ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Επεκτάσεις στο Kubeflow για Συνεργατικές Ροές Εργασίας Μηχανικής Μάθησης Καθοδηγούμενες από τα Δεδομένα στον Κυβερνήτη

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Κίμωνας Κ. Σωτήρχος

Εργαστήριο Υπολογιστικών Συστημάτων
Αθήνα, Νοέμβριος 2019

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

Επεκτάσεις στο Kubeflow για Συνεργατικές Ροές Εργασίας Μηχανικής Μάθησης Καθοδηγούμενες από τα Δεδομένα στον Κυβερνήτη

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Κίμωνας Κ. Σωτήρχος

Επιβλέπων Καθηγητής: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 11ή Νοεμβρίου 2019.

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

Εργαστήριο Υπολογιστικών Συστημάτων
Αθήνα, Νοέμβριος 2019

National Technical University of Athens

School of Electrical and Computer Engineering

Division of Computer Science

Kubeflow Extensions for Collaborative Data-Driven Machine Learning Workflows on Kubernetes

DIPLOMA THESIS

Kimonas K. Sotirchos

Computing Systems Laboratory

Athens, November 2019

. . . . . .

Κίμωνας Κ. Σωτήρχος
Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

Abstract

The creation of a product that utilizes Machine Learning demands the use of advanced and collaborative workflows. In order to execute such workflows people with different skill sets and roles are needed to cooperate efficiently and effectively, as well as a platform that can support and provide the necessary infrastructure and tooling. Kubeflow is the de facto platform for running Machine Learning on Kubernetes that aims to seamlessly support and execute such collaborative and data-driven workflows on the cloud.

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 getting in contact with end users as well as its developers we found out that Kubeflow was lacking in the data management part of such workflows. Specifically the support for data that lived inside the local in-cluster storage was limited. Therefore, we re designed the workflows for running Jupyter Notebooks and Tensorboard visualizations in order to natively support the use of Kubernetes Volumes and also implemented a new workflow for managing these Volumes and their contents. The implementation of the Jupyter Notebooks became part of the upstream project and the other two will be incorporated in the next iterations.

All in all, the implemented workflows are widely used by organizations, such as IBM, Cisco and Google, as well as other users, and is getting more and more attention every day. The aim of this thesis was to not only solve a problem, but for the solution to be intuitive and acceptable from the end users.

Keywords

Kubernetes, Kubeflow, Machine Learning, Jupyter Notebooks, Tensorboard, PVCViewers, PVCs, Controllers, Angular, Python, clusters, data volumes, containers

Περίληψη

Η δημιουργία ενός προϊόντος που χρησιμοποιεί Μηχανική Μάθηση καθιστά αναγκαία την αποτελεσματική συνεργασία και συντονισμό μίας ομάδας ανθρώπων η οποία θα εκτελέσει πολύπλοκες και συνεργατικές ροές εργασίας. Προκειμένου να εκτελεστούν τέτοιες ροές εργασίας θα χρειαστεί μηχανικοί από διαφορετικές ειδικότητες να μπορούν να επικοινωνούν αποδοτικά, καθώς και μία πλατφόρμα πάνω στην οποία θα μπορέσουν να τις εκτελέσουν και θα τους παρέχει τα εργαλεία τους. Το Kubeflow είναι η πιο διαδεδομένη πλατφόρμα Μηχανικής Μάθησης στο cloud με σκοπό να επιτρέπει την εκτέλεση τέτοιων ροών εργασίας πάνω από τον Κυβερνήτη.

Κατά τη διάρκεια αυτής της διπλωματικής γίναμε ένα μέλος της κοινότητας του Kubeflow και ήρθαμε σε επαφή με Data Scientists μέσω του Slack του Kubeflow και συμμετείχαμε στα Community Meetings. Από την επικοινωνία που είχαμε τόσο με τους χρήστες του Kubeflow όσο και με τους μηχανικούς του συνειδητοποιήσαμε πως το Kubeflow υστερούσε στο κομμάτι της διαχείρισης των δεδομένων. Συγκεκριμένα, οι ροές εργασίας που περιλάμβαναν Jupyter Notebooks, αλλά και η χρήση εργαλείων οπτικοποίησης, όπως το Tensorboard, δεν είχαν εύκολους και απλούς μηχανισμούς ώστε να αξιοποιούν τον τοπικό αποθηκευτικό χώρο της συστοιχίας. Ως αφορμή αυτήν την προβληματική συμπεριφορά πρώτον ξανά σχεδιάσαμε τις ροές εργασίας που περιλαμβάνουν αυτά τα εργαλεία προκειμένου να χρησιμοποιούν και τον τοπικό αποθηκευτικό χώρο και δεύτερον υλοποιήσαμε μια καινούργια ροή εργασίας που τους επιτρέπει να διαχειρίζονται αυτά τα τοπικά δεδομένα. Οι υλοποίηση των Jupyter Notebooks είναι μέρος της τελευταίας έκδοσης του Kubeflow και οι άλλες δύο θα εισαχθούν σε αυτό στις μελλοντικές εκδόσεις.

Συνολικά, οι ροές εργασίας που υλοποιήσαμε χρησιμοποιούνται καθημερινά από μεγάλες εταιρίες και οργανισμούς όπως IBM, Cisco και Google, καθώς και από άλλους χρήστες, και λαμβάνουν όλο και περισσότερη προσοχή μέρα με τη μέρα. Ο σκοπός αυτής της διπλωματικής ήταν όχι μόνο να λύσουμε ένα υπάρχον πρόβλημα, αλλά ταυτόχρονα η λύση μας να είναι αποδεκτή και εύχρηστη από τους τελικούς χρήστες που θα την αξιοποιήσουν.

Λέξεις-Κλειδιά

Kubernetes, Kubeflow, Machine Learning, Jupyter Notebooks, Tensorboard, PVCViewers, PVCs, Controllers, Angular, Python, clusters, data volumes, containers

Πρόλογος

Πριν προχωρήσω, οφείλω στο σημείο αυτό να ευχαριστήσω, τον επιβλέποντα της διπλωματικής μου Καθηγητή Νεκτάριο Κοζύρη ο οποίος μου καλλιέργησε το ενδιαφέρον μου για τον τομέα τον Υπολογιστικών συστημάτων.

Εν συνεχεία, θα ήθελα να εκφράσω την βαθιά ευγνωμοσύνη μου και να ευχαριστήσω τον Διδάκτορα Βαγγέλη Κούκη τόσο για τον μεταδοτικό ενθουσιασμό του αλλά κυρίως για τον τρόπο σκέψης που μου καλλιέργησε. Θέλω να ευχαριστήσω επίσης τον Ιωάννη Ανδρουλιδάκη, για τον χρόνο που μου αφιέρωσε και την βοήθειά του σε τεχνικά ζητήματα της διπλωματικής μου τόσο άμεσα όσο και έμμεσα θέτοντας μου τα κατάλληλα ερωτήματα για να σκεφτώ ώστε να επανατοποθετήσω τις απόψεις μου.

Τέλος θέλω να ευχαριστήσω από τα βάθη της καρδιάς μου τους γονείς μου που με συνεχή αυτοθυσία και αγάπη με υποστήριζαν και νοιάζονταν για την εξέλιξή μου όλα αυτά τα χρόνια, καθώς και τα αδέρφια και τους φίλους μου οι οποίοι είναι δίπλα μου τόσο στις ευχάριστες όσο και στις δυσάρεστες στιγμές μου.

Τα άτομα αυτά αποτελούν ένα μέρος του εαυτού μου για το οποίο είμαι ευγνώμων και περήφανος.

Κίμωνας Σωτήρχος
Νοέμβριος 2019

Contents

Abstract iii

Keywords iii

Πρόλογος vii

List of figures xv

1 Εκτενής Περίληψη 1

1.1 Εισαγωγή 1

1.1.1 Διατύπωση Προβλήματος 1

1.1.2 Κίνητρα 2

1.1.2.1 Βήματα Ροής Εργασίας Μηχανικής Μάθησης 3

1.1.2.2 Συνεργατικές Ροές Εργασίας 3

1.1.2.3 Κατάσταση του Kubeflow 6

1.1.3 Υπάρχουσες Λύσεις 7

1.1.3.1 Kubeflow: Jupyter Notebooks 7

1.1.3.2 Kubeflow: Κατανεμημένη Εκπαίδευση 8

1.1.3.3 Kubeflow: Pipelines 9

1.1.3.4 Kubeflow: Βελτιστοποίηση Υπερ-Παραμέτρων . . . 9

1.1.4 Προτεινόμενη Λύση 10

1.1.5 Συνεργασίες 12

1.2 Τεχνικό Υπόβαθρο 13

1.2.1 Κυβερνήτης 13

1.2.1.1 Αρχιτεκτονική 13
1.2.1.2 Αντικείμενα & Controllers 14
1.2.2 Kubeflow 15
1.3 Σχεδίαση 17
1.3.1 Αρχές Σχεδιασμού και Στόχοι 17
1.3.2 Προτεινόμενες Ροές Εργασίας 19
1.3.2.1 Jupyter Notebooks 19
1.3.2.2 Tensorboard & Απεικόνιση Εξέλιξης 20
1.3.2.3 Διαδραστικό Περιβάλλον Χειρισμού Δίσκων 22
1.3.3 Υπολογιστικά Pods/CRDs Καθοδηγούμενα από τα Δεδομένα 23
1.3.3.1 Εξειδικευμένα CRDs για Εργασίες Μηχανικής Μάθησης 23
1.3.3.2 Υποστήριξη Δίσκων με PVCs 24
1.3.4 Εκκίνηση/Παύση των Αδρανή Pods και TTL 25
1.3.5 Web UIs για Χειρισμό Υπολογιστικών Pods 26
1.3.5.1 Εξουσιοδότηση και Δικαιώματα 28
1.3.5.2 Frontend για Έκθεση των CRDs 28
1.3.5.3 REST Backend APIs Χωρίς Κατάσταση 29
1.4 Υλοποίηση 30
1.4.1 Λογισμικό που Χρησιμοποιήσαμε 30
1.4.2 UI Διαχείρισης 31
1.4.2.1 Χτίσιμο και Τρέξιμο Κώδικα 31
1.4.2.2 Frontend 32
1.4.2.3 Backend 33
1.4.2.4 Έλεγχος Ταυτοποίησης και Εξουσιοδότησης 34
1.4.3 Χρόνος Ζωής των Αδρανή CRDs 35
1.4.4 Controllers για τα CRDs 36
1.4.4.1 Jupyter Notebooks 36
1.4.4.2 Οπτικοποίηση Μετρικών & Tensorboard 37
1.4.4.3 PVCViewers 38
1.5 Επίλογος 41
1.5.1 Τελικές Παρατηρήσεις 41
1.5.2 Μελλοντικές Επεκτάσεις 41

2 Introduction **43**
2.1 Problem Statement . . . 43
2.2 Motivation . . . 44
2.2.1 Machine Learning Components . . . 45
2.2.2 Collaborative Workflows . . . 46
2.2.3 The State of Kubeflow . . . 47
2.3 Existing Solutions . . . 48
2.3.1 Kubeflow: Jupyter Notebooks . . . 49
2.3.2 Kubeflow: Distributed Training . . . 50
2.3.3 Kubeflow: Pipelines . . . 50
2.3.4 Kubeflow: Hyper Parameter Tuning . . . 51
2.4 Proposed Solution . . . 52
2.5 Collaborations . . . 54
2.6 Thesis Structure . . . 54

3 Background **55**
3.1 OS-Level Virtualization & Containers . . . 55
3.1.1 Overview . . . 55
3.1.2 Building Blocks . . . 56
3.1.3 Docker . . . 57
3.1.3.1 Images . . . 57
3.1.3.2 Registries . . . 58
3.2 Kubernetes . . . 59
3.2.1 Overview . . . 59
3.2.2 Architecture . . . 59
3.2.3 Objects & Controllers . . . 61
3.2.3.1 Structure of Objects . . . 62
3.2.3.2 Controllers & Reconciliation Loop . . . 62
3.2.4 Storage . . . 62
3.2.4.1 Volumes . . . 62
3.2.4.2 PersistentVolumes & PersistentVolumeClaims . . . 63
3.3 Kubeflow . . . 63
3.3.1 Overview . . . 63
3.3.2 Architecture . . . 64
3.3.3 Access Kubeflow . . . 65

4 Design **67**
4.1 Design Rationale & Goals . . . 67
4.2 Proposed Workflow Implementations . . . 68
4.2.1 Jupyter Notebooks . . . 69
4.2.2 Tensorboard & Visualization . . . 70
4.2.3 Volume Data Web-Editors . . . 71
4.3 Data-Driven Computation Pods/CRDs . . . 72
4.3.1 Specialized CRDs for ML Tasks . . . 72
4.3.2 Support for PVCs . . . 73
4.4 Start/Stop of Idle Pods and TTL . . . 74
4.5 Web UIs for Managing Computation Pods . . . 75
4.5.1 Permissions and Authorization . . . 76
4.5.2 Frontend for Exposing the CRDs . . . 76
4.5.3 Stateless REST Backend APIs . . . 78

5 Implementation **81**
5.1 Software Stack . . . 81
5.1.1 Frontend Technologies . . . 81
5.1.2 Backend Technologies . . . 82
5.1.3 Controller Libraries and Frameworks . . . 82
5.2 Management UIs . . . 82
5.2.1 Overview . . . 82
5.2.2 Build & Deploy . . . 83
5.2.3 Frontend . . . 84
5.2.3.1 Main Page . . . 85
5.2.3.2 New Instance Page . . . 87
5.2.3.3 Logs Page . . . 87
5.2.4 Backend . . . 88
5.2.5 User Identity & Permissions . . . 89
5.2.6 Jupyter Notebooks Management UI . . . 90
5.2.7 Tensorboard Management UI . . . 92
5.2.8 PVCs Management UI . . . 94
5.3 TTL for Idle CRDs . . . 95
5.3.1 Culling Library . . . 96

5.3.2 Culling of Notebooks 97
5.3.3 Culling of Tensorboards, PVCVIewers 97
5.4 Custom Resource Controllers 97
5.4.1 Jupyter Notebooks 98
5.4.1.1 Notebook Spec 98
5.4.1.2 Notebooks Reconciliation Loop 99
5.4.2 Metric Viewers and Tensorboard 100
5.4.2.1 Viewer Spec 100
5.4.2.2 Viewers Reconciliation Loop 101
5.4.3 PVC Viewers 102
5.4.3.1 PVCViewers Spec 102
5.4.3.2 PVCViewers Reconciliation Loop 103

6 Conclusion **107**
6.1 Concluding Remarks 107
6.2 Future Work 108

Bibliography & References **109**

List of figures

3.1 Visualization of Docker Image Layers 58

3.2 Kubernetes Architecture Overview 60

5.1 Jupyter WebApp Dockerfile 84

5.2 Authorization-checking code 91

5.3 Jupyter web app main page 93

5.4 Notebook Object 105

1 Εκτενής Περίληψη

1.1 Εισαγωγή

1.1.1 Διατύπωση Προβλήματος

Η δημιουργία λογισμικού που χρησιμοποιεί Μηχανική Μάθηση απαιτεί τόσο εξειδικευμένα εργαλεία, καθώς και επαγγελματίες από διαφορετικούς κλάδους να μπορέσουν να συνεργαστούν αποτελεσματικά και ο καθένας να εκτελέσει το δικό του μέρος. Πρόκειται για μία άκρως συνεργατική διαδικασία της οποίας η αποτελεσματικότητα καθορίζεται από τον βαθμό στον οποίο επιτυγχάνεται αποτελεσματική επικοινωνία μεταξύ των μελών. Αυτό καθιστά την πλατφόρμα ανάπτυξης και διαχείρισης τέτοιων πολύπλοκων και συνεργατικών ροών εργασίας μείζονος σημασίας.

Μία τέτοια πλατφόρμα όμως θα έχει αρκετά κινούμενα κομμάτια τα οποία θα πρέπει να λειτουργούν πλήρως εναρμονισμένα μεταξύ τους. Χρειάζεται λογισμικό για την διαχείριση των δεδομένων, την ανάπτυξη του μοντέλου Μηχανικής Μάθησης, τον έλεγχο της απόδοσής του καθώς και για την εφαρμογή του μοντέλου ώστε να λαμβάνουμε προβλέψεις. Ο συντονισμός αυτών όμως γίνεται ακόμη πιο πολύπλοκος αν λάβουμε υπόψιν πως τα δεδομένα μεταβάλλονται με τον χρόνο, με αποτέλεσμα και να υπάρχει και συνεχής ροή δημιουργίας τέτοιων μοντέλων. Δυστυχώς, ο συντονισμός γίνεται με συνδετικό κώδικα ή με εξειδικευμένα προγράμματα με αποτέλεσμα να υπάρχει αλληλεπικάλυψη του κώδικα καθώς ο κάθε ένας εφαρμόζει την δική του προσέγγιση. Όμως, υπάρχουν λύσεις ανοιχτού κώδικα, όπως το Kubeflow πάνω στο οποίο αναπτύσσουμε και κώδικα, οι οποίες προσπαθούν να αντιμετωπίσουν το πρόβλημα με

καθολικό τρόπο και σχεδιασμό δημιουργώντας ταυτόχρονα πρότυπα σχεδίασης και αντιμετώπισης του προβλήματος.

Συγκεκριμένα, το Kubeflow είναι ένα περιβάλλον για ανάπτυξη και εκτέλεση ροών εργασίας Μηχανικής Μάθησης πάνω από τον Κυβερνήτη[4]. Όμως, στην τωρινή του έκδοση[1] η υπάρχουσα Εμπειρία Χρηστών (UX) για την εκτέλεση τέτοιων συνεργατικών ροών δεν είναι αρκετά εύχρηστη και βολική. Αυτό οφείλεται σε δύο παράγοντες. Πρώτον, δεν παρείχε εύκολους μηχανισμούς για την διαχείριση δεδομένων που ζουν σε τοπικούς δίσκους στην υπάρχουσα συστοιχία υπολογιστών. Δεύτερον, απαιτεί από τους χρήστες να έχουν γνώση από έννοιες όπως ο Κυβερνήτης ή Containers τις οποίες πολλοί Επιστήμονες Μηχανικής Μάθησης δεν έχουν, και μάλιστα δεν είναι απαραίτητο καθώς δεν είναι άμεσο μέρος της δουλειάς τους.

Οι προαναφερθέντες δυσκολίες όμως έχουν άμεση επίπτωση στην αποτελεσματικότητα των μηχανικών που δουλεύουν πάνω στην πλατφόρμα, το Kubeflow. Η έλλειψη ουσιαστικής διαχείρισης των δεδομένων από τοπικούς δίσκους κάνει τις συνεργατικές ροές εργασίες δύσκαμπτες καθώς η αποτελεσματικότητά τους καθορίζεται από την εύκολη διακίνηση των δεδομένων. Επιπλέον, η προαπαίτηση να υπάρχουν γνώσεις του Κυβερνήτη και των Containers συνεπάγεται πως οι Επιστήμονες Μηχανικής Μάθησης να αφιερώσουν χρόνο στο να τις αποκτήσουν ή στο να επικοινωνούν με την ομάδα DevOps αντί να τον αφιερώνουν στο να κάνουν την δουλειά τους. Αυτές οι χαμένες εργατοώρες όμως μεταφράζονται άμεσα σε κόστος που επενδύεται για το εκάστοτε τελικό προϊόν.

1.1.2 Κίνητρα

Οι λόγοι που επιλέξαμε να εντρυφήσουμε στο συγκεκριμένο πρόβλημα ήταν η πολυπλοκότητα των βημάτων των ροών εργασίας Μηχανικής Μάθησης, η συνεργατική τους φύση καθώς και ο βαθμός στον οποίο οι υπάρχουσες λύσεις μπορούν να διεκπεραιώσουν τέτοιες ροές. Διαπιστώσαμε πως η κατάσταση ήταν ελαττωματική και πως θα μπορούσαμε να την βελτιώσουμε.

Αρχικά θα αναλύσουμε ποια είναι τα επιμέρους βήματα σε μία πλήρη ροή εργασίας Μηχανικής Μάθησης, ποιου είδους μηχανικοί είναι υπεύθυνοι για ποια από αυτά τα

[1] έκδοση 0.4

βήματα καθώς και πως μπορούν να μεταλαμπαδεύσουν τα αποτελέσματά τους στους συνεργάτες τους.

1.1.2.1 Βήματα Ροής Εργασίας Μηχανικής Μάθησης

- **Ανάλυση των Δεδομένων**: Είναι η διαδικασία στην οποία γίνεται επεξεργασία, καθαρισμός και τροποποίηση των δεδομένων με σκοπό την εκπόνηση χρήσιμης πληροφορίας και μοτίβων.
- **Μετασχηματισμός των Δεδομένων**: Αφού τα δεδομένα έχουν αναλυθεί και έχουν εντοπιστεί οι σχέσεις μεταξύ τους που χρειάζονται, τα δεδομένα αυτά θα τροποποιηθούν ώστε να διατηρηθεί μόνο αυτή η αναγκαία πληροφορία.
- **Επικύρωση των Δεδομένων**: Θα γίνει έλεγχος των δεδομένων για πιθανά προβλήματα με τις τιμές. Σε αυτή την περίπτωση, προβληματικές τιμές θα αντικατασταθούν με τεχνητές.
- **Ανάπτυξη Μοντέλου**: Περιλαμβάνει την σχεδίαση του μαθηματικού μοντέλου Μηχανικής Μάθησης καθώς και την εκπαίδευσή του στα δεδομένα που δημιουργούνται από τα προηγούμενα βήματα. Τα δεδομένα θα ακολουθούν κάποια άγνωστη κατανομή στην οποία θα προσαρμοστεί το μοντέλο ώστε να παράγει ακριβή αποτελέσματα και προβλέψεις.
- **Επικύρωση και Έλεγχος του Μοντέλου**: Θα γίνει σύγκριση των μοντέλων που έχουν δημιουργηθεί προκειμένου να επιλεγεί αυτό με την καλύτερη απόδοση όταν εκτίθεται σε καινούργια δεδομένα.
- **Εξαγωγή του Μοντέλου για Εξυπηρέτηση**: Το μοντέλο που επιλέχθηκε θα ανέβει σε κάποιον server προκειμένου να μπορούν να συνδέονται άλλοι clients σε αυτό και να χρησιμοποιούν τις προβλέψεις του.

1.1.2.2 Συνεργατικές Ροές Εργασίας

Λέγοντας *συνεργατικές ροές* εννοούμε μία σειρά από βήματα και εργασίες των οποίων συγκεκριμένα τμήματα πρέπει να ολοκληρωθούν από διαφορετικά άτομα. Ο βαθμός στον οποίο τα εμπλεκόμενα άτομα καταφέρνουν να επικοινωνήσουν αποτελεσματικά

και να ανταλλάξουν τα αποτελέσματα της δουλειάς τους μεταξύ τους θα καθορίσει την ποιότητα του τελικού αποτελέσματος, καθώς και τους πόρους που θα αναλωθούν σε αυτό.

