl


@dsl.pipeline(
    name="Example 4"
)
def example4(url):
    vop = dsl.VolumeOp(
        name="create-mypvc",
        resource_name="mypvc",
        size="10Gi"
    )

    step1 = dsl.ContainerOp(
        name="ingest data as a gzipped file",
        image="google/cloud-sdk:216.0.0",
        command=["sh", "-c"],
        arguments=["gsutil cat %s | tee /data/file1.gz" % url],
        pvolumes={"/data": vop.volume}
    )

    step1_snap = dsl.VolumeSnapshotOp(
        name="Snapshot step1",
        resource_name="step1-snap",
        volume=step1.pvolume
    )

    step2 = dsl.ContainerOp(
        name="gunzip",
        image="library/bash:4.4.23",
        command=["gunzip", "/data/file1.gz"],
        pvolumes={"/data": step1.pvolume}
    )
```

```
step2_snap = dsl.VolumeSnapshotOp(
    name="Snapshot step2",
    resource_name="step2-snap",
    volume=step2.pvolume
)

step3 = dsl.ContainerOp(
    name="output data to stdout",
    image="library/bash:4.4.23",
    command=["cat", "/data/file1"],
    pvolumes={"/data": step2.pvolume}
```

Similarly, in figure 3.3 we present the way figure 3.1 gets represented according to the new version of the DSL. And finally, figure 3.4 is a pipeline graph as shown by the Kubeflow Pipelines UI.

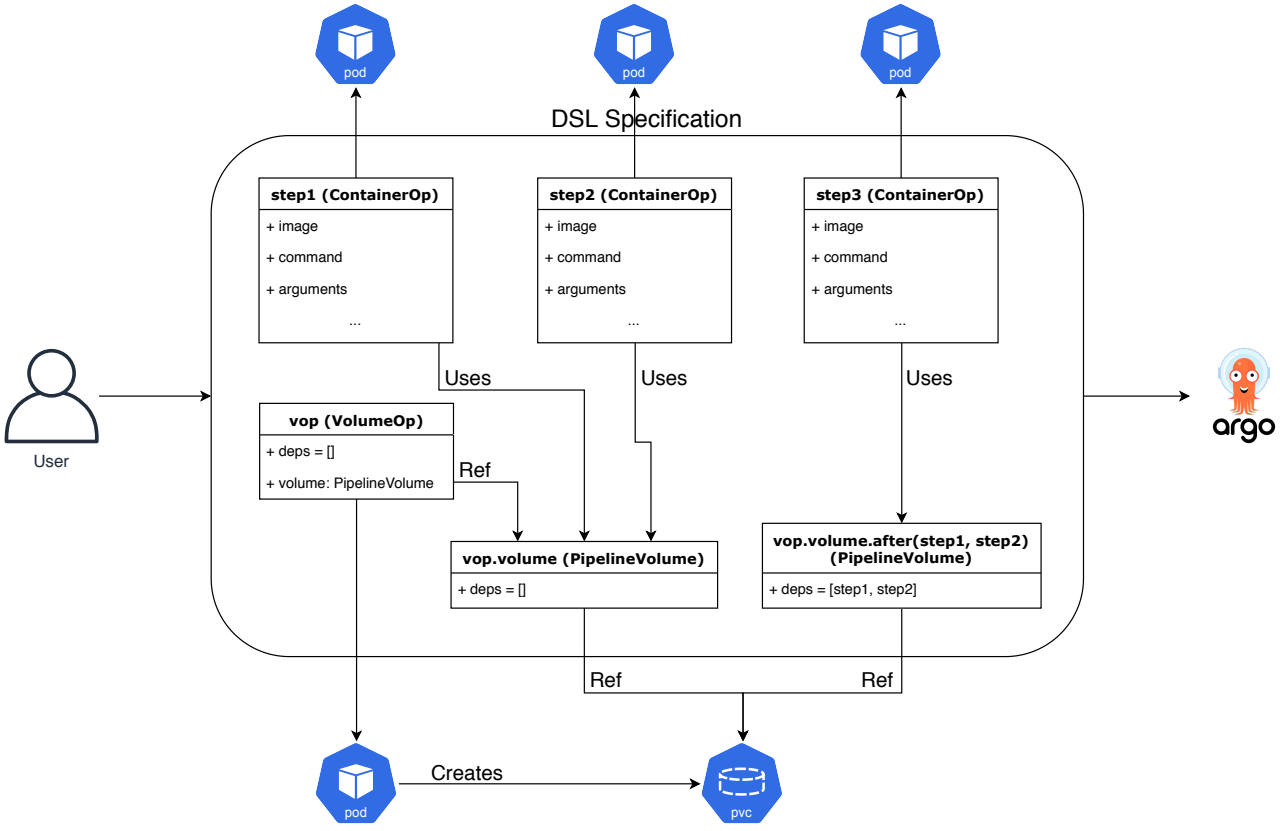


Figure 3.3: *Final Design Document - Visualization of Example 2*

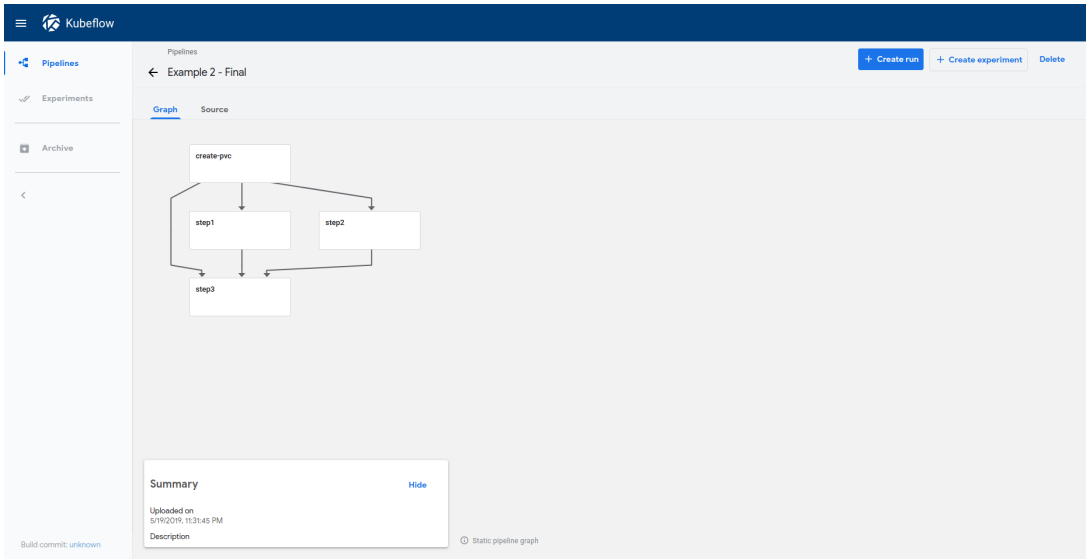


Figure 3.4: *Final Design Document - Graph of Example 2*

4 Implementation