Σε αυτή τη διπλωματική θα εστιάσουμε σε αυτές τις *συνεργατικές ροές εργασίας* όπου χρησιμοποιούν οι Επιστήμονες Μηχανικής Μάθησης στην καθημερινότητά τους και να βελτιώσουμε αυτές που παρέχει το Kubeflow. Στο προηγούμενο κεφάλαιο είδαμε τα διαφορετικά βήματα που υπάρχουν σε αυτές τις ροές. Μια σημαντική λεπτομέρεια είναι πως από αυτά τα βήματα μόνο το ένα, η Ανάπτυξη του Μοντέλου, απαιτεί βαθιές γνώσεις Μηχανικής Μάθησης. Όλα τα υπόλοιπα βήματα απαιτούν μία ευρεία γκάμα από επαγγελματικές εξειδικεύσεις όπως Μηχανικούς Δεδομένων[2], DevOps, Μηχανικούς Λογισμικού[3]. Ο κάθε ένας από αυτούς θα πρέπει να εργάζεται στο δικό του μέρος της συνολικής ροής, το οποίο όμως θα πρέπει ως είσοδο να λαμβάνει την δουλειά και αποτελέσματα των υπόλοιπων καθώς και να μεταφέρει τα αποτελέσματά του στα επόμενα βήματα. Η δουλειά όμως του κάθε ενός μεταφράζεται σε δεδομένα, τα οποία θα πρέπει να μπορούν να μεταφέρονται εύκολα και αποτελεσματικά μεταξύ αυτών των βημάτων.

Για να μπορέσουμε όμως να κατανοήσουμε καλύτερα αυτές τις πολύπλοκες ροές εργασίας θα χρειαστεί να ξεκαθαριστεί η φύση και η δομή των αποτελεσμάτων των διάφορων βημάτων, καθώς και οι μηχανισμοί με τους οποίους τα διάφορα άτομα μπορούν να τα επικοινωνήσουν. Τα αποτελέσματα αυτά μεταφράζονται άμεσα σε *δεδομένα* τα οποία είναι αναγκαίο να μπορούν να μεταφερθούν αποδοτικά και αποτελεσματικά μεταξύ των ατόμων που εμπλέκονται.

Για τα πρώτα βήματα, *Ανάλυση, Μετασχηματισμό* και *Επικύρωση των Δεδομένων*, τα αποτελέσματά τους είναι τα *datasets* με τα οποία δουλεύουν οι Μηχανικοί Δεδομένων. Αφού ολοκληρωθούν τα απαραίτητα βήματα επεξεργασίας των δεδομένων, οι Επιστήμονες Μηχανικής Μάθησης θα είναι σε θέση να χρησιμοποιήσουν τα τελικά δεδομένα προκειμένου να εργαστούν στην δημιουργία και εκπαίδευση των μοντέλων με βάση αυτά. Πρόκειται για μία επαναλαμβανόμενη διαδικασία τόσο από την πλευρά των Μηχανικών Δεδομένων όσο και από τους Επιστήμονες Μηχανικής Μάθησης. Συγκεκριμένα, Οι Μηχανικοί Δεδομένων θα πρέπει να επεξεργάζονται διαρκώς και να παράγουν νέα datasets με την εισαγωγή καινούργιων δεδομένων και οι Επιστήμονες

[2]Data Scientists
[3]Software Engineers

Μηχανικής Μάθησης να επαναπροσδιορίζουν, να ξανά εκπαιδεύουν και να προσαρμόζουν τα μοντέλα τους με βάση αυτών. Αυτό σημαίνει πως οι Μηχανικοί Δεδομένων και οι Επιστήμονες Μηχανικής Μάθησης θα πρέπει να είναι διαρκώς σε επικοινωνία και να μπορούν συντονίζουν αποτελεσματικά τα αποτελέσματά τους μεταξύ τους.

Για το τελευταίο βήμα, την *Εξαγωγή του Μοντέλου για Εξυπηρέτηση*, θα χρησιμοποιηθεί το εκπαιδευμένο μοντέλο, δηλαδή τα δεδομένα εξόδου, από την προηγούμενη διαδικασία που περιγράψαμε. Τις περισσότερες φορές ενθυλακώνεται σε μία δικτυακή εφαρμογή[4] η οποία απαντάει σε http κίνηση η οποία περιέχει καινούργια δεδομένα και επιστρέφει το αποτέλεσμα του μοντέλου σε αυτά. Για να ολοκληρωθεί με επιτυχία αυτό το βήμα χρειάζεται το εκπαιδευμένο μοντέλο, καθώς και ένα υποσύνολο από τα δεδομένα σε εξειδικευμένες περιπτώσεις.

Υπάρχουν αρκετοί διαφορετικοί τρόποι και μηχανισμοί με τους οποίους τα datasets και τα μοντέλα που παράγονται μπορούν να μεταφερθούν αποτελεσματικά μεταξύ διαφορετικών ατόμων και ομάδων. Μία τεχνολογία που καλύπτει αυτή την ανάγκη είναι τα Object Stores[14]. Πρόκειται για μία βάση δεδομένων η οποία αποθηκεύει τα δεδομένα της ως ζεύγη κλειδιών-δεδομένων. Τα δεδομένα αυτά μπορούν επίσης να έχουν και διαφορετικές εκδόσεις για ένα κλειδί. Έτσι σε αυτά μπορούμε να αποθηκεύσουμε τα διάφορα μοντέλα, datasets καθώς και τις διαφορετικές εκδόσεις τους. Όμως, η χρήση μόνο αυτών μπορεί να επιφέρει κάποια σημαντικά προβλήματα. Συγκεκριμένα, αν τα δεδομένα υπάρχουν μόνο σε τέτοιες βάσεις τότε θα πρέπει να τα κατεβάζουμε κάθε φορά που θα πρέπει να χρησιμοποιηθούν κάτι που αυξάνει αρκετά τον φόρτο στο δίκτυο. Μία πιο αποτελεσματική λύση θα ήταν τα δεδομένα να μπορούν επίσης να αποθηκεύονται και να χρησιμοποιούνται από τον τοπικό χώρο της συστοιχίας όσο το δυνατόν περισσότερο. Θα μπορούσαμε να ελαττώσουμε τον φόρτο του δικτύου ακόμη περισσότερο αν αντί να αποθηκεύαμε τις διαφορετικές εκδόσεις των δεδομένων σαν ξεχωριστές οντότητες να μπορούσαμε να κρατάμε μόνο τις διαφορές που έχουν οι εκδόσεις μεταξύ τους, όπως με το git. Έτσι, οι μηχανικοί θα μπορούν να χρησιμοποιούν διαφορετικές εκδόσεις των δεδομένων κατεβάζοντας μόνο αυτές τις διαφορές από τα τοπικά τους δεδομένα.

Βλέπουμε λοιπόν πως η αποτελεσματικότητα με την οποία μεταφέρονται τα δεδομένα μεταξύ των ατόμων, είτε αυτά είναι datasets είτε μοντέλα, παίζει καθοριστικό ρόλο για

[4] web app

το τελικά προϊόν.

1.1.2.3 Κατάσταση του Kubeflow

Το Kubeflow είναι μία πλατφόρμα Μηχανικής Μάθησης που αποσκοπεί στο να παρέχει τα περισσότερα εργαλεία και ροές εργασίας που χρησιμοποιούν οι Επιστήμονες Μηχανικής Μάθησης πάνω από τον Κυβερνήτη[4].

Όταν ξεκινήσαμε να το κοιτάμε[5] ήταν ακόμη σε αρχικά στάδια εξέλιξης καθώς υπήρχε μόνο για ενάμιση χρόνο. Υποστήριζε αρκετά από τα εργαλεία που χρησιμοποιούσαν οι Επιστήμονες Μηχανικής Μάθησης, όπως τα Jupyter Notebooks, Tensorboard ακόμη και εξειδικευμένη γλώσσα για την εκτέλεση ροών εργασίας με προκαθορισμένα βήματα, πάνω από τον Κυβερνήτη. Όμως, του έλλειπαν σημαντικές υλοποιήσεις προκειμένου να είναι έτοιμο για την παραγωγή αλλά και, το πιο σημαντικό από την οπτική της διπλωματικής, δεν είχε αντιληφθεί πλήρως την συνεργατική φύση αυτών των ροών εργασίας, καθώς και το γεγονός ότι είναι καθοδηγούμενες από τα δεδομένα.

Πιο συγκεκριμένα, από άποψη ετοιμότητας για χρήση στην παραγωγή του έλλειπαν συγκεκριμένες κρίσιμες υλοποιήσεις όπως μηχανισμοί ταυτοποίησης και εξουσιοδότησης. Δεν υπήρχε κανένας τρόπος να υπάρχει κάποια υποτυπώδης διαχώριση μεταξύ των πόρων των διαφορετικών χρηστών. Δηλαδή, ο κάθε χρήστης είχε δικαίωμα σε όλα τα αντικείμενα του Kubeflow, ακόμη και διαφορετικών χρηστών.

Από άποψη Εμπειρίας Χρήστη (UX) το Kubeflow υποστήριζε τα περισσότερα από τα εργαλεία που χρησιμοποιούν οι Επιστήμονες Μηχανικής Μάθησης στις ροές εργασίας τους, όμως δεν είχε αποδοτικούς και εύκολους μηχανισμούς για την χρήση τοπικών δεδομένων της συστοιχίας. Για παράδειγμα δεν υπήρχε άμεσος τρόπος να χρησιμοποιηθούν τα δεδομένα της συστοιχίας μέσα από τα Jupyter Notebooks του Kubeflow. Έτσι, οι χρήστες θα έπρεπε διαρκώς να κατεβάζουν και να ανεβάζουν τα δεδομένα τους σε Object Stores, κάτι που αύξανε και την πολυπλοκότητα στον κώδικα αλλά και τους δικτυακούς πόρους που καταναλώνονταν.

Τέλος, το Kubeflow σε μεγάλο βαθμό δεν έκρυβε τις αρκετά τεχνικές λεπτομέρειες του Κυβερνήτη από τους χρήστες. Αυτό έχει ως συνέπεια να αυξάνεται ο πήχης των προαπαιτούμενων γνώσεων ώστε να μπορέσει κάποιος να το χρησιμοποιήσει αποδοτικά,

[5] Έκδοση 0.4

μειώνοντας έτσι την συνολική Εμπειρία Χρήστη.

1.1.3 Υπάρχουσες Λύσεις

Σε αυτή τη διπλωματική εστιάσαμε στο Kubeflow ως υπάρχουσα λύση, μιας και είναι μία πλατφόρμα που προσπαθεί να υποστηρίξει και να παρέχει τις ροές εργασίας που επισημάναμε. Επιπλέον, ήταν το πιο δημοφιλές πρότζεκτ σε σχέση με αντίστοιχες πλατφόρμες έχοντας παράλληλα μία αρκετά μεγάλη και φιλική κοινότητα. Στα επόμενα κεφάλαια θα εντρυφήσουμε στις ροές εργασίας που μας παρέχει το Kubeflow, καθώς και το βαθμό στον οποίο είναι αποδοτικές και εύχρηστες.

1.1.3.1 Kubeflow: Jupyter Notebooks

Μέχρι και την έκδοση 0.4 το Kubeflow χρησιμοποιούσε το JupyterHub για τον χειρισμό και την δημιουργία των Notebooks. Για να δημιουργήσει τα *Pods* που θα τρέξουν τα Jupyter Notebooks το JupyterHub δημιούργησε το πρότζεκτ Kubespawner[17]. Πρόκειται για ένα λογισμικό το οποίο είναι υπεύθυνο για την παραμετροποίηση και δημιουργία αυτών των *Pods* στον Κυβερνήτη, τα οποία συντονίζονται από το κεντρικό *Pod* του JupyterHub.

Επιπλέον, το JupyterHub μπορεί να ρυθμιστεί ώστε να συνεργάζεται και να εφαρμόζει τους μηχανισμούς ταυτοποίησης που υπάρχουν στην συστοιχία. Στην πιο απλή περίπτωση, όπου τέτοιοι μηχανισμοί δεν υπάρχουν στην συστοιχία, ο χρήστης θα δηλώνει μόνο ένα string που θα αντιπροσωπεύει το όνομά του. Το JupyterHub στη συνέχεια θα είναι υπεύθυνο στο να συντονίζει τα Pods των χρηστών, να βλέπει ποιος είναι ο συνδεδεμένος χρήστης κάθε φορά και να στέλνει την κίνησή του στα αντίστοιχα *Pods* για να τα χρησιμοποιήσει.

Με αυτό τον τρόπο όμως, το JupyterHub *Pod* θα είναι αυτό υπεύθυνο για να συντηρεί, να ανανεώνει και να διαχειρίζεται την κατάσταση των *Pods* και όχι οι Controllers του Κυβερνήτη. Η συμπεριφορά αυτή, όμως, εισήγαγε αρκετά προβλήματα στην χρήση των Notebooks.

Το βασικό πρόβλημα είναι και η απόφαση που αναφέραμε προηγουμένως, ότι το JupyterHub είναι υπεύθυνο για την συνολική εμπειρία. Αυτό σημαίνει πως, οποιαδήποτε

στιγμή, για να μπορέσει κάποιος είτε να δημιουργήσει ένα καινούργιο Notebook αλλά είτε και για να συνδεθεί σε ένα υπάρχον δικό του θα πρέπει να συνδεθεί πρώτα στο κεντρικό Pod του JupyterHub. Αυτό σημαίνει πως το κεντρικό αυτό *Pod* αποτελεί Μοναδικό Σημείο Αποτυχίας[6], καθώς αν για οποιονδήποτε λόγο δεν μπορεί να απαντήσει σε κίνηση τότε κανένας χρήστης δεν θα μπορέσει να χρησιμοποιήσει Notebooks. Επιπλέον, το JupyterHub αρκετές φορές δεν μπορούσε να διαβάσει σωστά την κατάσταση την κατάσταση των *Pods* που δημιουργούσε, με αποτέλεσμα μερικούς χρήστες να τους παρότρυνε να δημιουργήσουν καινούργιο Notebook αντί να τους συνδέσει στο ήδη υπάρχον.

1.1.3.2 Kubeflow: Κατανεμημένη Εκπαίδευση

Το Kubeflow παρείχε έναν μηχανισμό ώστε η εκπαίδευση ενός μοντέλου να μπορεί να γίνεται κατανεμημένα στην συστοιχία. Με αυτό τον τρόπο, οι Επιστήμονες Μηχανικής Μάθησης μπορούν να δημιουργήσουν το μοντέλο τους στο τοπικό τους περιβάλλον ανάπτυξης και να τρέξουν το βήμα της εκπαίδευσης του μοντέλου κατανεμημένα στην συστοιχία, αξιοποιώντας του πόρους της.

Προκειμένου να εκμεταλλευτεί την κατανεμημένη φύση του Κυβερνήτη, προκειμένου να τρέξει τους βαρείς υπολογισμούς, το Kubeflow ενσωματώνει τον κατανεμημένο τρόπο τρεξίματος του TensorFlow[43] στον Κυβερνήτη ως έννοια πρώτης τάξης. Δηλαδή, δημιουργεί ένα *CustomResourceDefinition*, το *TFJob*, το οποίο ενθυλακώνει όλες τις τεχνικές λεπτομέρειες και την διαχείριση ώστε να μπορούν εργασίες TensorFlow να τρέξουν στον Κυβερνήτη.

Ένα τέτοιο αντικείμενο περιλαμβάνει πληροφορία σχετικά με τους *Parameter Servers, Worker* και *Chief* ρόλους του TensorFlow. Δηλαδή, ο χρήστης θα ρυθμίζει πως θέλει να είναι ο γράφος της κατανεμημένης εκπαίδευσης και μετά ο Κυβερνήτης θα ρυθμίσει πως να τον υλοποιήσει ο χρήστης θα ρυθμίζει πως θέλει να είναι ο γράφος της κατανεμημένης εκπαίδευσης και μετά ο Κυβερνήτης θα ρυθμίσει πως να τον υλοποιήσει. Ο χρήστης θα είναι σε θέση να ρυθμίζει τον κώδικα που θα τρέχει κάθε ένας από τους προηγούμενους ρόλους ορίζοντας τα Docker Images που επιθυμεί.

Ο *Controller* των *TFJob* θα είναι υπεύθυνος για την υλοποίηση του κατανεμημένου υπολογιστικού γράφου. Θα πρέπει επίσης να δημιουργήσει και να διαχειριστεί τα *Pods*

[6]Single Point of Failure

που θα υλοποιήσουν τελικά τον γράφο. Η πληροφορία σχετικά με το ποια *Pods* θα συμμετέχουν στον γράφο θα πρέπει να είναι διαθέσιμη σε όλα τα *Pods* καθώς αυτά μπορεί να χρειαστεί να επικοινωνήσουν μεταξύ τους και να ανταλλάξουν ενδιάμεσα αποτελέσματα.

Η συγκεκριμένη υλοποίηση είναι βολική, όμως δεν υπάρχει κάποιο γραφικό περιβάλλον για την δημιουργία *TFJob* αντικειμένων. Αυτό συνεπάγεται πως ο χρήστης θα πρέπει και να έχει πρόσβαση στον API Server του Κυβερνήτη, αλλά και να είναι έμπειρος με το πως να δημιουργεί και να χειρίζεται αντικείμενα σε αυτόν.

1.1.3.3 Kubeflow: Pipelines

Ένα pipeline είναι μια περιγραφή με προκαθορισμένα βήματα μιας ροής εργασίας Μηχανικής Μάθησης σε μορφή ενός γράφου. Το pipeline ορίζει για κάθε βήμα τις εισόδους και τις εξόδους του και κατά συνέπεια από ποια άλλα βήματα μπορεί να εξαρτάται. Έπειτα αυτός ο ορισμός από βήματα θα εκτελεστεί στην συστοιχία με Κυβερνήτη.

Από την οπτική του Κυβερνήτη, ένα pipeline είναι ένα αρχείο *yaml* όπου περιέχει μια ροή εργασίας ορισμένη με Argo[34]. Προκειμένου να τρέξει λοιπόν ένα pipeline θα πρέπει να δημιουργηθεί στην βάση του Κυβερνήτη ένα τέτοιο Argo αντικείμενο που θα περιγράφει μια ροή εργασίας.

Όμως, επειδή το να φτιάχνει ο χρήστης τέτοια αντικείμενα στον Κυβερνήτη είναι και χρονοβόρο αλλά και απαιτεί αρκετές γνώσεις από Κυβερνήτη, οι μηχανικοί στο Kubeflow δημιούργησαν ένα SDK όπου οι Επιστήμονες Μηχανικής Μάθησης μπορούν να χρησιμοποιήσουν για να δημιουργούν τέτοια αντικείμενα πιο άμεσα απευθείας μέσα από τον κώδικά τους. Μπορούν μέσα από ένα Jupyter Notebook να ορίσουν τα βήματα από το pipeline ρυθμίζοντας μόνο τα απαραίτητα, γλυτώνοντας αρκετές από τις έννοιες του Κυβερνήτη, και με την χρήση αυτού του SDK να δημιουργήσουν το τελικό *yaml*. Τέλος θα μπορούν μέσω του UI των Pipelines να το εκτελέσουν και να δουν τα αποτελέσματά του.

1.1.3.4 Kubeflow: Βελτιστοποίηση Υπερ-Παραμέτρων

Το Kubeflow παρέχει επίσης μια λύση για βελτιστοποίηση ύπερ παραμέτρων των μοντέλων μηχανικής μάθησης μέσω του *Katib*[44] το οποίο είναι εμπνευσμένο από το

Google Vizier[45]. Η κύρια ιδέα πίσω από το *Katib* είναι να ορίσει ο χρήστης σε έναν container όλη την διαδικασία εκπαίδευσης σαν ένα αυτόνομο βήμα. Ο container θα πρέπει να λαμβάνει σαν δεδομένα εισόδου το πλήθος και τις τιμές των υπερπαραμέτρων που θέλουμε να βελτιστοποιήσουμε, θα τρέχει την εκπαίδευση, και στο τέλος θα επιστρέφει μία μετρική απόδοσης.

Από την στιγμή που οι containers θα έχουν μία τέτοια προκαθορισμένη δομή, θα είναι αρκετά άμεσο και εύκολο να τους τρέξουμε αρκετές φορές, ακόμη και παράλληλα. Αυτό σε έναν υπολογιστή μπορεί να μην βγάζει αρκετό νόημα, σε μία συστοιχία υπολογιστών όμως θα μπορούμε να τρέχουμε πολλές δοκιμές υπερπαραμέτρων ταυτόχρονα.

Όπως και με τα *TFJobs* έτσι κι εδώ το Kubeflow ορίζει ένα καινούργιο *CustomResourceDefinition* όπου χειρίζεται όλες τις τεχνικές λεπτομέρειες και τον συντονισμό του Κυβερνήτη. Ο χρήστης θα επιλέγει τον container που θα εκπαιδεύει το μοντέλο, τον αλγόριθμο για την εξερεύνηση των βέλτιστων παραμέτρων, καθώς και το αποδεκτό κατώφλι σφάλματος και το Kubeflow θα αναλαμβάνει να τρέχει *Pods* μέχρι να επιτευχθεί η απαιτούμενη απόδοση.

1.1.4 Προτεινόμενη Λύση

Ο στόχος αυτής της διπλωματικής είναι να μπορούν οι Μηχανικοί Δεδομένων και οι Επιστήμονες Μηχανικής Μάθησης να βρίσκονται σε οικείο περιβάλλον ανάπτυξης με μόλις μερικά κλικ, χωρίς να χρειάζεται να έχουν εντρυφήσει στις έννοιες του Κυβερνήτη. Ταυτόχρονα, θέλουμε να μπορούν να αξιοποιήσουν τις δυνατότητες που τους παρέχει το cloud μέσα από το περιβάλλον ανάπτυξής τους. Τέλος, σκοπεύουμε να βελτιώσουμε τις υπάρχουσες ροές εργασίας του Kubeflow προκειμένου να αξιοποιούν και τον τοπικό αποθηκευτικό χώρο της συστοιχίας σε συνδυασμό με γραφικά περιβάλλοντα όπου θα τους επιτρέπουν να διαχειρίζονται τα δεδομένα που θα βρίσκονται εκεί. Με αυτό τον τρόπο το Kubeflow θα παρέχει εύχρηστες συνεργατικές ροές εργασίας που θα είναι πραγματικά *καθοδηγούμενες από τα δεδομένα*.

Στη συνέχεια θα περιγράψουμε μία end-to-end ροή Μηχανικής Μάθησης όπου θα γίνεται *Ανάλυση, Μετασχηματισμός* και *Αξιολόγηση των Δεδομένων* καθώς και κατανεμημένη *Εκπαίδευση Μοντέλου* αλλά και *Αξιολόγηση* μέσα από ένα Jupyter Notebook,

το οποίο είναι το πιο διαδεδομένο εργαλεία μεταξύ των Μηχανικών Δεδομένων και των Επιστημόνων Μηχανικής Μάθησης. Επιπλέον, όλα τα παραπάνω βήματα θα εκτελούνται στην συστοιχία υπολογιστών, στο cloud. Έτσι οι ροές αυτές θα μπορούν να είναι πιο αποδοτικές, καθώς θα αξιοποιούν στο μέγιστο τους πόρους που παρέχονται. Τέλος η ροή αυτή θεωρεί πως η υποδομή και η πλατφόρμα για Μηχανική Μάθηση θα είναι το Kubeflow και ο Κυβερνήτης.

Η ροή αυτή ξεκινάει με την αρχική τοποθεσία των datasets. Επίσης θεωρούμε πως υπάρχει λογισμικό το οποίο περιλαμβάνει ένα Object Store στο οποίο θα αποθηκεύεται κεντρικά όλη η πληροφορία, το οποίο όμως θα μπορεί να αποθηκεύει αποδοτικά τις διαφορές που υπάρχουν μεταξύ των διαφορετικών εκδόσεων των αντικειμένων. Επιπλέον, το λογισμικό αυτό θα μπορεί να συγχρονίζει αποδοτικά τα τοπικά δεδομένα με βάση τα δεδομένα που θα βρίσκονται σε αυτό το Object Store, καθώς θα χειρίζονται μόνο διαφορές στα αρχεία και όχι ολόκληρα τα δεδομένα κάθε αυτά.

Ροή Εργασίας

- **Σύνδεση στην Πλατφόρμα Μηχανικής Μάθησης**: Ο χρήστης θα συνδέεται στην πλατφόρμα με κάποια στοιχεία ταυτοποίησης που του αντιστοιχούν από τον browser. Έπειτα Θα οδηγηθεί στην κεντρική σελίδα, από την οποία θα συνδεθεί στο γραφικό περιβάλλον για την διαχείριση των Notebooks ή Δίσκων.

- **Εισαγωγή των Δεδομένων**: Υπάρχουν δύο τρόπου με τους οποίους μπορεί να γίνει αυτό το βήμα. Είτε να κατεβάσει τα δεδομένα εξ αρχής από το κεντρικό Object Store, είτε θα χρησιμοποιήσει δεδομένα που βρίσκονται στον τοπικό αποθηκευτικό χώρο της συστοιχίας. Και στις δύο περιπτώσεις τα δεδομένα θα καταλήξουν σε ένα *PVC* από όπου και θα μπορέσουν να καταναλωθούν και από το υπολογιστικό *Pod* στο οποίο θα τρέξει το Notebook.

- **Δημιουργία ενός Notebook**: Από το γραφικό περιβάλλον για την διαχείριση των Notebooks ο χρήστης θα ξεκινήσει ένα Notebooks στο οποίο θα του δώσει το *PVC* το οποίο περιέχει τα δεδομένα που κατέβασε από το προηγούμενο βήμα. Η πλατφόρμα θα έχει τους κατάλληλους μηχανισμούς ταυτοποίησης και εξουσιοδότησης ώστε να μην επιτρέπει σε χρήστες να χρησιμοποιούν Notebooks άλλων χρηστών. Έτσι ο χρήστης μόλις με λίγα κλικ θα βρίσκεται στο οικείο περιβάλλον του για ανάπτυξη.

- **Πειραματισμός μέσα από το Notebook**: Μέσα από το Notebook οι Μηχανικοί Δεδομένων και Επιστήμονες Μηχανικής Μάθησης είτε θα χρειαστεί να εκτελέσουν την διαδικασία του βήματος *Εισαγωγή των Δεδομένων* ώστε να φέρουν τα δεδομένα τους από το Object Store είτε θα έχουν πρόσβαση σε αυτά μέσω των συνδεδεμένων *PVCs*. Έτσι θα είναι σε θέση να ξεκινήσουν να δουλεύουν αξιοποιώντας αυτά τα δεδομένα.

- **Εκτέλεση Βημάτων αρκετές φορές**: Καθώς θα κάνουν δοκιμές και θα τρέχουν κώδικα θα έχουν την δυνατότητα να παίρνουν στιγμιότυπα, snapshots, τόσο από τα δεδομένα όσο και από το περιβάλλον εργασίας τους, τα οποία θα ζουν σε *PVCs*.

- **Κατανεμημένη Εκπαίδευση μέσα από το Notebook**: Αφού δημιουργηθεί η δομή και αρχιτεκτονική του μοντέλου Μηχανικής Μάθησης, οι Επιστήμονες Μηχανικής Μάθησης θα μπορούν να το εκπαιδεύουν στο cloud και να βλέπουν άμεσα την πρόοδο και τα αποτελέσματά του μέσα από το Notebook.

1.1.5 Συνεργασίες

Κατά τη διάρκεια της διπλωματικής εργασίας συνεργαστήκαμε διεξοδικά με τους προγραμματιστές και συνδεθήκαμε με την κοινότητα του Kubeflow και αναγνωρίστηκα ως ένα **μέλος** αυτής της κοινότητας.

1.2 Τεχνικό Υπόβαθρο

1.2.1 Κυβερνήτης

1.2.1.1 Αρχιτεκτονική

Ο Κυβερνήτης[4] είναι ένα πρότζεκτ Ανοιχτού Πηγαίου Κώδικα που έχει σκοπό να αυτοματοποιήσει το ξεκίνημα, την κλιμακωσιμότητα και τον χειρισμό από εφαρμογές που τρέχουν σε containers. Δημιουργήθηκε από την Google, με την πρώτη 1.0 έκδοση που βγήκε το 2015. Αναλαμβάνει και ρυθμίζει τη δικτύωση, τους υπολογιστικούς πόρους και την συνολική υποδομή προκειμένου να μπορούν να τρέχουν οι εφαρμογές σε containers σε μία συστοιχία από υπολογιστές.

Η βασική φιλοσοφία του Κυβερνήτη είναι να αφήνει τον χρήστη να καθορίζει πως θέλει να είναι η *επιθυμητή κατάσταση* και έπειτα ο ίδιος θα αναλάβει να παρακολουθεί και να χειρίζεται την υποδομή προκειμένου το σύστημα να καταλήξει σε αυτή την κατάσταση.

Η κατάσταση ορίζεται ως ένα σύνολο JSON αντικειμένων τα οποία είναι αποθηκευμένα σε μία κατανεμημένη βάση δεδομένων, υψηλής αντοχής σε σφάλματα και με μεγάλη διαθεσιμότητα, τον *etcd*[27].Έπειτα, έχουμε τα *master components*, κύριες συνιστώσες, οι οποίες παρακολουθούν και καταγράφουν συνεχώς την πραγματική κατάσταση της συστοιχίας και προσπαθούν να την φέρουν στην *επιθυμητή κατάσταση*. Οι κύριες συνιστώσες έχουν πρόσβαση στην κατάσταση της συστοιχίας μέσω ενός συγκεκριμένου-τομέα REST API το οποίο χειρίζεται την κατανεμημένη βάση δεδομένων που είναι αποθηκευμένη η κατάσταση της συστοιχίας και του Κυβερνήτη.

Ουσιαστικά οι κύριες συνιστώσες αποτελούν το control plane. Οι κυρίαρχες συνιστώσες παίρνουν τις αποφάσεις για την συστοιχία και αντιδρούν σε αλλαγές. Μπορούν να τρέξουν σε οποιονδήποτε από τους κόμβους, το σύνηθες όμως είναι να τρέχουν στον όλα κύριο κόμβο στον οποίο μάλιστα δεν τρέχει κανένας container.

Κύριες Συνιστώσες

1. **etcd**: Μία κατανεμημένη βάση δεδομένων που χρησιμοποιείται για να αποθηκεύει την κατάσταση του Κυβερνήτη και της συστοιχίας. Ο Κυβερνήτης ακολουθεί την φιλοσοφία του Αρχηγού-Ακόλουθου η οποία μπορεί να εισάγει κε-

ντρικό σημείο αποτυχίας[7], το οποίο μπορεί να αντιμετωπιστεί αυξάνοντας το πλήθος των κύριων κόμβων.

2. **kube-apiserver**: Εκθέτει το API του Κυβερνήτη. Οποιοσδήποτε θέλει να έχει πρόσβαση στην κατάσταση του Κυβερνήτη πρέπει να επικοινωνήσει με αυτή τη συνιστώσα. Μπορεί να γίνει πιο ανθεκτική σε σφάλματα αν την τρέχουμε σε περισσότερους από έναν κόμβους.

3. **kube-scheduler**: Παρακολουθεί την κατάσταση του Κυβερνήτη αντικείμενα *Pod* τα οποία δεν έχουν ανατεθεί σε κάποιον κόμβο για να εκτελεστούν και τα αναθέτει σε κάποιον. Οι παράγοντες που λαμβάνονται υπόψιν σε αυτή τη διαδικασία αποτελούν οι πόροι που απαιτεί να καταναλώσει, περιορισμοί υλικού ή λογισμικού, προτιμήσεις σε πόρους, τοπολογία των δεδομένων καθώς και η κατάσταση των ίδιων των κόμβων.

4. **kube-controller-manager**: Είναι η συνιστώσα η οποία συντονίζει και τρέχει τους Controllers του Κυβερνήτη για τα διάφορα αντικείμενα που υπάρχουν. Σχεδιαστικά, κάθε Controller είναι μία ξεχωριστή διεργασία αλλά για λόγους πολυπλοκότητας τρέχουν όλοι μαζί σαν μία καθολική διεργασία.

1.2.1.2 Αντικείμενα & Controllers

Ο Κυβερνήτης παρέχει ένα σύνολο από αφαιρέσεις οι οποίες χρησιμοποιούνται για να αναπαραστήσουν την κατάσταση του συστήματος. Αυτή αποτελείται από τις εφαρμογές σε containers που τρέχουν, τους πόρους που δεσμεύουν και καταναλώνουν καθώς και πληροφορία σχετικά με την κατάσταση της συστοιχίας. Όλη αυτή η πληροφορία αναπαριστάται από τις αφαιρέσεις που εισάγει ο Κυβερνήτης, τα αντικείμενα στο API του Κυβερνήτη.

Ένα αντικείμενο είναι μία δήλωση που περιγράφει μία επιθυμητή κατάσταση. Αφού δημιουργηθεί, οι μηχανισμοί του Κυβερνήτη θα ανανεώνουν συνέχεια στο *status* του αντικειμένου και θα προσπαθούν διαρκώς να φέρουν την πραγματική του κατάσταση στην επιθυμητή κατάσταση. Τα αντικείμενα αυτά έχουν μία προκαθορισμένη δομή:

- **Kind**: Περιγράφει το είδος του αντικειμένου, για παράδειγμα *Pod, Deployment*.

[7] Single Point of Failure

- **apiVersion**: Καθορίζει σε ποια έκδοση του ορισμού του αντικειμένου αφορά ο συγκεκριμένος ορισμός.
- **Metadata**: Δεδομένα που χρησιμοποιούνται για την ταυτοποίηση του αντικειμένου καθώς και έξτρα πληροφορία που δεν επηρεάζει άμεσα τον ορισμό του αντικειμένου.
- **Spec**: Ο ορισμός της επιθυμητής κατάστασης του αντικειμένου. Ο Κυβερνήτης θα προσπαθεί διαρκώς να εξασφαλίζει πως η πραγματική κατάσταση της συστοιχίας είναι συμβατή με αυτό τον ορισμό.
- **Status**: Η πραγματική κατάσταση για το συγκεκριμένο αντικείμενο. Μόνο ο Κυβερνήτης πειράζει αυτό το πεδίο για κάθε αντικείμενο.

Τα αντικείμενα όμως κάθε αυτά είναι μόνο JSON εγγραφές σε μία βάση δεδομένων. Το λογισμικό που είναι υπεύθυνο για να αντιδρά σε αυτά τα αντικείμενα είναι οι **Controllers**. Πρόκειται για προγράμματα που τρέχουν διαρκώς και παρατηρούν το API του Κυβερνήτη για τα αντικείμενα που είναι υπεύθυνοι. Αυτοί είναι υπεύθυνοι να τροποποιούν την κατάσταση της συστοιχίας για να ικανοποιούνται οι ορισμοί των αντικειμένων αυτών.

1.2.2 Kubeflow

Το Kubeflow είναι ένα πρότζεκτ Ανοικτού Πηγαίου Κώδικα όπου ήταν εμπνευσμένο από τον τρόπο που η Google έτρεχε το TensorFlow[28] εσωτερικά, βασισμένο στο πρότυπο του TensorFlow Extended[29]. Ξεκίνησε αρχικά ως μία εναλλακτική για να τρέχουμε δουλειές και εργασίες TensorFlow πιο εύκολα στον Κυβερνήτη. Από τότε όμως έχει εξελιχθεί σε μία ολόκληρη πλατφόρμα, υποστηρίζοντας υποδομές από διάφορους παρόχους, ικανή να τρέχει πολύπλοκες ροές εργασίας. Ο κύριος στόχος του πρότζεκτ είναι να παρέχει τα κατάλληλα εργαλεία ώστε να μπορέσουν να εκτελεστούν εργασίες Μηχανικής Μάθησης στον Κυβερνήτη.

Για να το επιτύχει αυτό, το Kubeflow αποτελείται από ένα σύνολο από *Components*, πακέτα λογισμικού ουσιαστικά, τα οποία επιλύουν συγκεκριμένα προβλήματα το κάθε ένα. Ο όρος *Component* προέκυψε από το ksonnet[35] το οποίο χρησιμοποιούνταν στην αρχή για να εφαρμόζει αυτά τα πακέτα σε μία συστοιχία με Κυβερνήτη. Το κάθε

πακέτο περιέχει, συνήθως, έναν ορισμό *CustomResourceDefinition* που θα αναπαριστά μία έννοια ή εργαλείο Μηχανικής Μάθησης, τον αντίστοιχο Controller που τον υλοποιεί, γραφικό περιβάλλον για τον χειρισμό αυτών των αντικειμένων καθώς και τα απαραίτητα δικαιώματα στα προηγούμενα.

Μέχρι και την έκδοση 0.4 το Kubeflow παρείχε αρκετά από τα εργαλεία και ροές για Μηχανική Μάθηση. Συγκεκριμένα προσέφερε Jupyter Notebooks για πειραματισμό, κατανεμημένη εκπαίδευση στην συστοιχία, βελτιστοποίηση ύπερ-παραμέτρων καθώς και μία γλώσσα ώστε οι χρήστες να μπορούν να καθορίζουν εξατομικευμένες ροές εργασίας με συγκεκριμένα, και επαναλαμβανόμενα, βήματα. Κάθε ένα από αυτά είναι και ένα αυτόνομο πακέτο που θα εφαρμοστεί στην συστοιχία όταν εγκαθίσταται το Kubeflow.

Όλα αυτά τα διαφορετικά κομμάτια λογισμικού συνδυάζονται ως μία οντότητα και παρέχουν μια συνεχή εμπειρία μέσω της κεντρικής σελίδας του Kubeflow. Αυτή η σελίδα απεικονίζει περιληπτικά τους διάφορους πόρους και αντικείμενα που υπάρχουν στο Kubeflow, καθώς και τους παρέχει τη δυνατότητα να χρησιμοποιήσουν τις υπάρχουσες λύσεις που προσφέρει.

Από την έκδοση 0.6 και μετά το Kubeflow απέκτησε την έννοια του προσωποποιημένου χρήστη. Για να μπορέσει κάποιος να χρησιμοποιήσει τις παροχές που προσφέρει το Kubeflow, εγκατεστημένο σε μία συστοιχία, θα πρέπει πρώτα να συνδεθεί με προσωπικά στοιχεία σε ένα κεντρικό σημείο ταυτοποίησης της πλατφόρμας. Η διαδικασία σύνδεσης και ταυτοποίησης εξαρτάται άμεσα από την φύση της υποδομής και από τον πάροχο που την προσφέρει. Μόνο αφού συνδεθεί θα μπορέσει να χρησιμοποιήσει το Kubeflow. Ταυτόχρονα, αφού υπάρχει ταυτοποίηση θα υπάρχει και εξουσιοδότηση ώστε να μπορεί να οριστεί πλήρως το σύνολο των δικαιωμάτων που έχουν οι χρήστες. Με αυτό τον τρόπο το Kubeflow υποστηρίζει πολλαπλούς χρήστες.

1.3 Σχεδίαση

1.3.1 Αρχές Σχεδιασμού και Στόχοι

Όπως αναφέραμε και στο κεφάλαιο 1.1.1 Διατύπωση Προβλήματος, στόχος αυτής της διπλωματικής είναι να βελτιώσουμε την αποτελεσματικότητα και την Εμπειρία των Χρηστών που παρέχουν οι συνεργατικές ροές εργασίας που υποστηρίζει το Kubeflow. Τα προβληματικά σημεία αφορούσαν την διαχείριση Jupyter Notebooks, τα εργαλεία που παρέχονταν ώστε οι χρήστες να επεξεργάζονται τα δεδομένα στη συστοιχία υπολογιστών, καθώς και τη διαχείριση εργαλείων για οπτικοποίηση των αποτελεσμάτων των μοντέλων Μηχανικής Μάθησης.

Πιο συγκεκριμένα, τα προβλήματα στις υποστηριζόμενες συνεργατικές ροές εργασίας είναι:

- **Notebooks**: Η διαχείριση των Notebooks ήταν προβληματική καθώς το Jupyter Hub[31], το οποίο ήταν υπεύθυνο, ήταν αρκετά ελαττωματικό. Αυτό οφείλονταν στο γεγονός ότι το JupyterHub προσπαθούσε να αναλάβει εξολοκλήρου την διαχείριση των Notebooks χωρίς να όμως να αξιοποιήσει τους υπάρχοντες μηχανισμούς του Κυβερνήτη, όπως τα *CustomResourceDefinitions* [32]. Επιπλέον δεν υπήρχε τρόπος οι χρήστες να μπορέσουν να χρησιμοποιήσουν υπάρχοντα αποθηκευτικό χώρο της συστοιχίας από τα Notebooks, μέσω *PersistentVolumeClaims*.

- **Διαχείριση Δεδομένων**: Η υποστήριξη για δίσκους, μέσω των *PVCs* ήταν σε πολύ αρχικά στάδια. Στις περισσότερες περιπτώσεις τα δεδομένα θα αποθηκεύονταν σε Object Stores[14] και οι χρήστες θα έπρεπε στη συνέχεια να κατεβάζουν και να ανεβάζουν τα δεδομένα τους από εκεί. Αυτό συνεπάγεται πως θα παράγεται και μετακινείται σημαντικός όγκος δεδομένων στο δίκτυο, καθώς οι συνεργατικές ροές δεδομένων απαιτούν να μπορούν να χειρίζονται τα δεδομένα από προηγούμενα βήματα. Εκτός αυτού, όμως, το Kubeflow δεν προσφέρει κάποιον αποτελεσματικό τρόπο ώστε να μπορούν οι χρήστες να χειρίζονται και να βλέπουν τα δεδομένα τους τα οποία ζουν σε δίσκους στη συστοιχία, PVCs.

- **Εργαλεία Οπτικοποίησης Αποτελεσμάτων**: Το Kubeflow δίνει στους χρήστες τη δυνατότητα να χρησιμοποιήσουν το Tensorboard προκειμένου να οπτικο-

ποιήσουν την εξέλιξη των μοντέλων τους και να παρακολουθούν τις μετρικές που παράγει καθώς αυτό εκπαιδεύεται. Όμως, για άλλη μια φορά, δεν υπάρχει υποστηρίξει για να χρησιμοποιηθούν δεδομένα που ζουν στους τοπικούς δίσκους της συστοιχίας. Έτσι θα πρέπει και το μοντέλο να γράφει τα δεδομένα του συνέχεια σε κάποιο δικτυακό χώρο αποθήκευσής αλλά και το Tensorboard να κοιτάει συνέχεια εκεί για αλλαγές, το οποίο συνεπάγεται επιβάρυνση του δικτύου λόγω της συνεχής ανταλλαγής δεδομένων.

Ο στόχος μας είναι να αντιμετωπίσουμε τα προβλήματα αυτά με ομοιόμορφο τρόπο. Προτείνουμε να χρησιμοποιήσουμε τα *CustomResourceDefinitions* του Κυβερνήτη για τα εργαλεία Μηχανικής Μάθησης που υποστηρίζει το Kubeflow, σε συνδυασμό με Web UIs τα οποία θα διαχειρίζονται αποτελεσματικά τα στιγμιότυπα αυτών των εργαλείων διευκολύνοντας παράλληλα της χρήση *PersistentVolumeClaims*.

Συγκεκριμένα προτείνουμε να:

1. Δημιουργήσουμε *CRDs* για αντικείμενα *Notebooks*, *Tensorboard* καθώς και για *PVC Viewer*. Οι υπεύθυνοι Controllers που θα παρακολουθούν τα αντικείμενα αυτά θα δημιουργούν τους απαραίτητους πόρους και αντικείμενα στον Κυβερνήτη προκειμένου να σηκωθούν τα εργαλεία των χρηστών και να μπορούν εύκολα και γρήγορα να τα χρησιμοποιούν. Με αυτό τον τρόπο αναθέτουμε την ευθύνη της διαχείρισης στον Κυβερνήτη και όχι σε τρίτο λογισμικό, όπως το JupyterHub για παράδειγμα.

2. Δημιουργήσουμε λιτά Web UIs τα οποία θα χειρίζονται τις αλληλεπιδράσεις των χρηστών με τα στιγμιότυπα των εργαλείων τους στην συστοιχία. Τα UIs θα μπορούν να εφαρμόζουν CRUD[8] ενέργειες πάνω στα *CRDs* της συστοιχίας. Έτσι λοιπόν, σε συνδυασμό με τους Controllers αυτών των *CRDs* όλες οι τεχνικές και επίπονες λεπτομέρειες προκειμένου να σηκωθούν τα εργαλεία στην συστοιχία αποκρύπτονται από τους τελικούς χρήστες.

3. Να ενσωματώσουμε τα *PVCs* με τον ορισμό των προαναφερθέντων *CRDs* ώστε να μπορούν να χρησιμοποιούνται άμεσα από αυτά. Έτσι, τα *PVCs*, θα μπορέσουν να ενταχθούν στις ροές εργασίας των χρηστών και να χρησιμοποιηθούν

[8]Create, Read, Update, Delete

προκειμένου να μεταφέρονται τα αποτελέσματα της δουλειάς των χρηστών εύκολα και γρήγορα μεταξύ των διαφόρων βημάτων.

1.3.2 Προτεινόμενες Ροές Εργασίας

Σε αυτό το κεφάλαιο θα προτείνουμε και θα αναλύσουμε τις ροές εργασίας που προτείνουμε να έχει το Kubeflow. Οι ροές αυτές θα πρέπει να είναι εύχρηστες αλλά και ανταπεξέρχονται στον συνεργατικό στοιχείο που τις χαρακτηρίζει. Στα επόμενα κεφάλαια θα αναπτύξουμε περισσότερο τις τεχνικές λεπτομέρειες προκειμένου να μπορέσουν να υλοποιηθούν τέτοιες ροές στον Κυβερνήτη και να γίνουν μέρος του οικοσυστήματος του Kubeflow.

1.3.2.1 Jupyter Notebooks

Μιας και τα Jupyter Notebooks είναι από τα πιο δημοφιλή και διαδεδομένα εργαλεία για Μηχανική Μάθηση, θέλουμε να είναι όσο το δυνατόν πιο εύκολο, άμεσο και κατανοητό να τα χειριστούν οι χρήστες μέσω του Kubeflow.

Πρώτον, όλη η ενορχήστρωση και διαχείριση των Notebooks καθώς και οι αλληλεπιδράσεις με αυτά, ως αντικείμενα του Κυβερνήτη στην συστοιχία, θα πραγματοποιούνται μέσω του UI Διαχείρισης για Notebooks. Αφού ο χρήστης συνδεθεί στο Kubeflow, το UI θα του εμφανίσει όλα τα Notebooks που είναι διαθέσιμα στο *Namespace* του συγκεκριμένου χρήστη[9]. Ταυτόχρονα θα υπάρχουν μηχανισμοί ελέγχου και προστασίας και το UI θα επιτρέπει στους χρήστες να χειρίζονται μόνο τα Notebooks για τα οποία έχουν δικαιώματα να τροποποιούν.