In this chapter, we give a detailed description of the DSL and the Compiler extension process. We analyze 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 the software patch we wrote to implement important functionality which was missing from Argo. Finally, we describe in detail the methods and tools used to ensure the correctness of our code.

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 complete source code regarding the upgrade of Kubeflow Pipelines SDK functionality is available in the merged pull request `https://github.com/kubeflow/pipelines/pull/926`.

4.1 Initial Design Document

This section is dedicated to the implementation of our initial design document.

4.1.1 PipelineVolume

At this point, we show the object's attributes.

`name`, `deps`, `inputs` and `is_exit_handler` are attributes which are common among all operator types of the DSL and promotes the uniform parsing of objects by the com-

piler.

In case users want to mount an existing `PersistentVolumeClaim`, they need to set the corresponding argument to the constructor and `pvc` attribute holds that name or `PipelineParam`.

The attributes `from_snapshot` and `from_scratch` are used as indicators which help digging out dependencies, when the instance gets used. These booleans are `False` when `pvc` is set.

`k8s_resource` attribute is the Kubernetes model describing the PVC which gets created or mounted.

When iterating through the `PipelineVolumes` defined, it is `create` attribute that helps detecting whether we need to create a `template` for that instance. That is, when `from_snapshot` or `from_scratch` are `True` and the instance does not come out of an `after()` call.