Αφού ο χρήστης επιλέξει το *Namespace* στο οποίο θέλει να εργαστεί, το UI θα του εμφανίσει έναν πίνακα στον οποίο θα απεικονίζεται, συνοπτικά, πληροφορία σχετικά με τα Notebooks που υπάρχουν σε αυτό το *Namespace*. Για κάθε Notebook ο χρήστης θα μπορεί να βλέπει πόσους πόρους καταναλώνει, τα *PVCs* που χρησιμοποιεί, το Docker image αλλά και διάφορες προκαθορισμένες ρυθμίσεις που μπορεί να έχουν γίνει. Επιπλέον ο χρήστης θα μπορεί να εκτελεί ορισμένες ενέργειες πάνω στο κάθε Notebook όπως να συνδεθεί σε αυτό, να το διαγράψει, να το σταματήσει ή να το ξανά ξεκινήσει ή και να δει τα logs του.

[9] Από την έκδοση 0.6.2 και μετά, ο κάθε χρήστης θα έχει τουλάχιστον το δικό του Namespace

Το UI θα έχει επίσης ένα κουμπί το οποίο θα προτρέπει τον χρήστη σε μία φόρμα όπου θα μπορεί να ξεκινήσει ένα καινούργιο Notebook ρυθμίζοντας συγκεκριμένα χαρακτηριστικά του. Συγκεκριμένα θα μπορεί να ρυθμίζει το όνομα του Notebook, τους πόρους και τα *PVCs* που θα μπορεί να καταναλώνει καθώς και να εφαρμόσει έξτρα προκαθορισμένες ρυθμίσεις που θα έχει ορίσει ο διαχειριστής. Ο διαχειριστής της συστοιχίας, που θα χειρίζεται και το Kubeflow, θα έχει τη δυνατότητα να ορίζει προκαθορισμένες τιμές στα διάφορα πεδία της φόρμας για τους χρήστες ή ακόμη να τους απαγορέψει ολοκληρωτικά να αλλάξουν κάποια από τα πεδία. Η πληροφορία αυτής της φόρμας θα αποτυπωθεί σε ένα *CusomResource*, από το backend, το οποίο θα δημιουργηθεί στο API του Κυβερνήτη και θα είναι υπεύθυνο για την κατάσταση του συγκεκριμένου Notebook στην συστοιχία.

Ο χρήστης θα μπορεί να συνδεθεί στα Notebooks μέσω αυτού του UI. Αφού πατήσει να συνδεθεί σε ένα υπάρχον Notebook θα παραπεμφθεί σε ένα καινούργιο tab από το οποίο θα έχει πρόσβαση σε αυτό. Από εκεί θα μπορέσει να κατεβάσει τα δεδομένα που χρειάζεται ή θα έχει πρόσβαση σε αυτά μέσω των συνδεδεμένων *PVCs*. Έχοντας όλα τα δεδομένα που χρειάζονται θα είναι σε θέση να ξεκινήσουν να κάνουν τους πειραματισμούς και τη δουλειά τους μέσα από το Notebook. Επίσης, σε συγκεκριμένες πλατφόρμες[10] θα είναι σε θέση να παίρνουν και snapshot από το περιβάλλον ανάπτυξής τους και από τα δεδομένα τους.

Τέλος θα του δίνεται η δυνατότητα να τρέχει συγκεκριμένα τμήματα του κώδικά του απευθείας στην συστοιχία με περισσότερους πόρους. Αφού ολοκληρωθεί το τρέξιμο θα μπορεί να δει τα αποτελέσματα άμεσα πάλι μέσα από το Notebook.

Ο στόχος της συγκεκριμένης ροής εργασίας είναι οι Επιστήμονες Μηχανικής Μάθησης να είναι σε θέση να εκτελέσουν όλα τα βήματα που χρειάζονται παραμένοντας συνέχεια στο Notebook, με το οποίο και είναι εξοικειωμένοι. Με την ροή αυτή οι χρήστες θα μπορούν να αξιοποιούν στο μέγιστο τους πόρους που θα τους παρέχει η συστοιχία χωρίς όμως να χρειάζεται να εντρυφήσουν στις τεχνικές λεπτομέρειες.

1.3.2.2 Tensorboard & Απεικόνιση Εξέλιξης

Μία κρίσιμη δυνατότητα που θέλουν να έχουν οι περισσότεροι Επιστήμονες Μηχανικής Μάθησης είναι να μπορούν να οπτικοποιούν και να παρακολουθούν την εξέ-

[10]Ένα παράδειγμα τέτοιας συστοιχίας είναι οι συστοιχίες με Kubeflow που παρέχει η Arrikto

λιξη των μοντέλων τους ζωντανά καθώς αυτά εκπαιδεύονται. Η μεθοδολογία αυτή είναι αρκετά χρήσιμη καθώς τους βοηθάει να αποκτήσουν καλύτερη διαίσθηση για τη συμπεριφορά του μοντέλου τους ώστε να μπορέσουν να το βελτιώσουν περαιτέρω. Το Tensorboard είναι ένα Python Module το οποίο παρέχει αυτές τις δυνατότητες για μοντέλα που αναπτύσσονται με το TensorFlow. Το TensorFlow, όμως, είναι το πιο δημοφιλή framework για ανάπτυξη μοντέλων Μηχανικής Μάθησης και είναι προ εγκατεστημένο στα Docker Images του Kubeflow. Αυτό καθιστά το Tensorboard απαραίτητο εργαλείο που θα πρέπει να μπορούν να χρησιμοποιούν εύκολα και γρήγορα οι Επιστήμονες Μηχανικής Μάθησης μέσω του Kubeflow.

Πρώτον, όπως και με τα Jupyter Notebooks, όλη η ενορχήστρωση και διαχείριση των Tensorboard στιγμιοτύπων καθώς και οι αλληλεπιδράσεις με αυτά, ως αντικείμενα του Κυβερνήτη στην συστοιχία, θα πραγματοποιούνται μέσω του UI Διαχείρισης για Tensorboard. Αφού ο χρήστης συνδεθεί στο Kubeflow, το UI θα του εμφανίσει όλα τα Tensorboard Pods που είναι διαθέσιμα στο *Namespace* του συγκεκριμένου χρήστη. Ταυτόχρονα θα υπάρχουν μηχανισμοί ελέγχου και προστασίας και το UI θα επιτρέπει στους χρήστες να χειρίζονται μόνο τα Tensorboard για τα οποία έχουν δικαιώματα να τροποποιούν.

Αφού ο χρήστης επιλέξει το *Namespace* στο οποίο θέλει να εργαστεί, το UI θα του εμφανίσει έναν πίνακα στον οποίο θα απεικονίζεται, συνοπτικά, πληροφορία σχετικά με τα Tensorboard που υπάρχουν σε αυτό το *Namespace*. Για κάθε Tensorboard ο χρήστης θα μπορεί να βλέπει την πηγή των δεδομένων τα οποία θα οπτικοποιούνται στον χρήστη. Για κάθε Tensorboard ο χρήστης θα μπορεί να συνδεθεί για να το χρησιμοποιήσει, να το διαγράψει ή να δει τα logs του.

Το UI θα έχει επίσης ένα κουμπί το οποίο θα προτρέπει τον χρήστη σε μία φόρμα όπου θα μπορεί να ξεκινήσει ένα καινούργιο Tensorboard ρυθμίζοντας το όνομά του καθώς και την πηγή των δεδομένων. Είναι πολύ σημαντικό ως πηγή δεδομένων να μπορούν να είναι δεδομένα που ζουν είτε σε Object Stores αλλά είτε στους τοπικούς δίσκους της συστοιχίας, σε PVCs του συγκεκριμένου *Namespace*. Σε συνδυασμό με το UI Διαχείρισης των Notebooks, ο χρήστης θα μπορεί να ξεκινάει ένα Notebook μέσα στο οποίο θα εκπαιδεύει το μοντέλο του. Το μοντέλο καθώς εκπαιδεύεται θα γράφει τα δεδομένα του σε ένα *PVC* που θα είναι διαθέσιμο στο Notebook. Έπειτα ο χρήστης μέσω του UI Διαχείρισης Tensorboard θα μπορεί να ξεκινήσει ένα Tensorboard χρη-

σιμοποιώντας ως πηγή δεδομένων το προηγούμενο *PVC* πάνω στο οποίο το μοντέλο θα γράφει τις μετρικές του. Έτσι θα μπορεί να βλέπει ζωντανά την εξέλιξή του καθώς αυτό θα εκπαιδεύεται μέσα από το Notebook. Να αναφέρουμε ότι το Tensorboard δεν το ενδιαφέρει πως εκπαιδεύεται το μοντέλο, αυτό κοιτάει μόνο τα δεδομένα που παράγονται όσο αυτό εκπαιδεύεται.

1.3.2.3 Διαδραστικό Περιβάλλον Χειρισμού Δίσκων

Όπως αναφέραμε και στα προηγούμενα κεφάλαια. ένας από τους βασικούς στόχους αυτής της διπλωματικής είναι να εντάξουμε τη χρήση των *PVCs* στις ροές εργασίας που υποστηρίζει το Kubeflow. Ο συγκεκριμένος στόχος εκπληρώνεται εν μέρη με τις δύο προηγούμενες προτεινόμενες ροές εργασίας, οι οποίες εντάσσουν πλήρως την χρήση των *PVCs*. Για να μπορέσουν όμως να ενταχθούν πραγματικά στις ροές του Kubeflow είναι απαραίτητο ο χρήστης να είναι σε θέση να μπορεί να βλέπει και να επεξεργάζεται εύκολα και απλά τα αρχεία που βρίσκονται στα *PVCs* αλλά και να μπορεί να να ρυθμίζει τον κύκλο ζωής τους.

Για να το επιτύχουμε αυτό, όπως και στις δύο προηγούμενες ροές εργασίας, θα εισάγουμε αρχικά ένα UI Διαχείρισης Δίσκων. Αφού ο χρήστης συνδεθεί στο Kubeflow, το UI θα του εμφανίσει όλα τα *PVCs* που είναι διαθέσιμα στο *Namespace* του συγκεκριμένου χρήστη. Ταυτόχρονα θα υπάρχουν μηχανισμοί ελέγχου και προστασίας και το UI θα επιτρέπει στους χρήστες να χειρίζονται μόνο τα *PVCs* για τα οποία έχουν δικαιώματα να τροποποιούν.

Αφού ο χρήστης επιλέξει το *Namespace* που επιθυμεί το UI θα του εμφανίσει έναν πίνακα στον οποίο θα απεικονίζεται, συνοπτικά, πληροφορία σχετικά με τα *PVCs* που υπάρχουν σε αυτό το *Namespace*. Για κάθε *PVC* ο χρήστης θα μπορεί να βλέπει το μέγεθός του, το mode καθώς και το *StorageClass* του. Το UI θα παρέχει, επίσης, ένα κουμπί με το οποίο θα παροτρύνει τον χρήστη σε μία φόρμα για να δημιουργήσει καινούργιο *PVC*. Για κάθε υπάρχον *PVC* ο χρήστης θα έχει τη δυνατότητα να το διαγράψει ή να δει και να πειράξει τα αρχεία που περιέχει. Αν επιλέξει να εξερευνήσει τα περιεχόμενά του, τότε το UI θα τον κατευθύνει σε ένα καινούργιο tab στο οποίο θα υπάρχει ένα άλλο web app το οποίο θα του επιτρέπει να εξερευνήσει και να πειράξει τα αρχεία του επιλεγμένου *PVC*.

Με τη συγκεκριμένη ροή εργασίας μειώνουμε αισθητά την υπάρχουσα τριβή που υπάρ-

χει για να χρησιμοποιηθούν τα *PVC* από τους Επιστήμονες Μηχανικής Μάθησης. Τώρα θα είναι σε θέση με μόλις ελάχιστα κλικ του ποντικιού να χειρίζονται τα δεδομένα τους που είναι αποθηκευμένα στον τοπικό αποθηκευτικό χώρο της συστοιχίας, μέσα σε *PVCs*.

1.3.3 Υπολογιστικά Pods/CRDs Καθοδηγούμενα από τα Δεδομένα

Λέγοντας *Καθοδηγούμενα από τα Δεδομένα* εννοούμε πως τα Pods, τα οποία θα εκτελέσουν των κώδικα για τα εργαλεία που θέλουν να χρησιμοποιήσουν οι χρήστες, θα πρέπει να είναι σε θέση να χειριστούν δεδομένα που θα βρίσκονται στον τοπικό αποθηκευτικό χώρο της συστοιχίας, μέσα σε *PVCs*. Αυτό, όμως, θα επηρεάσει άμεσα το scheduling των Pods καθώς πια τα δεδομένα θα έχουν βαρύτητα.

Ιδανικά θα θέλαμε τα δεδομένα να μπορούσαν να έχουν διαφορετικές εκδόσεις και στιγμιότυπα, τα οποία θα αποθηκεύονται σε ένα Object Store. Έτσι, θα μπορούμε να μεταφέρουμε τα δεδομένα σε οποιοδήποτε σημείο του τοπικού αποθηκευτικού χώρου.

1.3.3.1 Εξειδικευμένα CRDs για Εργασίες Μηχανικής Μάθησης

Σε αυτή τη διπλωματική προτείνουμε να δημιουργήσουμε *CustomResourceDefinitions* για κάθε ένα από τα *Notebook*, *Tensorboard* και *PVCViewers*. Αυτά τα *CRDs* θα ενθυλακώσει όλες τις τεχνικές λεπτομέρειες προκειμένου να τρέξουν τα εργαλεία στην συστοιχία παρέχοντας ταυτόχρονα στον χρήστη τη δυνατότητα να παραμετροποιήσει τα χαρακτηριστικά που χρειάζεται. Σε αυτή την ενότητα θα αναλύσουμε ποια δευτερεύοντα αντικείμενα θα πρέπει να δημιουργηθούν προκειμένου να μπορέσουν να εκτελεστούν τα εργαλεία των χρηστών και να τους είναι διαθέσιμα.

Τα *CRDs* αυτά θα έχουν αρκετά παρόμοια δομή. Η πιο βασική έννοια του Κυβερνήτη για να εκτελέσουμε κώδικα είναι ένα *Pod*. Γι᾽ αυτό και όλοι οι αντίστοιχοι Controllers τελικά θα πρέπει να φτιάχνουν κάποιο Pod, έμμεσα ή άμεσα, όπου θα εκτελεστεί το εργαλείο του χρήστη. Όμως, για να εκμεταλλευτούμε τους μηχανισμούς και τον έξυπνο σχεδιασμό του Κυβερνήτη, δεν θα θεωρούμε υπεύθυνους τους ίδιους τους Controllers για να ελέγχουν την κατάσταση των *Pods*, αλλά θα χρησιμοποιήσουμε άλλα βασικά αντικείμενα όπως *Deployments* και *StatefulSets*.

Έπειτα, θα χρειαστεί να ρυθμίσουμε την ροή της δικτυακής κίνησης προκειμένου να μπορούν οι χρήστες να συνδεθούν στα *Pods* που θα περιέχουν τα εργαλεία. Το πρώτο βήμα θα είναι να ρυθμίσουμε την εσωτερική κίνηση στην συστοιχία για κάθε ένα από αυτά τα *Pods*. Αυτό μπορεί να επιτευχθεί χρησιμοποιώντας άλλη μία βασική δομή του Κυβερνήτη, τα *Services*. Στη συνέχεια θα χρειαστεί να ρυθμίσουμε την κίνηση που θα έρχεται έξω από την συστοιχία να μπορεί να φτάνει στα σωστά *Pods* μέσα στην συστοιχία. Υπάρχουν δύο τρόποι να το πετύχουμε αυτό και εξαρτώνται από την έκδοση του Kubeflow που είναι εγκατεστημένη. Μέχρι και την έκδοση 0.5 χρησιμοποιούνταν ο Ambassador[36] για να γίνεται συστοιχία της κίνησης από το εξωτερικό προς το εσωτερικό της συστοιχίας των υπολογιστών. Σε αυτή την περίπτωση θα αρκεί μόνο να βάλουμε συγκεκριμένα *annotations* στο *Service* του κάθε *Pod*. Από την έκδοση 0.6 και ύστερα για την συστοιχία της κίνησης από το εξωτερικό προς το εσωτερικό της συστοιχίας υπολογιστών χρησιμοποιείται το ISTIO[38]. Σε αυτή την περίπτωση θα χρειαστεί ο Controller να φτιάχνει και ένα *VirtualService* για να επιτρέπει στην εξωτερική κίνηση να πηγαίνει στα εσωτερικά *Services*.

Αυτά τα αντικείμενα, *Deployments* ή *StatefulSets*, *Services* και ίσως *VirtualServices*, θα πρέπει να τα δημιουργούν οι Controllers καθενός από τα *CRDs*. Η κυριότερες διαφορές μεταξύ των *CRDs* θα είναι μικρές διαφοροποιήσεις στους ορισμούς των *Pods* και στους κανόνες που θα δέχονται και θα προωθούν κίνηση τα *Services*.

1.3.3.2 Υποστήριξη Δίσκων με PVCs

Η δημιουργία *CustomResourceDefinitions* μεταφέρουν την ευθύνη για την ενορχήστρωση και την διαχείριση των στιγμιότυπων των εργαλείων στον Κυβερνήτη. Τα στιγμιότυπα αυτά όμως δεν θα είναι εφήμερες εφαρμογές, αλλά θα έχουν κάποια μορφή κατάστασης. Θα πρέπει λοιπόν να μπορούμε να χρησιμοποιούμε *PVCs* με αυτά τα *CRDs* ώστε να αποθηκεύουμε την κατάσταση. Αφού τελικά δημιουργούμε *Pods* το να χρησιμοποιήσουμε *PVCs* είναι μια αρκετά απλή διαδικασία, καθώς χρειάζεται μόνο να επισημάνουμε στον ορισμό του εκάστοτε *Pod* τα *PVCs* που θέλουμε να χρησιμοποιήσει.

Όμως, δεν είναι αποδοτικό να περιμένουμε από τον χρήστη να δημιουργεί *CustomResources* στην συστοιχία προκειμένου να δημιουργεί στιγμιότυπα από τα εργαλεία του. Παρόλο που αυτά τα *CRDs* αποκρύπτουν αρκετές από τις τεχνικές λεπτομέρειες

εξακολουθούν να είναι αρκετά επίπονα για να γράφονται με το χέρι. Γι᾽ αυτό το λόγω είναι κρίσιμο τα UIs που θα δημιουργήσουμε να δείχνουν στον χρήστη μόνο τα απολύτως απαραίτητα που χρειάζεται, αποκρύπτοντας ακόμη περισσότερο τις έννοιες του Κυβερνήτη, ώστε να χειριστούν και να δημιουργήσουν τα εργαλεία τους. Είναι εξίσου σημαντικό, επίσης, τα UIs να διευκολύνουν αρκετά την επιλογή *PVCs*, δίσκων δηλαδή από την οπτική των χρηστών, για τα εργαλεία τους.

1.3.4 Εκκίνηση/Παύση των Αδρανή Pods και TTL

Σε αυτό το σημείο έχουμε καταφέρει να συγκεντρώσουμε όλο τον κόπο για να ξεκινήσουμε ένα στιγμιότυπο εργαλείου στην δημιουργία ενός *yaml* αρχείου που περιέχει ένα *CustomResource*. Με βάση αυτό μπορούμε εύκολα να εφαρμόσουμε CRUD[11] διεργασίες πάνω στα στιγμιότυπά μας. Όμως, ακόμη κι έτσι η εμπειρία που προσφέρουμε στους χρήστες είναι σχετικά μονοκόμματη. Ένα στιγμιότυπο εργαλείου μπορεί είτε να υπάρχει είτε όχι. Πολλές φορές όμως είναι προτιμότερο να έχουμε δημιουργήσει τις προδιαγραφές από ένα εργαλείο, από πλευράς πόρων που καταναλώνει για παράδειγμα, και να μπορούμε γρήγορα να το κατεβάζουμε και να το ξανασηκώνουμε από το να το διαγράφουμε τελείως και να το ξανά φτιάχνουμε από την αρχή.

Υπάρχουν δύο βασικοί λόγοι γιατί είναι χρήσιμη μια τέτοια συμπεριφορά. Πρώτον, ώστε να εξοικονομούμε πόρους και κατά δεύτερον χρόνο στο να δημιουργούμε εργαλεία ξανά και ξανά με τους ίδιους ορισμούς. Η χρησιμότητα της συγκεκριμένης συμπεριφοράς είναι πιο εμφανής στα Jupyter Notebooks, ειδικά για εκείνα που χρησιμοποιούν πολύτιμους και ακριβούς υπολογιστικούς πόρους όπως GPUs. Τέτοια Notebooks μπορούν να παράξουν μεγάλο κόστος σε περίπτωση που τρέχουν για αρκετή ώρα. Οπότε για να περιοριστεί το κόστος αυτό θα πρέπει οι Επιστήμονες Μηχανικής Μάθησης να διαγράφουν επιτόπου αυτά τα Notebooks αφού ολοκληρώσουν τις δοκιμές τους. Όμως, αυτή η ροή εργασίας είναι αρκετά ευάλωτη στον ανθρώπινο παράγοντα καθώς είναι αρκετά εύκολο κάποιος να ξεχάσει ένα Notebooks ανοιχτό. Επίσης ακόμη και αν οι χρήστες ήταν πάντα συνεπής και διέγραφαν τα Notebooks, θα ήταν και πάλι επίπονο να το ξανά φτιάχνουν σε περίπτωση που θέλουν να έχουν το ίδιο περιβάλλον ανάπτυξης.

[11] Create, Read, Update, Delete

Προκειμένου, λοιπόν, να αντιμετωπίσουμε αυτό το πρόβλημα προτείνουμε τις εξής αλλαγές:

1. **Εκκίνηση/Παύση Στιγμιότυπων**: Ο χρήστης θα μπορεί να θέτει τον αριθμό των *Pods* για ένα εργαλείο είτε στο μηδέν στο ένα. Με αυτό τον τρόπο ο ορισμός του στιγμιότυπου θα παραμένει αποθηκευμένος στον Κυβερνήτη αλλά δεν θα καταναλώνονται πόροι. Ο χρήστης θα έχει τον έλεγχο για το πότε θέλει ένα στιγμιότυπο να ξεκινάει να καταναλώνει πόρους, να ξεκινάει να υφίσταται δηλαδή, και πότε όχι. Καθοριστικό ρόλο σε αυτή τη ροή θα παίξει για άλλη μια φορά το UI. Θα πρέπει να μπορεί εύκολα με το πάτημα ενός κουμπιού να εναλλάσσει την κατάσταση των στιγμιότυπων.

2. **Σταμάτημα των Αδρανή Pods**: Προκειμένου να αντιμετωπίσουμε τον ανθρώπινο παράγοντα θα ήταν χρήσιμο η πλατφόρμα να μπορεί να σταματάει, με τον μηχανισμό παραπάνω, τα αδρανή *Pods*. Ο όρος *αδρανή* θα εξαρτάται από το εκάστοτε εργαλείο, αλλά θα υπάρχει και η γενική περίπτωση στην οποία ως μετρική θα χρησιμοποιείται ο ρυθμός δικτυακής κίνησης. Επίσης οι χρήστες θα έχουν τη δυνατότητα να ρυθμίζουν πόσο σύντομα επιθυμούν να θεωρούνται αδρανή συγκεκριμένα στιγμιότυπα. Για παράδειγμα, τα Notebooks που χρησιμοποιούν GPUs θέλουμε να θεωρούνται αδρανή πιο γρήγορα σε σχέση με αυτά που χρησιμοποιούν μόνο CPU.