```
class PipelineVolume(object):
    name                str
    deps                Set[str]
    pvc                 str
    k8s_resource        k8s_client.V1PersistentVolumeClaim
    from_snapshot       bool
    from_scratch        bool
    create              bool

    inputs              List[PipelineParam]
    is_exit_handler     bool
```

Code Listing 4.1: *The attributes of* `PipelineVolume` *class according to the initial design presented in pseudocode.*

Following, we present the method `after()` of the class.

This method is required in order to create a new instance of `PipelineVolume` having its dependencies extended. That way, we bind the volume instance with the tasks manipulating it. Consequently, these dependencies will be properly parsed when used by a `ContainerOp` and the task will run after them.

Moreover, a nice feature we have supplemented the simple implementation of `after()`

method with is the removal of all the old dependencies which become redundant by the new dependencies.

```
def after(*ops):
    new_instance = PipelineVolume(
        pvc=self.pvc,
        k8s_resource=self.k8s_resource,
        create=False
    )
    new_instance.name = self.name
    new_instance.deps = ops

    for dep in self.deps:
        if dep not implied by any of ops:
            new_instance.deps.add(dep)

    return new_instance
```

Code Listing 4.2: *The implementation of* `after()` *method in pseudocode.*

Finally, `PipelineVolume` class comes with a `snapshot()` method. This is an easy way to create a `PipelineVolumeSnapshot` instance out of a `PipelineVolume` instance.

Obviously, the task creating the `VolumeSnapshot` will run after the dependencies carried by that `PipelineVolume` instance, maintaining the information regarding the origin of the data contained in the PVC.

```
def snapshot(name, snapshot_class):
    return PipelineVolumeSnapshot(
        pipeline_volume=self,
        name,
        snapshot_class
    )
```

Code Listing 4.3: *The implementation of* `snapshot()` *method in pseudocode.*

4.1.2 PipelineVolumeSnapshot

The structure of `PipelineVolumeSnapshot` class is quite simpler.

The attributes `name`, `deps`, `k8s_resource`, `inputs` and `is_exit_hander` serve the same purposes as in `PipelineVolume`.

`new_snap` attribute indicates whether this instance refers to a `VolumeSnapshot` which is to be created, when `True`, or it is a reference to an existing `VolumeSnapshot` in the cluster.

```
class PipelineVolumeSnapshot(object):
    name                str
    deps                Set[str]
    k8s_resource        k8s_client.V1PersistentVolumeClaim
    new_snap            bool

    inputs              List[PipelineParam]
    is_exit_handler     bool
```

Code Listing 4.4: *The attributes of* `PipelineVolumeSnapshot` *class according to the initial design presented in pseudocode.*

4.1.3 Compiler Handling

In this subsection we present the compiler's approach simplified, so as not to tire readers with very specific functionalities which would diverge them from the main purpose of this thesis.

In the beginning, the compiler does some parsing identifying the input parameters and the implied dependencies between tasks.

Subsequently, the workflow's YAML composition takes place. The compiler generates the templates expressing the DAG of the pipeline, following with each operator's template.

`_op_to_template()` is the function creating the `ContainerOp` templates. We extend the compiler with the functions `_vol_to_template()` and `_snap_to_template()`, which append the templates list with those required for `PipelineVolumes` and `PipelineVolumeSnapshots` respectively.

The Code Listings below show the resource templates generated for the creation of a new PVC and the creation of a new `VolumeSnapshot`. The `manifest` field, un-

der `resource` field, contains the YAML representation of the `k8s_resource` attribute each of the `PipelineVolume` and `PipelineVolumeSnapshot` has.

We chose to output as parameters some attributes of the created resource. For the PVCs, we output their name, whereas for the `VolumeSnapshots` we export their name and their `restoreSize`[1].

```
name: mypvc
inputs:
  parameters:
    - ...
resource:
  action: create
  manifest: ...
outputs:
  parameters:
    - name: mypvc-name
      valueFrom:
        jsonPath: '{.metadata.name}'
```

Code Listing 4.5: *The compiler's generated template for a* `PipelineVolume` *instance referring to the creation of a new* `PersistentVolumeClaim`.

```
name: mysnap
inputs:
  parameters:
    - ...
    - ...
resource:
  action: create
  manifest: ...
outputs:
  parameters:
```

[1] The `restoreSize` is the minimum size required for a PVC to be created by that `VolumeSnapshot`.

```
    - name: mysnap-name
      valueFrom:
        jsonPath: '{.metadata.name}'
    - name: mysnap-size
      valueFrom:
        jsonPath: '{.status.restoreSize}'
```

Code Listing 4.6: *The compiler's generated template for a* `PipelineVolume` *instance referring to the creation of a new* `VolumeSnapshot`.

4.2 Final Design Document

Between the two major versions of the design another quite big pull request[31] was merged. This contained some great work regarding some functionality extension and major code refactoring. This exposed us to high quality code and actually helped us improve the code we delivered.

So, similarly to the previous section, here we give a rundown of the implementation of the merged design.

4.2.1 BaseOp: Noticing the Lowest Common Ancestor of Two Templates

`BaseOp` is a new class we introduce. It holds every attribute and method which is common between all of the Argo templates.

That class may then be the base for every extension regarding the supported templates of the DSL. As of that time, `ContainerOp` is the only subclass deriving from `BaseOp`, and implements Argo's `ContainerTemplate`.

```
class BaseOp(object):
    name            str
    inputs          List[PipelineParam]
    dependent_names List[str]
    node_selector   dict
```

```
volumes          List[V1Volume]
tolerations      List[V1Toleration]
pod_annotations  dict
pod_labels       dict
num_retries      int
sidecars         List[Sidecar]
is_exit_hander   bool

attrs_with_pipelineparams = [
    "node_selector", "volumes", "pod_annotations", "pod_labels",
    "num_retries", "sidecars", "tolerations"
]
```

Code Listing 4.7: *The attributes of* `BaseOp` *class in pseudocode.*

To not diverge from the focus of this thesis, we prefer not to elaborate on the purpose of all of these attributes, that mostly refer to Kubernetes or Argo specifics. Nevertheless, we will mention the `attrs_with_pipelineparams` motive, a compiler specific attribute.

A very important job of the compiler is to recursively iterate through DSL entities, look for `PipelineParams` (some of which may imply dependencies) and replace them according to the Argo way of referencing parameters. For that procedure to take place, the compiler needs to know what fields of each entity should be iterated through. The attribute `attrs_with_pipelineparams` serves that cause.

Finally, note that the class `BaseOp` is just a base class. We have it to avoid duplicate code and to easily extend and maintain the DSL. It is also just part of any Argo template. Hence, it should not be used by data scientists and, therefore, it is not exposed to them.

4.2.2 ResourceOp: Incorporating the New Template

The DSL entity `ResourceOp` implements another Argo template type: the `Resource Template`. It derives from `BaseOp`, in a similar fashion to `ContainerOp`, and it is also a task of the pipeline.

This class is the root of the breaking change we introduce within the scope of the thesis.

```
class ResourceOp(BaseOp):
    k8s_resource        K8sModel
    attribute_outputs   dict
    outputs             dict
    resource            Resource

    attrs_with_pipelin_params = (
        super().attrs_with_pipelineparams +
        ["resource", "k8s_resource", "attribute_outputs"]
    )