Η κατάσταση που θα δείχνουν οι πίνακες για τα στιγμιότυπα του κάθε *CustomResource* θα πρέπει να λαμβάνουν υπόψιν και να απεικονίζουν την κατάσταση κατάλληλα όταν τα Pods είναι σταματημένα με τον παραπάνω μηχανισμό. Υπεύθυνοι για να αποφασίζουν αν ένα στιγμιότυπο είναι αδρανή και να το σταματάνε θα είναι οι Controllers.

1.3.5 Web UIs για Χειρισμό Υπολογιστικών Pods

Δημιουργώντας τα προαναφερθέντα *CRDs* καταφέρνουμε να συγκεντρώσουμε όλα τα βήματα και τις λεπτομέρειες για το τρέξιμο των εργαλείων στην συστοιχία σε απλές CRUD ενέργειες πάνω σε αντικείμενα στον Κυβερνήτη. Όμως, δεν είναι αποδοτικό να

περιμένουμε από τον χρήστη να δημιουργεί αυτά τα αντικείμενα ο ίδιος στην συστοιχία προκειμένου να δημιουργεί στιγμιότυπα από τα εργαλεία του. Πρώτον, διότι αυτό σημαίνει πως ο χρήστης θα πρέπει να έχει πρόσβαση το API του Κυβερνήτη το οποίο είναι επικίνδυνο από πλευράς ασφάλειας και κατά δεύτερον διότι προαπαιτεί από τους χρήστες να έχουν γνώσεις στο να χειρίζονται μία συστοιχία με Κυβερνήτη.

Και τα δύο αυτά προβλήματα μπορούν να αντιμετωπιστούν μέσω πρακτικών και στοχευμένων UIs Διαχείρισης. Θα πρέπει λοιπόν τα UIs να προσθέτουν άλλο ένα επίπεδο αφαιρετικότητας πάνω από τον χειρισμό των *CRDs* στην συστοιχία. Πρώτον, θα απεικονίζουν στον χρήστη μόνο τις απαραίτητες πληροφορίες που τον ενδιαφέρουν σχετικά με τα στιγμιότυπα, αποκρύπτοντας όσο το δυνατόν περισσότερο έννοιες του Κυβερνήτη. Δεύτερον, το backend αυτών των εφαρμογών θα είναι υπεύθυνο να χειρίζεται όλες τις επίπονες και τεχνικές λεπτομέρειες για την διαχείριση των εκάστοτε *CRDs*.

Πιο συγκεκριμένα, θα πρέπει αυτά τα web applications να παρέχουν τις εξής λειτουργίες:

- **CRUD Ενέργειες στα CRDs**: Ο χρήστης θα πρέπει να έχει πλήρη έλεγχο πάνω στα στιγμιότυπα των εργαλείων. Τα UIs θα πρέπει να του απεικονίζουν τα ζωντανά στιγμιότυπα, να τον αφήνουν να διαγράφει όσα δεν χρειάζεται, να συνδέεται σε αυτά ή να δημιουργεί καινούργια. Επιπλέον θα τον ενημερώνει για την κατάστασή τους και θα ανταποκρίνεται ζωντανά σε αλλαγές που θα γίνονται στην συστοιχία.

- **Αποκρύπτουν τις λεπτομέρειες του Κυβερνήτη**: Ο χρήστης δεν θα πρέπει να εμπλέκεται καθόλου με τον Κυβερνήτη. Τα UIs θα πρέπει να τον αφήνουν να ρυθμίζει μόνο τις απαραίτητες ρυθμίσεις σχετικά με το εργαλείο και τους πόρους που θα καταναλώσει αποκρύπτοντας ταυτόχρονα όσο το δυνατόν περισσότερο τις έννοιες και λεπτομέρειες του Κυβερνήτη.

Στα επόμενα κεφάλαια θα περιγράψουμε με περισσότερες λεπτομέρειες τις αρχές σχεδίασης που διέπουν το Frontend και Backend μέρος αυτών των εφαρμογών.

1.3.5.1 Εξουσιοδότηση και Δικαιώματα

Ένα ζήτημα μείζονος σημασίας αποτελεί ο τρόπος με τον οποίο τα web apps χειρίζονται την πιστοποίηση ταυτότητας και πως γίνεται έλεγχος εξουσιοδότησης. Ο συγκεκριμένος μηχανισμός επηρεάζει άμεσα τα στιγμιότυπα που θα μπορεί να βλέπει ο κάθε χρήστης καθώς και τις ενέργειες που θα επιτρέπεται να εκτελέσει πάνω σε αυτά τα στιγμιότυπα. Ο τρόπος με τον οποίο γίνεται η πιστοποίηση ταυτότητας, και κατ' επέκταση και ο έλεγχος εξουσιοδότησης, πέρασε από δύο στάδια.

Μέχρι και την έκδοση 0.5 του Kubeflow δεν υπήρχε καμία μορφή πιστοποίησης ταυτότητας και γενικότερα η έννοια του εξατομικευμένου χρήστη. Όλοι οι χρήστες είχαν δικαιώματα σε όλα τα αντικείμενα και μπορούσαν να πειράξουν οποιοδήποτε στιγμιότυπο σε οποιοδήποτε *Namespace*. Οπότε, αντίστοιχα και τα web apps δεν θα κάνουν κανέναν έλεγχο ταυτοποίησης ή εξουσιοδότησης. Θα εκτελέσουν, δηλαδή, οποιαδήποτε ενέργεια τους κληθεί.

Από την έκδοση 0.6 και μετά εισήλθε στο Kubeflow η έννοια του εξατομικευμένου χρήστη. Για να μπορέσει ένας χρήστης να χρησιμοποιήσει το Kubeflow θα πρέπει πρώτα να συνδεθεί σε μία κεντρική σελίδα ταυτοποίησης. Οποιοδήποτε πακέτο έχει προορισμό ένα μονοπάτι στην διεύθυνση της συστοιχίας θα περνάει από αυτό το κεντρικό σημείο ταυτοποίησης. Η λεπτομέρειες για την διαδικασία ταυτοποίησης εξαρτάται άμεσα από τον πάροχο της συστοιχίας, για παράδειγμα *GCP*, *AWS*, *on-prem*. Αφού συνδεθεί ο χρήστης, πριν προχωρήσει οποιοδήποτε πακέτο στο εσωτερικό της συστοιχίας θα του προστίθεται ένα *http header* το οποίο θα δηλώνει την ταυτότητα του χρήστη. Αυτή τη φορά, τα web apps καλούνται να χρησιμοποιήσουν αυτή την πληροφορία και να ελέγξουν αν ο συγκεκριμένος χρήστης έχει την άδεια να διεκπεραιώσει τη εκάστοτε ενέργεια. Αυτό θα το ελέγξει κάνοντας ένα ερώτημα στο *kfam* [37] το οποίο θα του απαντήσει αν ο χρήστης έχει ή όχι την εξουσιοδότηση για την συγκεκριμένη ενέργεια.

1.3.5.2 Frontend για Έκθεση των CRDs

Τα UIs θα πρέπει να είναι εύχρηστα και κατανοητά αποκρύπτοντας όσο το δυνατόν περισσότερο τις τεχνικές λεπτομέρειες της πλατφόρμας. Για να το πετύχουμε αυτό λάβαμε τις ακόλουθες σχεδιαστικές αποφάσεις σχετικά με τον τρόπο λειτουργίας του

Frontend κάθε αυτού αλλά και πως αυτό συνεννοείται με το Backend.

Καθώς, από την έκδοση 0.6 και μετά, υπάρχει έλεγχος ταυτότητας και εξουσιοδότησης το frontend δεν μπορεί να αφήνει τον χρήστη να επιλέγει οποιοδήποτε *Namespace* επιθυμεί. Θα πρέπει κάπως να αποφασίζει σε ποια από τα *Namespaces* έχει δικαίωμα να πειράξει ώστε να του εμφανίζει μόνο αυτά. Τελικά αποφασίσαμε πως γι αυτόν τον έλεγχο θα είναι υπεύθυνη η κεντρική σελίδα του Kubeflow, η οποία υπάρχει πάντα και απεικονίζει όλα τα άλλα UIs μέσω iframes[53]. Αυτή η απόφαση πάρθηκε κυρίως διότι αλλιώς θα έπρεπε να έχουμε την ίδια λογική σε όλα τα UIs του Kubeflow, κάτι που θα οδηγούσε σε πολύ μεγάλη αντιγραφή κώδικα σε πολλά σημεία. Έτσι λοιπόν τα frontends θα μιλάνε με την κεντρική σελίδα του Kubeflow η οποία θα τα ενημερώνει για το ποια *Namespaces* μπορούν να δείξουν στον χρήστη.

Εξίσου σημαντικό ζήτημα είναι το πως θα βρίσκει τα δεδομένα που χρειάζεται το frontend. Υπήρχαν δύο εναλλακτικές, το frontend να μιλάει το ίδιο με τον Κυβερνήτη ή το backend του να είναι αυτό υπεύθυνο για αυτή την επικοινωνία. Τελικά, κυρίως για λόγους ασφάλειας, επιλέξαμε την δεύτερη υλοποίηση. Το backend θα έχει όλα τα απαραίτητα πιστοποιητικά για να μπορεί να μιλάει με τον Κυβερνήτη της συστοιχίας και το frontend θα χρειάζεται μόνο να ρωτάει το backend για τις πληροφορίες που χρειάζεται. Αυτό όμως απαιτεί και την αντίστοιχη προσοχή στην επιβολή ελέγχου εξουσιοδότησης, καθώς το backend θα είναι προσιτό σε οποιονδήποτε γνωρίζει την διεύθυνση της συστοιχίας που τρέχει Kubeflow.

Τέλος το frontend θα πρέπει να έχει, τουλάχιστον, τρεις βασικές σελίδες. Μία αρχική, μία φόρμα για δημιουργία καινούργιων στιγμιότυπων και μία για να βλέπουμε τα logs και πιθανά σφάλματα των ζωντανών στιγμιοτύπων.

1.3.5.3 REST Backend APIs Χωρίς Κατάσταση

Το backend θα είναι υπεύθυνο να διαβάζει τα δεδομένα από τον Κυβερνήτη, να τα διαμορφώνει σε κατάλληλη μορφή και να τα σερβίρει στο frontend. Ταυτόχρονα θα πρέπει να έχει ισχυρούς ελέγχους εξουσιοδότησης ώστε να μην δώσει την δυνατότητα σε χρήστες να πειράξουν πόρους για τους οποίους δεν έχουν δικαιώματα.

Αρχικά, η πιο σημαντική σχεδιαστική απόφαση είναι πως όλη η κατάσταση που πρέπει να γνωρίζει το backend θα βρίσκεται στον Κυβερνήτη. Αυτό σημαίνει πως για οποιο-

δήποτε ερώτημα του frontend θα πρέπει το ίδιο να κάνει ένα ερώτημα στον Κυβερνήτη ώστε να αποκτήσει τα κατάλληλα δεδομένα. Αυτό θα είναι αρκετά βολικό καθώς υπάρχει ήδη μία πληθώρα από διαθέσιμες βιβλιοθήκες οι οποίες διευκολύνουν αρκετά την επικοινωνία με τον Κυβερνήτη. Επιπλέον, το γεγονός ότι το ίδιο το backend δεν πρέπει να κρατάει κάποια κατάσταση διευκολύνει αρκετά στο να μπορέσουμε να αυξήσουμε την διεκπεραίωση, καθώς το μόνο που χρειάζεται να κάνουμε είναι να αυξήσουμε τον αριθμό των *Pods* που τρέχουν το backend.

Το backend θα πρέπει να δίνει την δυνατότητα στους διαχειριστές της συστοιχίας να μπορούν να έχουν έλεγχο πάνω στις ρυθμίσεις που θα μπορούν να πειράξουν οι χρήστες όταν θέλουν να δημιουργήσουν ένα καινούργιο στιγμιότυπο του εργαλείου τους. Συγκεκριμένα, θέλουμε να έχουν την δυνατότητα να μπορούν να επιλέγουν προκαθορισμένες τιμές για όλα τα πεδία, αυτό συμβάλει επίσης και την Εμπειρία Χρηστών (UX), αλλά και να μπορούν να κλειδώσουν συγκεκριμένα πεδία ώστε να μην μπορούν να τα πειράξου οι χρήστες. Αυτό μπορούμε να το επιτύχουμε χρησιμοποιώντας το βασικό αντικείμενο *ConfigMap* του Κυβερνήτη.

Τέλος, αλλά εξίσου σημαντικό, είναι το γεγονός ότι το backend θα χρειαστεί να εφαρμόζει ελέγχους εξουσιοδότησης. Αρχικά προκειμένου να μπορέσει να φτάσει ένα πακέτο έξω από την συστοιχία στο backend θα πρέπει πρώτα ο χρήστης να έχει συνδεθεί στην κεντρική σελίδα ταυτοποίησης της πλατφόρμας. Έτσι, κάθε πακέτο που θα φτάνει τελικά στο backend θα περιέχει πληροφορία σχετικά με την ταυτότητα του χρήστη σε κάποιο *http header*. Το backend θα χρησιμοποιήσει αυτή την πληροφορία και θα κάνει ένα request στο *kfam* [37] ώστε να μάθει αν ο χρήστης έχει το δικαίωμα να εκτελέσει την ενέργεια που επιθυμεί.

1.4 Υλοποίηση

1.4.1 Λογισμικό που Χρησιμοποιήσαμε

Οι ροή εργασίας που προτείναμε στο κεφάλαιο 1.1.2.2 Συνεργατικές Ροές Εργασίας είναι αρκετά περίπλοκη και απαιτεί αρκετά στάδια υλοποίησης. Θα χρειαστεί κώδικα για εξειδικευμένους Controller Κυβερνήτη καθώς και υλοποιήσεις τόσο για frontend όσο και για backend.

Για το **frontend** είχαμε αρκετές εναλλακτικές. Στην αρχή για λόγους απλότητας αλλά και για να εκμεταλλευτούμε τον ήδη υπάρχοντα κώδικα γράψαμε το frontend σε σκέτη JavaScript και HTML. Όμως, καθώς η λογική γίνονταν όλο και περισσότερο πολύπλοκη, κρίναμε πως ήταν απαραίτητο να χρησιμοποιήσουμε ένα framework. Επέλεξα την Angular [46] καθώς είχα περισσότερη εμπειρία με αυτό το framework, αλλά οποιοδήποτε από τα άλλα όπως React, Vue θα ήταν αρκετό.

Για το **backend** είχαμε εξίσου πληθώρα από επιλογές για να επιλέξουμε. Τελικά χρησιμοποιήσαμε το Flask[48] micro-framework κυρίως διότι είχα περισσότερη εξοικείωση με Python, καθώς και γιατί είχε έτοιμη client βιβλιοθήκη για επικοινωνία με τον Κυβερνήτη.

Για την δημιουργία των εξειδικευμένων **Controllers** του Κυβερνήτη για τα καινούργια *CRDs* επιλέξαμε την Go[49]. Αυτό διότι τόσο ο Κυβερνήτης αλλά και ολόκληρο το οικοσύστημα από εφαρμογές που έχουν αναπτυχθεί γύρω από αυτόν είναι όλα γραμμένα στην Go. Έτσι υπήρχε μια πληθώρα από έτοιμα εργαλεία και βιβλιοθήκες για να χρησιμοποιήσουμε.

Πιο συγκεκριμένα χρησιμοποιήσαμε το kubebuilder[50] το οποίο είναι ένα framework για να φτιάχνουμε Controllers και *CustomResourceDefinitions*. Πρόκειται για αρκετά ώριμη δουλειά, όπου δημιουργούσε αρκετή από την προκαθορισμένη δομή για Controllers, εντάσσοντας όλη τη σοφία που είχαν οι σχεδιαστές τόσο του project όσο και του Κυβερνήτη, ώστε εμείς να μπορούμε να ασχοληθούμε μόνο με την λογική του Controller.

1.4.2 UI Διαχείρισης

Στο κεφάλαιο 1.3 Σχεδίαση αναλύσαμε τον τρόπο σκέψης και τις σχεδιαστικές αποφάσεις πίσω από τις προτεινόμενες ροές εργασίας. Σε αυτό το κεφάλαιο θα εντρυφήσουμε στις πιο τεχνικές λεπτομέρειες της υλοποίησης.

1.4.2.1 Χτίσιμο και Τρέξιμο Κώδικα

Τα web applications αποτελούνται από τα αρχεία της JavaScript, HTML, CSS για το frontend και από αρχεία Python για το backend. Το τελικό εκτελέσιμο θα είναι ένα

Docker Image το οποίο θα τα περιέχει και θα χρησιμοποιείται από *Pods* στην συστοιχία.

Πιο συγκεκριμένα, το **frontend** γίνεται compile από την TypeScript σε σκέτη JavaScript κάνοντας και διάφορες βελτιστοποιήσεις, όπως *transpiling, bundling, minifying* και *packaging*. Το **backend** δεν χρειάζεται να μεταγλωττιστεί καθώς η Python είναι γλώσσα που τρέχει από διερμηνέα. Τα αρχεία του frontend θα πακεταριστούν μαζί με του backend σε ένα Docker Image μέσω του *Dockerfile*. Έτσι το backend θα είναι υπεύθυνο να σερβίρει και το REST API αλλά και τα αρχεία του UI. Το τελικό αυτό Docker Image θα είναι περίπου 600MBs το οποίο είναι αρκετά ελαφρύ.

Για τους εξειδικευμένους *Controllers* θα χρειαστεί να φτιάξουμε και εδώ Docker Images ώστε να τους τρέξουμε σε *Pods*. Το kubebuilder που χρησιμοποιήσαμε για να δημιουργήσουμε τις βάσεις των Controllers δημιουργεί από μόνο του το απαραίτητο *Dockerfile* το οποίο θα χρησιμοποιήσουμε. Θα χρειαστεί, όμως, να δημιουργήσουμε έξτρα yaml αρχεία τα οποία θα δίνουν δικαιώματα στα *Pods* να χειρίζονται τα εκάστοτε στιγμιότυπα.

1.4.2.2 Frontend

Ο σκοπός του frontend είναι να διευκολύνει τον χρήστη να χειρίζεται τα στιγμιότυπα των εργαλείων του. Για να το επιτύχει αυτό, το frontend θα πρέπει να έχει τουλάχιστον τρεις σελίδες: την αρχική, μία φόρμα για δημιουργία καινούργιων στιγμιοτύπων και μία σελίδα για τα logs.

Ο σκοπός της **κύριας σελίδες** είναι να ενημερώνει αποτελεσματικά τον χρήστη για την κατάσταση των στιγμιοτύπων των εργαλείων του ζωντανά. Αυτό σημαίνει πως θα πρέπει να μπορεί να είναι συνεχώς ενήμερο αλλά και να δείχνει στον χρήστη τις πληροφορίες που τον ενδιαφέρουν. Για να μπορεί να είναι ενήμερο, το frontend θα εφαρμόσει τον μηχανισμό *polling* και συγκεκριμένα *Exponential Backoff* [55]. Δηλαδή, θα ρωτάει το backend ανά χρονικά διαστήματα για την κατάσταση των στιγμιοτύπων και θα την απεικονίζει. Το περιεχόμενο κάθε αυτό που θα απεικονίζει το κάθε frontend εξαρτάται μεν από το εκάστοτε εργαλείο, όμως για κάθε ένα θα υπάρχουν κάποιες μετρικές που θα υποδηλώνουν τους πόρους που θα καταναλώνει αυτό το στιγμιότυπο, είτε αυτοί είναι υπολογιστικοί είτε δεδομένα.

Επιπλέον, θα δίνει στον χρήστη την επιλογή να δημιουργήσει ένα καινούργιο στιγμιότυπο του εργαλείου του, να διαγράψει ή να συνδεθεί σε ένα υπάρχον. Στην περίπτωση της διαγραφής το frontend θα στείλει ένα *DELETE* ρήμα στο REST backend για το συγκεκριμένο στιγμιότυπο, το οποίο μετά θα στείλει αντίστοιχα request στον Κυβερνήτη. Όταν ο χρήστης θελήσει να *συνδεθεί* σε υπάρχον στιγμιότυπο τότε θα τον παροτρύνει σε νέο tab όπου θα μιλάει με το *Service* που είναι υπεύθυνο για την δικτυακή κίνηση.

Για την **φόρμα** όπου ο χρήστης θα φτιάχνει καινούργια στιγμιότυπα είναι πιο απλή η υλοποίηση. Η σελίδα αυτή θα είναι, όπως είπαμε πριν, μία φόρμα η οποία όμως θα πρέπει να λάβει τις αρχικές τιμές της από το backend. Το backend θα έχει ένα συγκεκριμένο μονοπάτι όπου θα στέλνει στο frontend ποιες τιμές να βάλει σε ποια πεδία, καθώς και ποια πεδία να μην αφήσει τον χρήστη να τα πειράξει. Εδώ ο διαχειριστής θα έχει επιλέξει αυτές τις τιμές. Αφού ο χρήστης συμπληρώσει τις τιμές της φόρμας τότε το frontend θα στείλει ένα *POST* ρήμα στο backend με τις συμπληρωμένες τιμές, το οποίο πάλι με την σειρά του θα φτιάξει τον αντικείμενο και θα το δημιουργήσει στην βάση του Κυβερνήτη ώστε να το χειριστούν οι αντίστοιχοι *Controllers*.

Τέλος, για τα **logs** το frontend θα δείχνει στον χρήστη τα απαραίτητα στοιχεία ώστε να μπορεί να καταλαβαίνει αναλυτικά σε τι κατάσταση βρίσκονται τα στιγμιότυπά του. Συγκεκριμένα για κάθε στιγμιότυπο θα του εμφανίζει σε μία καρτέλα τα logs από τους Containers που τρέχουν στο *Pod* και σε μία άλλη όλα τα *Events* που έχουν δημιουργηθεί από αυτό το Pod. Τα *Events* είναι κι αυτά βασικά αντικείμενα του Κυβερνήτη, όμως θα εμφανίζουμε στον χρήστη μόνο το σχετικό μήνυμα καθενός *Event* το οποίο υπάρχει ακριβώς γι αυτό τον λόγο.

1.4.2.3 Backend

Όπως αναφέραμε και στην ενότητα 1.3.5.3 REST Backend APIs Χωρίς Κατάσταση το backend του κάθε web app θα έχει αποθηκευμένη την κατάστασή του στην εξωτερική, για αυτό, βάση του Κυβερνήτη. Οπότε θα χρειάζεται να μιλάει συνέχεια με αυτήν για να μπορεί να τροφοδοτεί τα frontends με τα απαραίτητα δεδομένα.

Επειδή όμως πρέπει να μιλήσει με τον Κυβερνήτη για να λάβει τα δεδομένα που χρειάζεται, εισάγεται άλλο ένα βήμα το οποίο μπορεί να αποτύχει. Θα πρέπει γι αυτό να υπάρχει μια σύμβαση μεταξύ του frontend και του backend σχετικά με το πότε το

backend κατάφερε να επιστρέψει τα δεδομένα πίσω επιτυχώς ή αν συνέβη κάποιο σφάλμα. Αρχικά, τα requests θα έχουν json format το οποίο δηλώνεται με το *Content-Type: application/json* http header. Έπειτα, τα μηνύματα που θα στέλνει το backend θα έχουν πάντα ένα πεδίο *success* το οποίο θα δηλώνει αν το backend κατάφερε να κάνει την ενέργεια που του ζήτησε το frontend. Σε περίπτωση που απέτυχε, είτε για λόγους εξουσιοδότησης, σφάλμα με την επικοινωνία με τον Κυβερνήτη ή οτιδήποτε άλλο, τότε αυτό το πεδίο θα είναι false και θα υπάρχει άλλο ένα πεδίο *log* στο response το οποίο θα περιέχει ένα μήνυμα όπου θα μπορεί να εμφανίσει στον χρήστη.