class Resource(object):
    action              str
    merge_strategy      str
    success_condition   str
    failure_condition   str
```

Code Listing 4.8: *The extra attributes of* `ResourceOp` *class and the definition of* `Resource` *class in pseudocode.*

- The attribute `k8s_resource` is the representation of the Kubernetes model to be manipulated.
- `attribute_outputs` enable the user to request output parameters from that step. This attribute is a dictionary having parameter names as keys and `jsonPaths`, within the Kubernetes resource, as values. By default, we output the resource's name as well as its whole manifest, but the user may even overwrite these two.
- `outputs` is similar to the previous dictionary but its values are the corresponding `PipelineParam`s, in order to use and pass them between tasks.
- The last attribute, `attrs_with_pipelineparams`, serves the same purpose, as explained in 4.2.1 and is extended to include all the extra attributes where `PipelineParam`s may be found.
- `Resource` is a class we implemented to match the `resource` field of the Argo template. This structure contains everything which is specific to the Argo template for a resource. In other words, it contains the information for the action

to be performed (`get`, `create`, `apply`, `delete`, `replace`, `patch`), the `merge_strategy` in the case of `patch`, some conditions regarding the success and the failure of the task and, lastly, the manifest of the resource (that is, the conversion of the `k8s_resource` to YAML string).

`ResourceOp` actually solves the problem we attempt to tackle: the dynamic creation of resources. Especially PVCs and `VolumeSnapshots` to perform data management and data versioning.

Although, it is widely accepted that defining a resource via the Kubernetes client requires a lot of knowledge over Kubernetes as well as boilerplate code. In addition, there can be no dependency implication having that entity by itself. And, lastly, even mounting PVCs on tasks requires lots of code and Kubernetes specifics.

We solve all these issues by introducing `VolumeOp`, `VolumeSnapshotOp` and some API extension for `ContainerOps`.

4.2.3 What Happened to PipelineVolume?

Following our decision described in 3.4.1, the class `PipelineVolume` is shown below.

We extended the standard `Volume`'s attributes with `dependent_names` (the name was chosen for the sake of uniformity).

We simplified the creation of a `PipelineVolume` referencing a PVC using `pvc` argument in the constructor. Last but not least, we simplified the creation of a `PipelineVolume` from an existing `Volume` (or an inherited type) exposing a `volume` argument in the constructor.

```
class PipelineVolume(k8s_client.V1Volume)
    dependent_names List[str]

    def __init__(self,
                 pvc,
                 volume)
```

Code Listing 4.9: *The extra attributes of* `PipelineVolume` *class and the definition of its constructor in pseudocode.*

The method `snapshot()` is removed as it seemed superfluous, while `after()` method is preserved.

```
def after(*ops):
    new_instance = self.__class__(volume=self)
    new_instance.dependent_names = [op.name for op in ops]

    for olddep in self.dependent_names:
        if olddep not implied by any of ops:
            new_instance.dependent_names.append(olddep)

    return new_instance
```

Code Listing 4.10: *The method* `after()` *of* `PipelineVolume` *class in pseudocode.*

4.2.4 VolumeOp

The class `VolumeOp` makes the common case fast as far as the creation of `PersistentVolumeClaims` is concerned. It encapsulates that part of the functionality we have presented in 4.1.1.

Deriving from `ResourceOp`, it has the same methods just one additional attribute, the `volume`. This is a `PipelineVolume`, as explained in 4.2.3, which can be used seamlessly by `ContainerOps`.

The difference is focused on the arguments of its constructor, where we expose some of the PVC's options and help users define simple resources of this kind. The potentiality of passing a fully defined specification using the Kubernetes client is available, too.

Finally, we append the `attribute_outputs` in such manner to get the size of the bound `PersistentVolume` as an output parameter. That is the actual size backed by the PVC and it is always greater or equal than the requested size.

```
class VolumeOp(ResourceOp):
    volume  PipelineVolume

    def __init__(self,
                 resource_name,
```

```
                size,
                storage_class,
                modes,
                annotations,
                data_source,
                **kwargs)
```

Code Listing 4.11: *The extra attributes of* `VolumeOp` *class and the definition of its constructor in pseudocode.*

4.2.5 VolumeSnapshotOp

Totally similar to `VolumeOp`, `VolumeSnapshotOp` is the DSL entity making the creation of `VolumeSnapshots` simple.

We still have the `VolumeSnapshot`'s `restoreSize` as an output parameter of `VolumeSnapshotOp`, and it is actually an extra entry in the `attribute_outputs`.

```
class VolumeSnapshotOp(ResourceOp):
    snapshot    k8s_client.V1TypedLocalObjectReference

    def __init__(self,
                 resource_name,
                 pvc,
                 snapshot_class,
                 annotations,
                 volume,
                 **kwargs)
```

Code Listing 4.12: *The extra attributes of* `VolumeSnapshotOp` *class and the definition of its constructor in pseudocode.*

4.2.6 Compiler Handling

The changes to the compiler regard mostly adjustments and conflict resolving, which are required because of the big refactoring performed by [31].

Furthermore, since the DSL entities which refer to templates are derivations from the same base class, we take advantage of their common structure and differentiate them only when that is needed. In other words, those differences concern the `container` and `resource` parts of the templates. Apart from the resource and container specifics, their output parameters are divergent, too. `ContainerOp`s refer to paths in their filesystem, while `ResourceOp`s refer to `jsonPath`s.

4.3 Testing

In this section, we present the testing methods used to verify the correctness of all the functions and their output. The fact that the developers of Kubeflow have put effort on unit, functional and end-to-end testing has helped us a lot. Actually, this procedure revealed minor bugs that we eliminated. Consequently, these extra iterations made our implementation more robust and reliable.

4.3.1 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`[32] testing framework. We had to extend these by adding some corresponding tests on our newly introduced classes and methods.

4.3.2 Functional Testing

Functional testing is a quality assurance process and a type of black-box testing that bases its test cases on the specifications of the software component under test. Functions are tested by feeding them input and examining the output, and internal program structure is rarely considered.

Another testing script contained in the Kubeflow Pipelines project performs this type of testing. That script tests the whole, end-to-end, compiler's functionality. Some pipeline expressed in Python DSL as well as the correct output in YAML is provided, then the Python code is compiled and compared to the expected output.

On these tests, we appended some extra pipelines testing all the new compiler features.

4.3.3 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

All in all, the primary goal of this thesis was to implement a design covering our rationale as described in the section 3.1 Design Rationale and Goals. Despite that, there was one more underlying goal: to effectively cooperate with developers and users from all over the world as well as contribute to an open-source project of that scale. We may now conclude that both objectives were met.

In more detail, regarding the main goal, the DSL extension, we achieved a breaking change: Kubeflow Pipelines can now run seamlessly with volume support for on-prem clusters, also enabling reproducible workflows. As a proof of concept for our implemented rationale, a number of organizations are using the feature we implemented, such as Cisco, IBM and Seldon, as well as private users. In addition, more and more people get to use it day by day. This was a result of the process, which included agreeing on a communicated a design document that the whole Kubeflow community embraced.

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.

5.2 Future Work

Although we have implemented a couple of enhancements to the User Interface (UI), upgrading the overall User Experience (UX), there is room for more development.

In addition, we can create additional classes, deriving from `ResourceOp`, to facilitate the creation of other Kubernetes resources that are useful to data scientists. For instance, a `TFJob` is widely used to run training jobs on Kubernetes. We could implement a `TFJobOp` following the same principle. That is making the common case fast, while allowing the definition of a complete resource specification.

Moreover, we should look for integration with Kubeflow Metadata service [33]. This, among others, keeps track of workflow executions and their inputs and outputs. These may be referencing input and output volume snapshots which could then be explored with just one click.