Το backend είναι, επιπλέον, υπεύθυνο στο να στέλνει τις προκαθορισμένες τιμές στην φόρμα για την δημιουργία νέου στιγμιότυπου στο frontend. Ο διαχειριστής θα έχει γράψει τις τιμές αυτές, καθώς και ποια από τα πεδία επιθυμεί να είναι κλειδωμένα σε ένα *ConfigMap*. Αυτό θα είναι mounted στο *Pod* του backend ως ένα αρχείο, το οποίο θα και θα διαβάσει για να το στείλει. Έτσι ο διαχειριστής μπορεί να αλλάξει απλά τις τιμές αυτού του *ConfigMap* επιτόπου και τα *Pods*, αργά ή γρήγορα, θα δουν αυτή την αλλαγή και θα στέλνουν τις καινούργιες ρυθμίσεις.

1.4.2.4 Έλεγχος Ταυτοποίησης και Εξουσιοδότησης

Όπως αναφέραμε και στο κεφάλαιο 1.3.5.1 Εξουσιοδότηση και Δικαιώματα τα web apps θα πρέπει να σέβονται τους κανόνες εξουσιοδότησης[12]. Η πληροφορία για τον συνδεδεμένο χρήστη θα υπάρχει πάντα στα requests που έρχονται σε κάποιο *http header*. Αυτή την πληροφορία θα χρησιμοποιήσει το backend για να ελέγξει αν ο χρήστης έχει τα δικαιώματα να εκτελέσει τη συγκεκριμένη ενέργεια.

Πιο συγκεκριμένα, αφού αποκομίσει την ταυτότητα του χρήστη από το *http header* θα χρησιμοποιήσει το *kfam* [13]. Σε περίπτωση που δεν μπορέσει να βρει την ταυτότητα του χρήστη τότε θα θέσει το *success* πεδίο σε false μαζί με αντίστοιχο μήνυμα στο πεδίο *log* και θα επιστρέψει. Αφού πάρει το όνομα του συνδεδεμένου χρήστη θα κάνει *GET* request στο *kfam* το οποίο θα του επιστρέψει μία λίστα με *Namespaces* στα οποία ο χρήστης έχει δικαιώματα, με βάση την οποία θα αποφανθεί αν θα του επιτρέψει να εκτελέσει την ενέργεια ή όχι.

[12] για εκδόσεις Kubeflow 0.6 και πάνω
[13] Kubeflow Access Management

1.4.3 Χρόνος Ζωής των Αδρανή CRDs

Ο ορισμός του αδρανή *CustomResource* διαφέρει από αντικείμενο σε αντικείμενο, ακόμη και του ίδιου είδους. Επιπλέον, διαφορετικά εργαλεία μπορεί τα ίδια να έχουν τον δικό τους ορισμό για το πότε είναι αδρανή, όπως τα Jupyter Notebooks για παράδειγμα. Θα πρέπει λοιπόν η λογική για τον σταματημό των αδρανή *CustomResources* να μπορεί να λαμβάνει υπόψιν εξειδικευμένους ορισμούς αλλά ταυτόχρονα να έχει και μία προκαθορισμένη συμπεριφορά για να δουλεύει σε μία γενική περίπτωση.

Ο βασικός κώδικας αυτού του μηχανισμού αφορά την απόφαση για το αν ένα *CustomResource* είναι αδρανή ή όχι. Στην γενική περίπτωση θα αποφανθούμε για το αν ένα *CustomResource* είναι αδρανή ή όχι με βάση τον ρυθμό δικτυακής κίνησης που καταλήγει στο *Pod*. Αυτές τις μετρικές μας τις δίνει το ISTIO. Αυτή η λογική θα γραφεί ως μία βιβλιοθήκη Go προκειμένου να μπορέσει να χρησιμοποιηθεί από οποιονδήποτε από τους Controllers, είτε για *Notebooks, Tensorboard, PVCViewers* και για οποιοδήποτε άλλο *CustomResource*. Για εξειδικευμένες περιπτώσεις, όπως για τα Notebooks, όπου τα ίδια τα εργαλεία εξάγουν αυτή την πληροφορία θα πρέπει ο εκάστοτε Controller να γράφει την δικιά του συνάρτηση που θα αποφασίζει αν ένα *CustomResource* είναι αδρανή ή όχι.

Αφού αποφασιστεί ότι ένα *CustomResource* είναι αδρανή θα πρέπει ο Controller να προχωρήσει στην επόμενη φάση όπου είναι το σταμάτημα των *Pods*. Σύμφωνα με τις καλές προγραμματιστικές πρακτικές των Controllers, όταν είναι αδρανή ένα *CustomResource* ο Controller θα του προσθέτει απλά ένα *annotation* [14] με τιμή το timestamp όπου το έβαλε. Έπειτα θα πρέπει ο κάθε Controller στο *Reconciliation Loop* που εκτελεί να κοιτάει αν αυτό το *annotation* είναι γραμμένο. Αν είναι, τότε θα πρέπει ο ίδιος να πειράζει το πλήθος των *Pods* που δημιουργεί. Αλλιώς θα σιγουρεύει ότι φτιάχνει, τουλάχιστον, ένα. Μπορεί όμως ένας Controller να θελήσει να διαγράψει τελείως το *CustomResource* σε περίπτωση που είναι αδρανή. Αυτό εφαρμόζεται στους PVCViewers, καθώς δεν έχει νόημα να υπάρχουν από την στιγμή που δεν θα χρησιμοποιούνται άμεσα. Πάλι είναι ελεύθερος ο κάθε Controller στο ποια λογική θα ακολουθήσει αν δει το συγκεκριμένο *annotation*.

[14] kubeflow-resource-stopped

1.4.4 Controllers για τα CRDs

Όλοι οι Controllers, για *Notebooks, Tensorboard, PVCViewers*, είναι γραμμένοι σε Go. Για Notebooks και Tensorboard είναι γραμμένοι με το πρότζεκτ kubebuilder[50]. Παρόλο που καθένας τους συμβάλει σε ένα διαφορετικό μέρος της προτεινόμενης ροής εργασίας και οι τρεις τους ακολουθούν το ίδιο σχεδιαστικό μοτίβο. Υπάρχει το *Reconciliation Loop* το οποίο τρέχει συνέχεια και για κάθε *CustomResource* εξασφαλίζει ότι φτιάχνονται τα απαραίτητα *Deployments, Services* και *VirtualServices*.

Τα *Service* και *VirtualService* που φτιάχνουν είναι σχεδόν ίδια μεταξύ των τριών Controllers. Η μόνη διαφορά έχει να κάνει με το μονοπάτι στο οποίο θα ακούει το *VirtualService* για εξωτερική από την συστοιχία δικτυακή κίνηση.

Οι Controllers θα είναι, επίσης, υπεύθυνοι για να ελέγχουν περιοδικά αν τα αντικείμενά τους είναι αδρανή για αρκετό διάστημα ώστε να τα σταματήσουν. Ο έλεγχος αυτός είναι μέρος του *reconciliation loop*.

1.4.4.1 Jupyter Notebooks

Το spec ενός *Notebook CustomResource* έχει μόνο ένα πεδίο, το οποίο είναι ένα *PodTemplateSpec*. Ο Controller θα φτιάξει ένα *Deployment* για αυτό το Notebook χρησιμοποιώντας αυτόν τον ορισμό. Ουσιαστικά θα είναι ένα αφαιρετικό επίπεδο παραπάνω για την δημιουργία *Pods*. Ο λόγος γι αυτή την απόφαση ήταν ότι δεν θέλαμε να περιορίσουμε τις δυνατότητες παραμετροποίησης του Notebook και να μην αλλάζουμε συνέχεια τον ορισμό του προκειμένου να υποστηρίξουμε παραπάνω παραμετροποίηση, που υπάρχει ήδη στα *Pods*. Δεδομένου αυτού του ορισμού θα παίξει μείζον ρόλο το UI το οποίο θα χειρίζεται τα Notebooks, καθώς θα είναι υπεύθυνο να διευκολύνει τον χρήστη ποιες τιμές να χρησιμοποιήσει για να ξεκινήσει ένα Notebook. Ο Controller θα φτιάχνει επιπλέον ένα *Service* και ένα *VirtualService* προκειμένου να ρυθμίσει τους κανόνες δρομολόγησης.

Το κάθε Notebook αντικείμενο θα έχει και ένα πεδίο *status* το οποίο θα ανανεώνει ο Controller καθώς αλλάζει η κατάσταση του αντικειμένου. Το πεδίο αυτό θα χρησιμοποιείται και από το backend του UI προκειμένου να στέλνει στο frontend την κατάσταση του εκάστοτε Notebook. Πιο συγκεκριμένα σε αυτό το πεδίο θα απεικονίζεται αρχικά το *ContainerStatus* του πρώτου Container του *Pod*. Υπάρχει μια μικρή προβλη-

ματική συμπεριφορά με την εξής επιλογή. Σε περίπτωση που δεν μπορεί να φτιαχτεί το Notebook, τότε ο Container δεν θα φτιαχτεί ποτέ και το *status* αυτού του αντικειμένου δεν θα αρχικοποιηθεί ποτέ. Για τον λόγω αυτό το UI απεικονίζει επίσης και τα *Events* του Notebook. Ο Controller παρακολουθεί ταυτόχρονα και όλα τα *Events* που δημιουργεί το *StatefulSet* που φτιάχνει και τα ξανά στέλνει ως *Notebook Events*.

Κάθε Notebook που δημιουργείται είναι προσβάσιμο από ένα μοναδικό URL στο οποίο μπορεί να συνδεθεί ο χρήστης. Το URL αυτό έχει μορφή [15]. Το *VirtualService* που δημιουργεί ο Controller είναι υπεύθυνο για να ανακατευθύνει την κίνηση από αυτό το σημείο στο αντίστοιχο *Service* ώστε τελικά να φτάσει στο *Pod*. Το πακέτο που θα φτάσει μέσα στο Notebook, όμως, θα έχει ως προορισμό το συγκεκριμένο μονοπάτι. Ένα Notebook ουσιαστικά είναι ένας Server που δέχεται δικτυακή κίνηση. Για να μπορέσει να απαντάει σωστά το Notebooks θα πρέπει να γνωρίζει ότι οι χρήστες θεωρούν ως μονοπάτι όχι τη ρίζα / αλλά το προηγούμενο μονοπάτι. Το Notebook έχει ένα flag για αυτή την περίπτωση, το *–NotebookApp.base_url*, το οποίο θα θέτουμε κατάλληλα στο entrypoint του Container.

Μπορεί όμως κάποιος να θέλει να φτιάξει το δικό του Notebook image και να το χρησιμοποιήσει στο Kubeflow. Όμως σε αυτή την περίπτωση δεν μπορεί να δώσει με το χέρι το base_url που θέλει για το Notebook. Γι αυτή την περίπτωση ο Controller θα θέτει μία μεταβλητή περιβάλλοντος στον Container, την *NB_PREFIX*, η οποία θα μπορεί να χρησιμοποιηθεί από το entrypoint του Container ώστε να τίθεται δυναμικά το base_url.

1.4.4.2 Οπτικοποίηση Μετρικών & Tensorboard

Αυτή τη φορά το spec του συγκεκριμένου *CustomResource* το ενδιαφέρει μόνο να του ορίσουμε που ζουν τα δεδομένα. Σε αυτή την περίπτωση θα είναι πολύ πιο σύντομο και θα περιέχει δύο πιθανά πεδία, ένα *log* το και ένα *pvc-name*.

Το πρώτο θα είναι για να ορίζουμε την τοποθεσία δεδομένων για απεικόνιση logs τα οποία βρίσκονται σε κάποιο δικτυακό αποθηκευτικό σύστημα. Το entrypoint του Container θα ξεκινήσει το Tensorboard και θα του δώσει ως είσοδο δεδομένων την συγκεκριμένη διεύθυνση. Το Pod που θα δημιουργηθεί θα πρέπει να έχει πρόσβαση

[15] */notebook/<namespace>/<notebook-name>*

σε αυτή τη διεύθυνση. Για το πεδίο *pvc-name* ο Controller θα δίνει πρόσβαση στο συγκεκριμένο *PVC* στο *Pod* που θα δημιουργήσει και θα το ρυθμίσει ώστε να διαβάζει τα δεδομένα από εκεί.

Υπάρχει όμως περίπτωση το *PVC* να είναι σε mode *ReadWriteOnce* με αποτέλεσμα ο scheduler από μόνος του να μην ξεκινάει ποτέ το Tensorboard *Pod* με το συγκεκριμένο *PVC* μέχρι αυτό να ελευθερωθεί. Για να αντιμετωπίζουμε τον περιορισμό αυτό χρησιμοποιήσαμε κάποια *ψηλά γράμματα* του Κυβερνήτη. Μπορούμε να έχουμε δύο *Pods* να χρησιμοποιούν το ίδιο *PVC* που είναι *ReadWriteOnce* αρκεί όμως και τα δύο να τρέχουν στον ίδιο Κόμβο. Έτσι ο Controller πριν ξεκινήσει το *Pod* θα κοιτάξει αν το *PVC* χρησιμοποιείται και αν ναι και είναι και *ReadWriteOnce*, τότε θα θέσει το πεδίο *nodeName* κατάλληλα ώστε το *Pod* να ξεκινήσει σε εκείνο τον Κόμβο. Για να σιγουρέψουμε ότι θα ξεκινήσει θα του θέσουμε ελάχιστους πόρους για να καταναλώνει, το οποίο δεν είναι σημαντικό πρόβλημα μιας και από την φύση του δεν έχει ανάγκη για αρκετούς.

Όπως και πριν ο Controller θα χρειαστεί να φτιάχνει και ένα *Service* και *VirtualService* προκειμένου να ρυθμίσου τους κανόνες δρομολόγησης.

Τέλος, θα πρέπει να παρακολουθεί τα *Events* που θα δημιουργεί το *Deployment* και να τα προωθεί ως *Viewer Events*. Έτσι το UI θα μπορεί να τα απεικονίζει στον χρήστη στην σελίδα logs. Επιπλέον, ο Controller θα χρειαστεί να ανανεώνει το πεδίο *status* του κάθε αντικειμένου κατάλληλα με βάση την πληροφορία που έχει από το *Deployment*.

1.4.4.3 PVCViewers

Αυτός ο Controller είναι αρκετά παρόμοιος με τον προηγούμενο, για τα Tensorboard. Δημιουργεί και αυτός ένα *Deployment, Service* καθώς και *VirtualService* προκειμένου να σηκώσει ένα web app το οποίο θα επιτρέπει στους χρήστες να χειρίζονται τα αρχεία από *PVCs*.

Το spec από αυτό το *CustomResource* θα περιέχει μόνο ένα πεδίο, το όνομα του *PVC* που θέλουμε να εξερευνήσουμε. Ο Controller θα εφαρμόσει την ίδια λογική για να ξεκινήσει ελαφριά *Pods* στους κατάλληλους Κόμβους προκειμένου να μπορεί πάντα ο χρήστης να επεξεργάζεται τα αρχεία από οποιοδήποτε *PVC*.

Αφού ο Controller έχει εξασφαλίσει πως το *Pod* θα μπορέσει να ξεκινήσει, θα χρειαστεί

να ρυθμίσουμε την εφαρμογή που θα τρέξει στον Container. Θα χρησιμοποιήσουμε το Docker Image *Cloud Commander* το οποίο είναι παρέχει ένα διαδραστικό περιβάλλον μέσα από τον browser του χρήστη. Θα πρέπει να ρυθμίσουμε το web app αυτό πρώτον να δέχεται κίνηση από μονοπάτι εκτός της ρίζας και κατά δεύτερον να ξεκινάει με αρχικό φάκελο το *PVC* που του έδωσε ο χρήστης.

Ο περιορισμός με το μονοπάτι είναι ο ίδιος και με το Notebook *CustomResource*. Το αντικείμενο θα είναι διαθέσιμο στους χρήστες σε συγκεκριμένο μονοπάτι [16], οπότε τα πακέτα που θα φτάνουν στο web app θα έχουν αυτήν ως διεύθυνση αποστολής. Το εκτελέσιμο έχει το δικό του flag για να του ορίζουμε αυτή τη συμπεριφορά, το *–prefix* και *–prefix-socket*.

Τέλος, ο Controller οφείλει να ενημερώνει το *status* του κάθε *PVCViewer* αντικειμένου, καθώς και να προωθεί τα αντίστοιχα *Events*. Το συγκεκριμένο resource δεν θα εμφανίζεται στον χρήστη αναλυτικά, καθώς θα χρησιμοποιείται μόνο όταν επιλέγει να εξερευνήσει έναν δίσκο. Όμως, σε περίπτωση που κάτι πάει στραβά θα βοηθήσει αρκετά στο debugging ο συγκεκριμένος μηχανισμός.

[16] */volumes/devices/<namespace>/<pvc-name>*

1.5 Επίλογος

Σε αυτό το τελευταίο κεφάλαιο θα παρουσιάσουμε μία σύνοψη από την δουλειά μας, εξετάζοντας τους αρχικούς μας στόχους και τη τελική σχεδίαση. Έπειτα θα παρουσιάσουμε πιθανές μελλοντικές επεκτάσεις και προοπτικές που μπορούν να εφαρμοστούν.

1.5.1 Τελικές Παρατηρήσεις

Συνολικά, ο στόχος μας ήταν να βελτιώσουμε τις υπάρχουσες ροές εργασίας που παρέχει το Kubeflow λαμβάνοντας υπόψιν την συνεργατική τους φύση καθώς και την βαρύτητα των δεδομένων. Εκτός, όμως, από αυτό είχαμε θέσει και έναν δευτερεύον στόχο: να μπορέσω να συνεργαστώ ισάξια και αποτελεσματικά με άλλους μηχανικούς και προγραμματιστές από όλο τον κόσμο συνεισφέροντας ταυτόχρονα σε ένα πρότζεκτ ανοιχτού κώδικα βεληνεκούς όπως του Kubeflow. Στο σημείο αυτό πιστεύω μπορούμε να πούμε πως πετύχαμε και τους δύο μας στόχους.

Πιο αναλυτικά, όσο αφορά τον πρώτο μας στόχο, οι προτεινόμενες ροές εργασίας έγιναν αμέσως αποδεκτές από την κοινότητα και τους χρήστες του Kubeflow. Η ροή με τα Jupyter Notebooks είναι μέρος του Kubeflow από την έκδοση 0.5 και των Tensorboard στην έκδοση 0.8. Μάλιστα, ως απόδειξη της χρησιμότητάς τους, ο κώδικάς μας και ο σχεδιασμός μας χρησιμοποιείται από σημαντικές εταιρείες στον χώρο όπως η IBM, η Cisco και η Google.

Όσο αφορά τον δεύτερο στόχο, λόγω της έλλειψης εμπειρίας μου υπήρχαν στιγμές που ήταν αρκετά πιεστικές. Όμως, με συνεχή προσπάθεια βγάλαμε ένα αρκετά ικανοποιητικό αποτέλεσμα και, πιστεύω, στάθηκα με αξιοπρέπεια στους υπόλοιπους προγραμματιστές και μηχανικούς όπου συναναστράφηκα.

1.5.2 Μελλοντικές Επεκτάσεις

Παρόλο που βελτιώσαμε τις υπάρχουσες ροές εργασίας του Kubeflow υπάρχει σίγουρα περιθώριο για περαιτέρω εξέλιξη.

Αρχικά, μιας και καινούργιες μέθοδοι και εργαλεία δημιουργούνται μέρα με την μέρα θα πρέπει συνεχώς να ενημερωνόμαστε και να είμαστε σε επικοινωνία με τους χρή-

στες. Έτσι θα μπορέσουμε να εντοπίσουμε σφάλματα και ατέλειες στις υπάρχουσες υλοποιήσεις που δεν είχαμε λάβει αρχικά υπόψιν.

Επιπλέον, τα UIs που δημιουργήσαμε μπορεί να παρέχουν αρκετές διευκολύνσεις όμως ταυτόχρονα έχουν και κάποιους περιορισμούς. Υπάρχει μια πληθώρα ρυθμίσεων του Κυβερνήτη που θα μπορούσαν να τροποποιηθούν από τα UIs που όμως δεν είναι εφικτό με την τωρινή υλοποίηση. Για αυτό το λόγω, θα χρειαστεί να σχεδιάσουμε ακόμη πιο ολιστικό σχεδιασμό που θα καλύπτει και πιο πολύπλοκες παραμετροποιήσεις.

Τέλος, άλλες βελτιστοποιήσεις που θα μπορούσαμε να κάνουμε από πλευράς υποστήριξης *PVCs* και τοπικών δεδομένων θα ήταν πάνω στο *component* για την εξαγωγή και το σερβίρισμα των Μοντέλων. Η τωρινή λύση του Kubeflow, το *KFServing*, έχει μηδαμινή υποστήριξη για *PVCs* και υποστηρίζει μόνο δεδομένα που ζουν σε Object Stores.

2 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.

2.1 Problem Statement

Launching a Machine Learning product requires advanced tooling, skill-sets, as well as workflows. Professionals specialized in different roles will need to be able to both apply their expertise on the necessary parts of the pipeline as well as hand off their work for next phases. It is a highly collaborative process that demands efficient communication between the engineers working on the different moving parts. This highlights the need for a platform that will both include the necessary tooling and provide a competent User Experience (UX) for running such convoluted and collaborative workflows.

But, creating and maintaining a platform for reliably producing and deploying machine learning models requires careful orchestration of many components: a learner for generating models based on training data, modules for analyzing and validating both data as well as models, and finally infrastructure for serving models in production. This becomes particularly challenging when data changes over time and fresh

models need to be produced continuously. Unfortunately, such orchestration is often done ad-hoc using glue code and custom scripts developed by individual teams for specific use cases, leading to duplicated effort and fragile systems with high technical debt. There is a plethora of OSS[3] solutions, such as Kubeflow which we will work on, that try to tackle this problem but they are still on their early stages of development.

Specifically, Kubeflow[5] is an open source Kubernetes-native[4] stack for developing, orchestrating, deploying, and running scalable and portable machine learning workloads. It is a bundle of software artifacts for running and orchestrating tools and pipelines that the Data and ML Scientists are used to working with on top of Kubernetes. However, in its current version[1] the provided UX is not intuitive for performing such collaborative workflows regarding how the data is passed between different steps, which is a key factor in these workflows. Also, Kubeflow expects the user to be comfortable and competent with Kubernetes concepts in order to be able to run and debug their workflow. This is a significant issue, since most Machine Learning Scientists are not familiar with technologies such as Containers or Kubernetes, and they shouldn't be since their job is Machine Learning. As a result, the communication with the DevOps teams will increase dramatically which leads to valuable engineering time to be wasted on friction with the platform.

2.2 Motivation

The motivation for selecting the specific topic to analyze was the different steps in a Machine Learning pipeline, the collaborative nature of such workflows as well the level that existing solutions, such as Kubeflow, manage to provide a compelling platform for running and orchestrating such tasks.

We will first explain what are the steps in a Machine Learning pipeline, what workflows do the engineers use and how can they leverage the work of their colleagues. Then we will showcase Kubeflow's components and tooling that different engineers can use for their distinctive tasks, as well as how effective it is on providing an efficient User Experience and Workflows.

[1] kubeflow v0.4

2.2.1 Machine Learning Components

- **Data Analysis**: Data analysis is a process of inspecting, cleansing, transforming and modeling data with the goal of discovering useful information, informing conclusions and supporting decision-making. Data analysis has multiple facets and approaches, encompassing diverse techniques under a variety of names, and is used in different business, science, and social science domains. In today's business world, data analysis plays a role in making decisions more scientific and helping businesses operate more effectively.

- **Data Transformation**: Data transformation is the process of converting data from one format or structure into another format or structure. It is a fundamental aspect of most data integration and data management[10] tasks such as data wrangling[7], data warehousing[8], data integration[9] and application integration.

- **Data Validation**: Data validation is the process of ensuring data have undergone data cleansing to ensure they have data quality, that is, that they are both correct and useful. It uses routines, often called "validation rules" "validation constraints" or "check routines", that check for correctness, meaningfulness, and security of data that are input to the system. The rules may be implemented through the automated facilities of a data dictionary, or by the inclusion of explicit application program validation logic.

- **Learner**: A core objective of a learner is to generalize from its experience. Generalization in this context is the ability of a learning machine to perform accurately on new, unseen examples/tasks after having experienced a learning data set. The training examples come from some generally unknown probability distribution (considered representative of the space of occurrences) and the learner has to build a general model about this space that enables it to produce sufficiently accurate predictions in new cases.

- **Model Evaluation and Validation**: Model Evaluation helps to find the best model that represents the data and how well the chosen model will work in the future. Model validation is the task of confirming that the outputs of a statistical model have enough fidelity to the outputs of the data-generating process

that the objectives of the investigation can be achieved.

- **Serving**: Model Serving is the process in which the now trained Machine Learning Model gets deployed to production. The model is hosted in a server, so that clients can connect to it and use it for inference.

2.2.2 Collaborative Workflows

By *Collaborative Workflows* we mean a sequence of tasks in which some of the parts must be completed from different people. The level in which people can effectively communicate and hand-off their work and findings drastically dictates the quality of the end result as well as the overall cost invested in the product.

In this thesis our main focus is on the *collaborative workflows* that Machine Learning Scientists have to perform in order to create a product that is utilizing some form of Machine Learning in its internal structure and how to improve them. From the previous section we can see the different Machine Learning Components that are used in such Pipelines. A note-worthy detail is the fact that only one of the tasks, the *Learner*, actually demands the skill-set of a Machine Learning Scientist. All of the other tasks require a broad gamma of professions, such as *Data Scientists* [11], *IT*, *DevOps*, *Software Engineers* just to mention a few. All of them will need to work on their own parts of the Pipeline and continually evaluate their work, as well as the work they have as input from their colleagues.

To better understand such advanced workflows we will need to understand the structure and shape of the artifacts that each one of them produces, as a result of their work, as well as the ways which enable them to hand these results off to their colleagues. In a Pipeline with the mentioned steps, the artifacts that need to be passed on is *data*.

For the first steps of the Pipeline, *Data Analysis, Transformation and Validation*, these artifacts are the datasets[12] that the Data Scientists are working on. After the necessary processing has been done to the available datasets, then the Machine Learning Scientists can use them to create and train their ML models. But this task is not a straight forward one. In order for a competent model to be generated, an iterative process is required. The ML Scientist will have to repeatedly evaluate the generated model's performance to ensure that it satisfies established quality criteria. These are

the *Model Evaluation and Validation* steps in the Pipeline. On top of that, these models will need to be able to be trained again from new data and datasets that might become available. This means that the Data Scientists and the Machine Learning Scientists will need to continually be in touch and exposing their state. It is critical that the Data Scientist will be able to constantly and efficiently deliver their produced datasets to the ML Scientists for further processing.

For the last step, the *serving*, the *input* data will be the trained and polished model and the *output* an application that will be exposing it for inference. Inference is the result of the trained model when it gets exposed to new data, most often from users and the real world. This is done, most of the time, with a REST API[13] that receives requests with the necessary parameters and responds back with the model's inference. So the required input from the previous steps would be trained models, which could also be different timestamps or versions of the same model.

There are many possible ways in which the datasets and the trained models can be handed off between different people and teams. A technology that fits for this role in an *Object Store* [14]. These stores can have the different iterations and versions of both datasets and models stored and ready for use. The only issue in this case would be the network bandwidth required if the data is continuously changing, which most of the time is. The new datasets would have to be uploaded and downloaded multiple times from the different people that would be using them. A more efficient solution would be to allow the data to live in the cluster's local storage and use it whenever possible. Also, it would greatly reduce the network traffic if we could only keep track of the differences in the datasets as different versions, just like git, and not the entire data. Then people could easily sync up with the most recent changes by only downloading these differences on their local files and the remote ones.

At this point it should be clear that the friction for moving the datasets and models between people is the key factor that will dictate the effectiveness of the Pipeline.

2.2.3 The State of Kubeflow

Kubeflow is a Machine Learning stack that provides most tools and pipelines that Machine Learning Scientists use on top of Kubernetes. It aims to make the execution and

management of such collaborative workflows seamless on the cloud.

As of version 0.4, which was the latest version when we started, Kubeflow was a relatively new project with only one and a half year of development under its belt. At this point in time it had successfully incorporated an adequate set of tools that Machine Learning Scientists use, such as Jupyter Notebooks, Tensorboard and even a procedure for running end-to-end workflows, on top of Kubernetes. But still, it lacked necessary features in order to be production ready, such as authentication and multi-user support and most importantly from this thesis' point of view it still didn't effectively capture the collaborative and data driven nature of the Machine Learning Scientists' workflows and thus didn't offer an intuitive user experience.

More specifically in terms of production readiness it was missing user authentication, authorization and most importantly access control mechanisms. There was also no isolation between the resources that belonged to different users so each users could interfere and disrupt the workflows of other users that are using the platform.

From the User Experience point of view while Kubeflow provides most of the tools that Machine Learning Scientists use, the support for moving data from the different steps of a workflow is limited. For example, the workflow of using Jupyter Notebooks did not have an intuitive way for utilizing PVCs. As a result the users would have to constantly move data back and forth between Object Stores which both adds more steps and complexity as well as increase the consumed network bandwidth and resources.

Lastly, Kubeflow was not mature enough to be able to hide the underlying Kubernetes concepts from the end users. This introduces quite a steep learning curve for engaging with it, since most Data and Machine Learning Scientists are not familiar with technologies such as Containers and Kubernetes, and also hurts the overall UX.

2.3 Existing Solutions

In this thesis we decided to work on Kubeflow, as a Machine Learning stack, because it was the most popular at the time in terms of development and community support. In the following sections we will showcase what are the workflows and tools that Kubeflow provides and how competent they are on capturing the real world needs of the end-to-

end Machine Learning workflows.

2.3.1 Kubeflow: Jupyter Notebooks

As of version 0.4, Kubeflow was using JupyterHub for managing and deploying Jupyter Notebooks. For deploying Pods in Kubernetes, JupyterHub created the Kubespawner[17] project. Kubespawner is a backend service for creating the Notebooks as Pods in the cluster, which is managed from the main JupyterHub Pod that is running and coordinating the life-cycle of other Notebooks.

JupyterHub also supported hooks for authentication. In case that no authentication mechanisms are in place in the existing cluster, then the user will have to provide a string that would represent their username. This way each user could create and access their own Hub for Notebooks. After creating one with Kubespawner, the user could access it by first connecting to the central JupyterHub which would then redirect their traffic to their respective Hub Pod.

The main JupyterHub pod was also responsible for health checking the status of the underlying Notebook Pods. If a user's Pod would be deleted, then it would urge the user to create a new one. However, JupyterHub introduced some problematic user experience.

The root of the issue was the fact that JupyterHub was the one responsible for tracking the state, health and do the Load Balancing of the traffic for the Notebooks. More specifically, the the Pod that was running the main JupyterHub was a Single Point of Failure. If this Pod would go down, then all the users would be unable to connect to their Notebooks. Secondly, the Hub's health checks might take longer than expected to detect that a Notebook was unhealthy. This resulted in traffic being redirected to an IP that, most of the times, wouldn't exist. The user would have to manually check the Notebook Pod and if they couldn't debug it they would have to delete it in order for JupyterHub to allow them to create a new one.

2.3.2 Kubeflow: Distributed Training

Kubeflow supported a method for running the training of the model in a distributed fashion. This way the users could write their code locally and then utilize the cluster's resources and horse power by running the actual training on it.

In order to take advantage of Kubernetes' distributed nature for running computations Kubeflow implements the Tensroflow Distributed Training[43] as a Kubernetes first class citizen concept. For this it creates a *TFJob CustomResource Definition* that encapsulates the logic for deploying Distributed TensorFlow Jobs on top of Kubernetes.

A *TFJob* object provides an abstraction on top of the *PS, Worker, Chief* and *Evaluator* processes concepts. With this abstraction the user will be able to fully customize and define their execution graph of the distributed training and Kubernetes will take care of instantiating with the cluster's resources. The user will be able to define the code that each one of the mentioned TensorFlow components will run by specifying Docker Images.

The TFJob Operator will be responsible for keeping track of the set of Pods and their roles in a distributed training session. This information must also be provided to the running Pods since it is necessary for successfully splitting and partitioning the required data and work. Under the hood the controller will set the **TF_CONFIG** environment var on each Pod's containers and the underlying code will need to update its ClusterSpec from this environment variable.

While this implementation manages to hide some of the complexities of running a distributed TensorFlow job on Kubernetes, it has a terrible User Experience since there is no UI for managing the **TFJob** objects. This requires the user to both have access to the Kubernetes API Server as well as to be proficient enough to be able to create and manage such objects.

2.3.3 Kubeflow: Pipelines

A pipeline is a description of an ML workflow, including all of the components in the workflow and how they combine in the form of a graph. The pipeline includes the definition of the inputs (parameters) required to run the pipeline and the inputs and

outputs of each component. When a pipeline is run the system launches one or more Kubernetes Pods corresponding to the steps (components) in the workflow.

In terms of Kubernetes terminology a Pipeline is a *yaml* file that contains an Argo workflow[34]. In order for a Pipeline to run the user must *apply* their respective yaml file to a Kubernetes cluster.

But, because writing such yaml files can become cumbersome and difficult to maintain, the Kubeflow developers have developed an SDK as a Python Module to reduce that friction. With this SDK the Machine Learning Scientists can define the exact steps they want to be executed and the dependencies between the different steps. Then, the SDK will compile the code to an Argo yaml file which can be then submitted to the Kubernetes API. This SDK comes preinstalled with the Kubeflow Jupyter Notebook images. This way the Machine Learning Scientists can write their code in a Notebook launched from Kubeflow and then run parts of their code as a Pipeline in a distributed fashion.

Kubeflow also provides a Web UI to ease the management and life-cycle of Pipeline runs. With this UI users can create or delete Pipelines as well as check the results of older Pipeline runs and their artifacts and logs.

2.3.4 Kubeflow: Hyper Parameter Tuning

Kubeflow has a component for optimizing the hyperparameters of models, Katib[44] which is heavily inspired by Google Vizier[45]. The main idea behind it is to define the training procedure inside a container and run this logic multiple times in order to a satisfying set of hyperparameters. This is achieved by creating an *Experiments CustomResourceDefinition* and by requiring the code to be containerized in a specific way.

The containerized code will need to be able to run standalone. This means that the created container will need to be able to both access the data and train the model. The input in the code will be the hyper parameters, either in the *args* field or as environment variables, and will output in a result metric.

The containers structured in this manner can be run multiple times, even in parallel, with different hyperparameters as inputs in search of a set of them that will ac-

complish a specified performance quota. There are also multiple searching strategies for navigating through the hyperparameters space such as Cross-Validation, Random search, Bayesian optimization to name a few of them. The results from each run are also stored in a central database that is persisted with the use of *PersistentVolumes*. This is all orchestrated from the *Experiment CustomResource* Controller that is responsible for deploying Kubernetes Jobs for running the training code, keeping track of the performance of each run, applying the searching algorithm and finally deciding when to stop the optimization process.

2.4 Proposed Solution

In this thesis our goal will be to make it easy for the Data and Machine Learning Scientists to get to their development environment with a few clicks and without requiring deep knowledge of neither Kubernetes nor Containers. At the same time, we aim to incorporate the benefits of running in the cloud from inside the development environment of the users. Lastly, we will enhance Kubeflow's provided solutions to support the handling of in-cluster local storage and also provide users with UIs to seamlessly handle their data in any step of their workflows. This way Kubeflow will be able to provide a compelling and intuitive UX for running *data driver* collaborative workflows.

We will describe an end-to-end story and workflow for *Data Analysis, Transformation and Validation* as well as distributed *Model Training, Evaluation and Validation* from mostly inside a Jupyter Notebook, which is the most popular tool used from Data and ML Scientists. Also, all of the proposed steps will take place on the cloud. This way the Pipelines can scale up to the size of the underlying infrastructure. Lastly, this workflow will be presented with Kubeflow and Kubernetes in mind as the backing Machine Learning platform.

The workflow will start with the initial location of the datasets. We expect that there will be software support for having an Object Store that can hold data by storing diffs, as explained in the *Collaborative Workflows* section. This software will also be able to efficiently pull the data from such an object store to a *PersistentVolumeClaim*, which might have a previous version of the data or none whatsoever.

Workflow:

- **Log in to the ML Platform**: The end-user will login to the platform from their browser where they will be redirected to a central dashboard. From this central dashboard the user will navigate to the page for managing Jupyter Notebooks or Volumes in the cluster.

- **Import the Datasets**: There are two different ways for storing huge datasets for Machine Learning. The datasets could live on a Object Store from where the ML Scientists would connect to in order to fetch the data, or they could reside on the in-cluster local storage. In both cases the data will end up in *PersistentVolumeClaims* that will be able to be consumed from Pods which will be deploying the tools that ML Scientists use.

- **Create a new Notebook**: From the central dashboard the user will be able to connect to a UI that will allow them to launch a Jupyter Notebook. That UI will let them parameterize the resources the new Notebook will use as well as the underlying persistent storage, *PersistentVolumeClaims*, that will be backing it. The platform should have authorization mechanisms in place in order to prevent the user from creating resources if they don't have the necessary permissions. At this point the user is exposed in the environment he is comfortable and used to working with with just a few clicks and it will be hosted entirely on the cloud.

- **Experiment inside the Notebook**: Once inside the Jupyter Notebook the Data Scientists will either need to the necessary datasets from an Object Store or they could have direct access to the data, or to a copy of the data, through the mounted PVCs. In some vendor solutions, like Arrikto's Rok[1], the Notebook will have a button on its interface that will allow the user to check different snapshots of their data and pull this data inside their Notebook. Once they have the data available to them the Data Scientists can start working on cleaning and transforming the data and the Machine Learning Scientists on developing their models.

- **Commit experimentation work multiple times**: While the Data and ML Scientists work on the data and their model they will be working in an iterative manner. During this process they should be also able to take multiple snapshots both of their data as well as from their working environment[2].

[2]The working environment can be snapshotted in case it is persisted in PersistentVolumeClaims

- **Initiate distributed training from the Notebook**: After developing the core logic of their models, the ML Scientists will be able to initiate the training outside of their Notebook and still be able to monitor the progress of the training and it results. Kubeflow provides SDKs[2] for running a section of the code in the cluster. These can be combined with the data that exists in the PVCs, which could also be snapshotted in some vendor solutions[3], allows the issue of training jobs that can have the exact same environment with the development one.

2.5 Collaborations

During this thesis, we thoroughly collaborated with the developers and engineers of the Kubeflow project as well as with engineers from Arrikto. In the end, I was also acknowledged as a **member** of the Kubeflow Organization.

2.6 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.

[3] Arrikto's managed Kubeflow clusters

3 Background

In this chapter we provide the key theoretical elements for the understanding of our work. First, we explain several fundamental principles from the area of containers, distributed and storage systems.

3.1 OS-Level Virtualization & Containers

3.1.1 Overview

Operating System Level Virtualization is a technology in which a kernel allows multiple isolated user-space instances to co-exist. These instances, also known as Containers, look like real computers from the point of view of programs and processes that run inside them. Each Container shares the host's OS. This means that it uses the OS's normal system call interface and do not need to be subjected to emulation or be run in an intermediate virtual machine. This makes Containers very lightweight, since they require less overhead in order to be launched, in comparison to full virtualization technologies.

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.

3.1.2 Building Blocks

As mention earlier, Containers are essentially just a way of partitioning up a system into a number of sandboxed execution environments with their own resource limits, while all continue to share a single operating system kernel. This new virtualization technology is based on three fundamental kernel features; namespaces, cgroups and union filesystems.

- **Namespaces**: The Linux Namespaces are a kernel mechanism that, at a high level, allows for isolation of global system resources between independent processes. This mechanism does not restrict access to resources like CPU or disks. It achieves isolation by exposing a specific subset of them to the processes that run inside the namespace. Examples of these resources include PIDs, ipc, network, mount points, users and network.

- **CGroups**: CGroups is another Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes. With cgroups, the administrator can set limits to a set of processes as to how many resources they can consume.

- **Union Filesystems**: *Unification filesystems* allows files and directories of separate file systems, known as branches, to be transparently overlaid, forming a single coherent file system. Contents of directories which have the same path within the merged branches will be seen together in a single merged directory, filesystem.

Emphasis should be given to the fact that these features exist only in Linux-based kernels. This was a problem for Microsoft Windows' adoption of Containers, since Windows was not developed with the concept of containers in mind. Not only was it missing trivial features like *cgroups* and *namespaces*, but also Windows could only run in *one* of two modes: User Mode and Kernel Mode. Because of these limitations, Containers had a warm start in the Linux-based workloads.

But, Windows eventually caught up with the other platforms by implementing the necessary mechanisms. These where equivalent features to *cgroups*, *namespaces* as well as

a special *Host User Mode*, which runs core Windows services and the container management functions for the Container User Modes.

3.1.3 Docker

Docker is a set of PaaS[1] products that use operating-system-level virtualization to deliver software in packages. Docker used the building blocks mentioned above to create an interface on top to make it easier to manipulate and parameterize the lifetime of Containers.

3.1.3.1 Images

Docker introduced the concept of a *Container Image*. An image is a static representation that determines the execution of a Container. It has information on both the structure of the filesystem that will be used, as well as which processes will be started inside the Container. The Image is an *immutable* file which essentially is a snapshot of the Container.

The Image's filesystem is created by stacking up a list of read-only layers, by using a *union filesystem*. Then, when a container is instantiated from this Image, another thin writeable layer is added on top. This layer is also called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer.

Under the hood, an Image is a standard TAR that combines:

- **Rootfs (container root filesystem)**: A directory on the system that looks like the standard root (/) of the operating system. For example, a directory with */usr*, */var*, */home*, etc.
- **JSON file (container configuration)**: Specifies how to run the rootfs; for example, what *command* or *entrypoint* to run in the rootfs when the container starts; *environment variables* to set for the container; the container's working directory; and a few other settings.

[1] platform-as-a-service

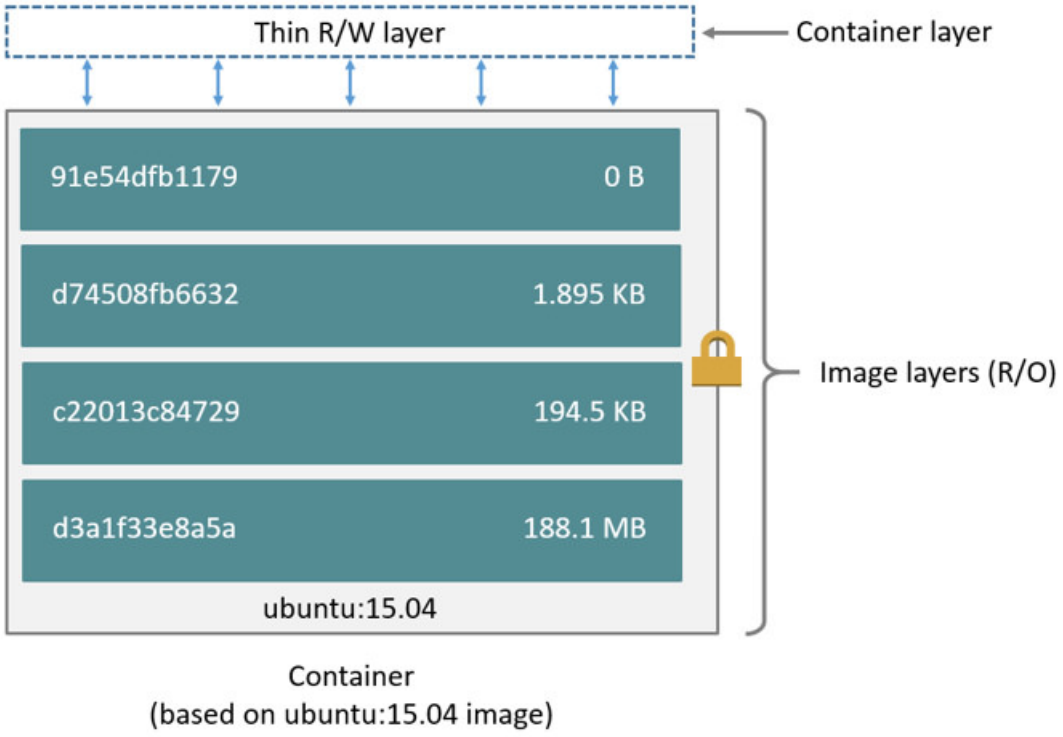

Figure 3.1: *Visualization of Docker Image Layers*

The definition of a Container Image was eventually standardized by the Open Container Initiative (OCI)[21] standards body as the OCI Image Specification[22].

Docker *TARs up* the rootfs and the JSON config and creates a *base Image*. From this Image, additional contents to the rootfs can be installed by creating a new JSON file and *tar* the difference between the original Image and the new Image with the updated JSON file. This will create a new *layered Image*.

3.1.3.2 Registries

Since Images are tar files, they can be shared and versioned. These files could be uploaded, have specific versions and downloaded from other users. Docker named the central places where images would be hosted *Registries* and implemented one themselves, DockerHub[20].

Registries, conceptually, are very similar with GitHub Repositories but for Images instead of code. Users can upload their Images, assign specific tags to versions of their Images and even have different branches for their Images.

3.2 Kubernetes

3.2.1 Overview

Kubernetes[4] is an Open Source system for or automating deployment, scaling, and management of containerized applications and services. It was originally created by Google, with version 1.0 launched in 2015. It orchestrates computing, networking and storage infrastructure on behalf of users' containerized workloads.

The conception of Kubernetes is due to the realization that the benefits of containerization go beyond the technical aspects of containers themselves as an operating-system level virtualization implementation. Containerization transforms the datacenter from being machine-oriented to being application-oriented, by abstracting away various details related to machines and operating systems from the application environment and the deployment infrastructure. It also shifts the management APIs from machine-oriented to application-oriented.

3.2.2 Architecture

The main philosophy behind Kubernetes is that you can decleratively define the desired state, and the system will consonantly monitor itself and strive to achieve this state.