Finally, the logic behind our design paves the way for further features covering more Argo functionalities. For example, Argo has a template type to run a script and gather its results, without having to create an image containing this script and then requesting a container running it. That sounds really useful for machine learning scientists. We just need to verify this guess and design it with their help.

Bibliography & References

[1] Wikipedia – The Free Encyclopedia, *Machine Learning*, https://en.wikipedia.org/wiki/Machine_learning, accessed on the April 24, 2019.

[2] Kubeflow, https://www.kubeflow.org/, accessed on the April 24, 2019.

[3] Kubeflow Pipelines, https://github.com/kubeflow/pipelines, accessed on the April 23, 2019.

[4] Expert System Team, *What is Machine Learning? A definition*, https://www.expertsystem.com/machine-learning-definition/, accessed on the April 26, 2019.

[5] Wikipedia – The Free Encyclopedia, *List of self-driving car fatalities*, https://en.wikipedia.org/wiki/List_of_self-driving_car_fatalities, accessed on the April 26, 2019.

[6] Yael Gavish, *#2: What You Need to Know About Machine Learning Algorithms and Why You Should Care*, Jul 2017, https://medium.com/@yaelg/product-manager-guide-part-2-what-you-need-know-machine-learning-algorithms- accessed on the April 26, 2019.

[7] Wikipedia – The Free Encyclopedia, *Cloud Computing*, https://en.wikipedia.org/wiki/Cloud_computing, accessed on the May 2, 2019.

[8] Kubernetes https://kubernetes.io/, accessed on April 24, 2019.

[9] Cloud Native Computing Foundation, https://www.cncf.io/, accessed on the May 2, 2019.

[10] Microservices, https://microservices.io/, accessed on the May 2, 2019.

[11] Kubernetes Documentation, *Extending your Kubernetes Cluster*, https://kubernetes.io/docs/concepts/extend-kubernetes/extend-cluster/, accessed on the May 6, 2019.

[12] Pachyderm project, https://www.pachyderm.io/, accessed on the April 23, 2019.

[13] Git, https://git-scm.com/, accessed on the April 23, 2019.

[14] Wikipedia – The Free Encyclopedia, *Amazon S3 – S3 API and competing services*, https://en.wikipedia.org/wiki/Amazon_S3#S3_API_and_competing_services, accessed on the May 2, 2019.

[15] Gerben Oostra, *Pachyderm for data scientists*, November 2018, https://medium.com/bigdatarepublic/pachyderm-for-data-scientists-d1d1dff3a2fa, accessed on the April 23, 2019.

[16] Argo, https://github.com/argoproj/argo, accessed on the April 24, 2019.

[17] Wikipedia – The Free Encyclopedia, *Domain Specific Language*, https://en.wikipedia.org/wiki/Domain-specific_language, accessed on the April 24, 2019.

[18] Wikipedia – The Free Encyclopedia, *Application programming interface*, https://en.wikipedia.org/wiki/Application_programming_interface, accessed on the May 7, 2019.

[19] Kubeflow Organization, https://github.com/kubeflow, accessed on the May 13, 2019.

[20] Argoproj Organization, https://github.com/argoproj, accessed on the May 13, 2019.

[21] Wikipedia – The Free Encyclopedia, *Domain-specific language*, https://en.wikipedia.org/wiki/Domain-specific_language, accessed on the June 30, 2019.

[22] Google Cloud, *Containers at Google*, https://cloud.google.com/containers/, accessed on the June 30, 2019.

[23] Kubernetes Concepts, *What is Kubernetes*, https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/, accessed on the July 1, 2019.

[24] Anoop Vijayan, *Understanding the Container Storage Interface (CSI)*, Aug 2018, https://medium.com/google-cloud/understanding-the-container-storage-interface-csi-ddbeb966a3b, accessed on the July 1, 2019.

[25] Kubernetes Concepts, *Persistent Volumes*, https://kubernetes.io/docs/concepts/storage/persistent-volumes/, accessed on the July 1, 2019.

[26] Kubernetes Concepts, *Volume Snapshots*, https://kubernetes.io/docs/concepts/storage/volume-snapshots/, accessed on the July 1, 2019.

[27] Argo, *Workflows & Pipelines*, https://argoproj.github.io/argo/, accessed on the July 1, 2019.

[28] GitHub, https://github.com, accessed on the May 15, 2019.

[29] GitHub, *About Pull Requests*, https://help.github.com/en/articles/about-pull-requests, accessed on the May 15, 2019.

[30] Kubernetes Concepts, *Volumes*, https://kubernetes.io/docs/concepts/storage/volumes/, accessed on the June 5, 2019.

[31] GitHub Pull Request on kubeflow/pipelines Repository, *Feature: sidecar for ContainerOp (#879)*, https://github.com/kubeflow/pipelines/pull/879, accessed on the June 5, 2019.

[32] The Python Standard Library, unittest – *Unit testing framework*, https://docs.python.org/3/library/unittest.html, accessed on the June 10, 2019.

[33] Kubeflow Metadata, https://github.com/kubeflow/metadata, accessed on the July 4, 2019.