The state is expressed as a set of yaml[26] Objects that are persisted in a distributed, hight-availability Key-Value Store *etcd*[27]. Then, there are the master components that watch this store for changes to relevant objects and try to update the status of the cluster towards the desired state. These components can access the state exclusively through a domain-specific REST API that applies higher-level versioning, validation, semantics, and policy, in support of a more diverse array of clients.

Essentially, master components provide the cluster's control plane. Master components make decisions about the cluster and they detect and respond to cluster events. Master components can be run on any machine in the cluster. However, for simplicity, set up scripts typically start all master components on the same machine, and do not run user containers on this machine. Because of this, we will group the components into two categories, based on the type of Node they are deployed in.

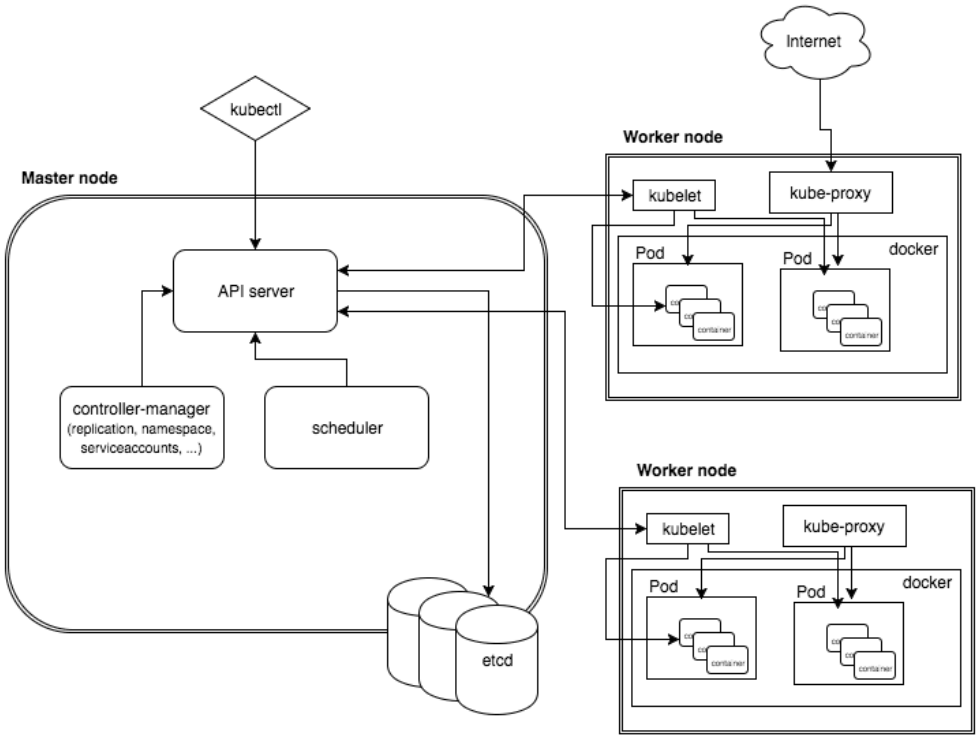

Figure 3.2: *Kubernetes Architecture Overview*

Master Components:

1. **etcd**: Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data. Even though Kubernetes uses a Master-Follower architecture which inherently represents a Single Point Of Failure (SPOF), it can achieve fault-tolerance by running multiple master nodes and using the distributed nature of etcd.

2. **kube-apiserver**: Exposes the Kubernetes API. It is the front-end for the Kubernetes control plane. It is designed to scale horizontally – that is, it scales by deploying more instances.

3. **kube-scheduler**: Watches for newly created Pods that have no Node assigned, and selects a Node for them to run on. Factors taken into account for scheduling decisions include individual and collective resource requirements, hardware/-software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference and deadlines.

4. **kube-controller-manager**: Component on the master that runs controllers. Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

Node Components:

1. **kubelet**: An agent that runs on each node in the cluster. It makes sure that containers are running in a pod. The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.

2. **kube-proxy**: kube-proxy is a network proxy that runs on each node in the cluster, implementing part of the Kubernetes Service concept. Kube-proxy also maintains network rules on nodes. These network rules allow network communication to the Pods from network sessions inside or outside of the cluster. It uses the operating system packet filtering layer if there is one and it's available. Otherwise, kube-proxy forwards the traffic itself.

3. **Container Runtime**: The container runtime is the software that is responsible for running containers. Kubernetes supports several container runtimes: Docker, containerd[23], cri-o[24], rktlet[25] and any implementation of the Kubernetes CRI[2].

3.2.3 Objects & Controllers

Kubernetes contains a number of abstractions that represent the state of the system: deployed containerized applications and workloads, their associated network and disk resources, and other information about what the cluster is doing. These abstractions are represented by objects in the Kubernetes API.

A Kubernetes object is a "record of intent" – once the user creates an object, the Kubernetes system will constantly work to ensure that object exists and has the desired status.

[2]Container Runtime Interface

3.2.3.1 Structure of Objects

Every object in Kubernetes will have some of the following fields; Spec, Status, Kind, apiVersion and Metadata.

- **Kind**: the kind of Object we want to create (e.g. Pod, Deployment, Service)
- **apiVersion**: specifies the version of that object's spec the following definition is fo's spec the following definition is for.
- **Metadata**: Data that helps uniquely identify the object, including a name string, UID, and optional namespace
- **Spec and Status**: Every Kubernetes object includes two nested object fields that govern the object's configuration: the object *spec* and the object *status*. The spec, which the user provides, describes the desired state and the *status* describes the *actual* state of the Object. At any given time, the Kubernetes Control Plane actively manages an object's actual state to match the desired state you supplied.

3.2.3.2 Controllers & Reconciliation Loop

Kubernetes Controllers are programs that continuously watch the API Server for changes in Objects and try to make the Status of that Object to match the Spec. This is happening inside a *Reconciliation Loop*.

3.2.4 Storage

3.2.4.1 Volumes

On-disk files in a Container are ephemeral, which presents some problems for non-trivial applications when running in Containers. First, when a Container crashes, kubelet will restart it, but the files will be lost - the Container starts with a clean state. Second, when running Containers together in a Pod it is often necessary to share files between those Containers. To tackle both of these problems, for Containers and Pods, Kubernetes has created the *Volume* abstraction.

In order for a Pod to use Volumes it will specify in its spec[3] which ones it wants to use as well as where to mount those into the Containers[4]. A process in a container sees a filesystem view composed from their Docker image and volumes. The Docker image is at the root of the filesystem hierarchy, and any volumes are mounted at the specified paths within the image.

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.

3.2.4.2 PersistentVolumes & PersistentVolumeClaims

Kubernetes provides the *PersistentVolume* subsystem via its API to users in order to abstract the details of how storage is provided from how it is consumed. For this it provides the *PersistentVolume* and *PersistentVolumeClaim* API Objects.

A *PersistentVolume* is an entity that represents a piece of storage in the cluster that has been provisioned, either statically from a user or dynamically. It is a resource in the cluster just like a node is a cluster resource. 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* is a request from a user to consume storage. Just like a Pod requests to consume a Node resource, a *PersistentVolumeClaim* requests to consume *PersistentVolume* resource.

3.3 Kubeflow

3.3.1 Overview

Kubeflow started as an open sourcing of the way Google ran TensorFlow[28] internally, based on a pipeline called TensorFlow Extended[29]. It began as just a simpler way to run TensorFlow jobs on Kubernetes, but has since expanded to be a multi-architecture,

[3].spec.volumes
[4].spec.containers.volumeMounts

multi-cloud framework for running entire machine learning pipelines. The project is dedicated to making deployments of machine learning (ML) workflows on Kubernetes simple, portable and scalable. Because ML practitioners use a diverse set of tools, one of the key goals is to customize the stack based on user requirements and let the system take care of the "boring stuff".

3.3.2 Architecture

Kubeflow aims to provide a cohesive experience for creating, managing and deploying Machine Learning Applications on Kubernetes. To achieve this, it provides a wide set of Machine Learning tools and pipeline orchestration software. These tools range from Jupyter Notebooks[30], Tensorboard[33] to multi-step Machine Learning Pipelines using Argo[34].

To deploy those tools in Kubernets, Kubeflow added one abstraction on top and created *Components*. The naming was a result of the underlying packaging software, *ksonnet*[35], which was used to bundle the necessary manifests together for each application. Most of the *Components* was exposing a Web UI for managing the underlying Kubernetes structures and providing the supported tools to the end users. These Web UIs would then be glued together via a central UI, later named the *central dashboard* which would allow the users to navigate to the supported UIs.

As of Kubeflow 0.4, the supported tools are the following

- **Jupyter Notebooks**: Kubeflow supports Notebooks by deploying and exposing a JupyterHub[31] service.
- **Distributed Training**: Distributed training is supported via a Kubernetes *CustomResourceDefinition*[32], the *TFJob Object* and *Operator*. Essentially, it is a wrapper around Tensorflow's distributed training.
- **Hyper Parameter Tuning**: Kubeflow has its own Kubernetes *Custom Resource Definition* for orchestrating running model trainers in parallel and for finding favorable hyper parameters.
- **Pipelines**: Kubeflow has created their own Machine Learning Pipelines which

are based on Argo. The users can define and run a DAG of Machine Learning steps.

3.3.3 Access Kubeflow

Kubeflow is a set of *CustomResourceDefinitions* and Web Applications for manipulating these as well as the Central Dashboard which links them all together to provide a cohesive experience. This includes the Jupyter Notebooks, Pipelines as well as TFJobs[5]. The user is expected to interact with Kubeflow via these UIs. As a result it is necessary to expose a subset of the Services that Kubeflow creates in the cluster in order for the user to be able to send traffic and communicate with the provided applications.

For up until version 0.5 Kubeflow was using Ambassador[36] to expose the different Services while from 0.6 and forward Kubeflow is using ISTIO[38]. In the first case, Ambassador will create some proxy Pods and expose them with a NodePort of LoadBalancer Service. The user will be accessing Kubeflow from this endpoint. Then, by setting specific annotations to the deployed Services in the cluster the traffic that will arrive at the Ambassador Pods will be redirected to the underlying application. in case of ISTIO an Ingress is created and the Kubernetes Services for the Applications will be exposed through this Ingress Object, via ISTIO custom Objects. Making the IP of the Kubeflow endpoint public, in both cases, will depend on the infrastructure (GCP, on-premises).

The final phase for accessing the Kubeflow artifacts, after connecting to the exposed IP, would be to authenticate and login. Authentication is provided from Kubeflow version 0.6 and forwards, which also uses ISTIO. The authentication method is once again dependent on the infrastructure in which Kubeflow is deployed. For GCP the authentication would happen via Google IAP[39]. For the on-premises case the Dex[40] identity service is used for handling the user login. Dex uses OpenID Connect[41] protocol for authentication and supports a wide range of *connectors* for working with different user databases such as LDAP[42].

After the user is authenticated the respective authenticator will need to set a header on the requests the will contain the information regarding the logged in user. ISTIO's

[5]The TFJob Management UI was removed from version 0.6 and forward

gateway will be checking for this header, which can be configured, and if it finds it in a request it will forward the request to its destination. Else, it will block the request since the user is not authenticated. ISTIO can also use this header to apply authorization such as to block requests on specific exposed Services, like other user's Jupyter Notebooks, that the logged in user is not authorized to view.

4 Design

4.1 Design Rationale & Goals

As noted in 2.1 Problem Statement, our goal is to improve the user experience around the Collaborative Workflows that Kubeflow offered. The main pain points we located where around deploying and managing Jupyter Notebooks, the tools and workflows for manipulating their data, as well as the visualization tools that would operate on that data.

More specifically, the issues with the current Collaborative Workflows are:

- **Notebooks**: The management of Notebooks was problematic due to JupyterHub which was adding significant friction. The Hub would try to centralize the handling of the Notebooks to itself and not use the functionality of Kubernetes. Also, there was no straight forward way to mount and use the cluster's storage directly from the Jupyter Notebooks, via *PersistentVolumeClaims*.
- **Data handling**: Kubeflow's support for disks, via Kubernetes' *PersistentVolumeClaims* was still in its early stages. Most of the data was stored in object stores and the Pods would have to constantly download it. This means that a lot of traffic would be generated for Workflows that require many people to work on and be able to use each others data. Also, there was no tooling included in Kubeflow for viewing and editing the files of *PersistentVolumeClaims*, making it even harder include them in the users' workflows.
- **Visualization Tools**: Kubeflow has support for Tensorboard for visualizing the

Tensorflow Graphs and metrics about the execution of a Machine Learning Model. But, again, there was no support for *PersistentVolumeClaims*, which means that Tensorboard could not work with data and metrics stored on local disks. Instead it would only work if the metrics would be uploaded in an Object Store, which the Pod running Tensorboard would need to be able to pull the data from.

In this thesis, we aim to solve these issues with a uniform method. We propose the use of Kubernetes' *CustomResourceDefinitions* for the tooling that Kubeflow provides, Web UIs for managing the tool instances and lastly support for *PersistentVolumeClaims*.

More specifically, we propose to

1. Create *CustomResourceDefinitions* for *Notebooks*, *Tensorboard* and *PVC Viewer* instances. The underlying controllers that would be responsible for these CRDs[1] would handle the management of Pods and Services that need to be created. This way, we shift the management responsibility of our resources from custom software to Kubernetes.

2. Provide thin Web UIs to allow the user to interact with the tooling. Under the hood, the UIs will be performing basic CRUD operations on the *CustomResourceDefinitions* that we created in the previous step. The UIs, combined with the respective Controllers will take care of the tedious details of deploying, launching and connecting with Machine Learning Tools on the cluster.

3. Integrate *PersistentVolumeClaims* with these *CRDs* and simplify their usage, in order to make them a part of the user's Workflows. This way using the result data of other people's work won't introduce extensive amounts unnecessary of network traffic to move around that data.

4.2 Proposed Workflow Implementations

In this section we will describe and analyze the workflows that will enhance the productivity of the Machine Learning Scientists. These workflows should capture the collaborative nature of the pipelines that Machine Learning Scientists use in order to de-

[1] Custom Resource Definitions

liver their products and artifacts. In the next session we will showcase the design decisions for implementing the described workflows on top of Kubernetes and integrate them with the Kubeflow platform.

4.2.1 Jupyter Notebooks

Since Jupyter Notebooks are one of the most popular and widely used tool from Machine Learning Scientists we want to make it as easy and fast as possible to manage them.

First off the orchestration of the Notebooks and all of the interactions with them, as instances inside the cluster, will happen via the Notebooks Management UI. After the user has successfully logged in to Kubeflow the Notebooks UI will show them the Notebooks that will be available for them in their own Namespace[2]. Access control will be enforced in the UI and it will only allow the users to select and modify Notebooks in the Namespaces in which they are authorized to.

After selecting the Namespace they want to work with the user will be exposed to a thin dashboard that depicts the state of the Notebook instances that exist in that particular Namespace. For each Notebook Server the user will be able to view the resource quotas, the Docker image as the mounted volumes. Also, for each one the user will be able to perform specific actions such as to delete the particular Notebook, to connect to it once it is successfully deployed, to stop or start it or to view its logs.

The UI will have a button that will redirect the user to a form to fill the necessary information to launch a new Notebook Server. The user will be able to configure the name, resource quotas such as CPU or RAM, the Docker image for the container, a list of volumes to attach to it as well as extra default configurations to be applied to the new Notebook Server. The cluster administrator will be able to set default values for the fields of the form as well as set some of the fields as read-only. This information will be mapped to a *Notebook CustomResource* that will be created.

The user will be able to connect to their Notebook servers with one click from the UI. After connecting to a newly created or an existing Notebook the user will be redirected in a new tab where they will be provided with the Jupyter Notebook environ-

[2]With 0.6.2 every user will have a Namespace only for them available

ment. From there the user will be able to fetch the necessary datasets or directly access them from the mounted PersistentVolumeClaims. With the data available they can start experimenting inside the Notebook and also take snapshots of the Notebooks environment, which is the home directory plus the mounted volumes, along the way[3].

Lastly they will be able to run specific sections of their code and run it in the cluster but with more resources available. Once the job has successfully finished they will be able to view the results directly from inside of their Notebook.

The goal of the mentioned workflow is for the Machine Learning Scientist to remain as much as possible in the environment that they feel comfortable in, the Jupyter Notebooks. The workflow aims to make it seamless to work on a Jupyter Notebook and utilize the cluster infrastructure but at the same time without getting in the way that the users are accustomed of doing their work.

4.2.2 Tensorboard & Visualization

One crucial functionality that most Machine Learning Scientists request is to be able to view and visualize their models live as they train. This is extremely helpful for gaining insights about their model, how it handles the data, possible pitfalls as well as its overall performance. Tensorboard is a Python package for visualizing the progress of a TensorFlow model. Since TensorFlow comes preinstalled with the Kubeflow Jupyter Notebook images, making it seamless to deploy Tensorboard for the model that is being trained inside a live Notebook is essential.

Firstly, just like with Jupyter Notebooks, there will be a Tensorboard Management UI that will allow the user to manage Tensorboard instances. This UI will also be enforcing control access and will be providing the user to select from a list of Namespaces that they are authorized to edit.

After selecting the Namespace, they will be able to view the Tensorboard instances that live inside of it. For each one they should be able to view the data source that the instance will be visualizing. Also, for each one the user will be able to perform specific actions such as to delete the particular Tensorboard instance or to connect to it once

[3]This feature will need additional software that will implement the actual snapshotting logic in the Kubernetes level

it is successfully deployed or to view its logs.

The UI will have a button that will redirect the user to a form to fill the necessary information to launch a new Tensorboard instance. The user will be able to configure name, and the data source for the new Tensorboard. It is crucial that for the data source the user will be able to select data from both *Object Stores* as well as from *PersistentVolumeClaims* in the selected Namespace. In conjunction with the Notebooks Management UI, the user will be able to deploy a Notebook and attach a small volume to it in which the model will be exporting logs while it is being trained. Then the user will be able to create a Tensorboard instance and select the mentioned disk, *PersistentVolumeClaim*, and visualize the progress of the model live while it is being trained from inside of the Notebook.

4.2.3 Volume Data Web-Editors

As described in the previous sections one of the main missions this thesis design is to integrate the use of Kubernetes *PersistentVolumeClaims* in the Machine Learning Scientists' workflows. This has been partially achieved by making it trivial to use PVCs from Notebooks and Tensorboard instances. But, the most important feature that is required to make the use of PVCs a part of such workflows is to have an intuitive way of viewing and editing the contents of them as well as orchestrating their their life-cycle.

As with the previous use cases the orchestration and management of the PVCs will be happening from a friendly UI, the Volumes Management UI. After the user has successfully logged in to Kubeflow, the Volumes UI will show them the PVCs that exist in the selected in the user's Namespace. Access control will be enforced in the UI and it will only allow the users to select and modify PVCs in the Namespaces in which they are authorized to.

After selecting the Namespace they want to work with the use will be exposed to a thin dashboard that depicts the state of the PVC instances that exist in the particular Namespace. For each PVC the user will be able to view its size, mode as well as the *StorageClass* that is backing it. The UI will also provide a form for creating a new PVC. Also, for each PVC the user will be able to delete it or browse and edit its contents. With these features the friction for working with PVCs will be minimal since with a

few click the user will be able to view, create, delete or even edit the contents of any PVC they have access to.

When the user will select to *browse* the contents of the PVC they will be redirected in a new tab. This tab will contain another web application that will be exposing the contents of the volume to the user. With this workflow the user will be able to manipulate the contents of the volume directly from their browser.

4.3 Data-Driven Computation Pods/CRDs

By the term *Data-Driven* we mean that the Pods, containing the tools, will need to be able to natively support local data. As a result, their scheduling will be affected by the location of the data in the cluster. They can not be arbitrarily created in any node. They are gravitated towards the data.

Ideally, the data could be versioned, snapshotted and stored in a central Object Store. Then, with proper software it could be moved between Nodes in order to have a more flexible scheduling of the Pods that will use the Data.

4.3.1 Specialized CRDs for ML Tasks

In this thesis, we will create a separate *CustomResourceDefinition* for each one of *Notebooks*, *Tensorboard* and *PVC Viewers*. These CRDs will need to encapsulate the necessary information and provide the user options to parameterize the software. In this section we will analyze what sub-resources will need to be deployed for each tool in order to be both customizable and easily accessible from the user.

All three CRDs will have a very similar structure. The only way to deploy an application in Kubernetes is inside a Pod. So all of the Controllers in the end will need to create a Pod. But, we want to leverage Kubernetes' advanced patterns for handling our *stateful Data-Driven* Pods and not have to manage and health check them ourselves. To achieve this, we will use one of Kubernetes' Objects the *StatefulSet*.

Next, we need to configure the traffic flow between our *CustomResource* and the world, both inside and outside of the cluster. For the inner-cluster traffic we will, once again,

rely on Kubernetes' Objects. We will create a *Service* which will handle the traffic and add another layer of abstraction on top of the Pods. Now in order for any program inside the cluster to communicate with our *CustomResource* it will only need to use the DNS name of the CR's *Service*.

At this point we have decided how to deploy and manage the actual Pods that will contain the tooling software, as well as how to get traffic to and from these Pods. One last decision we need to make is regarding how we are going to expose each Pod to the end-user, who is outside of the cluster. As stated in the *Access Kubeflow* section, Kubeflow had two ways of exposing its components. As with version 0.5 all the outer-cluster traffic was redirected to the corresponding component via Ambassador[36]. With 0.6 Ambassador was completely removed and replaced with *ISTIO* [38]. In the first case, special annotations will need to be added to the Resource's *Service*. In the second one, the Controllers will also need to create a *VirtualService* to expose the Service.

These three resources, *StatefulSet, Service* and *VirtualService* will need to be created from all three controllers. The main difference between the three will be the Pod's *Image, command, args* as well as some *annotations, labels* and maybe *ENV Vars*. Implementation details regarding the three CRDs will be given in the *Implementation* section later.

4.3.2 Support for PVCs

The *CustomResourceDefinitions* will move the management burden of our tools to Kubernetes. But, we these Resources and tools must not be stateless applications. We must make it non trivial to integrate the *PersistentVolumeClaims* with these *CRDs*. Since we have let kubernetes handle our instances, adding support for local data and disks is non trivial. We only need to add the desired *PersistentVolumeClaims* to the *PodTemplateSpec* that our *StatefulSet* will create.

But, having to manually edit the CRDs in order to use local disks is not a viable solution in terms of User Experience, which is what we want to improve. To integrate *PersistentVolumeClaims* to the user's Workflows we will need to abstract away the tedious details of including PVCs to PodTemplates. Instead, the UI and it's backend will take care of properly formatting the data in the CRs. Also, the UIs will need to make

the creation of new disks a non-trivial task for the end-user.

4.4 Start/Stop of Idle Pods and TTL

At his point we have abstracted the details of deploying and exposing our tools to the user with basic CRUD[4] operations on *CRDs*. But the current Experience is kind of binary. A resource can either exist in the cluster or not. But, it turns out that most of the time its more preferable to have the definition of the tool predefined and then just start or stop the live instances of that spec.

There are two reasons why this is the case. Firstly, we can cut down on cost since the life cycle of the resources that are being used can be more fine grained. Secondly, we can save a lot of time since the users won't have to create the same definitions over and over if they want to recreate an identical development environment. This is mostly the case with Notebooks which are resource intensive, especially the ones that require a GPU which is a valuable resource. Once created, then it will bind some of the system resources for as long as it exists. But, in case that the user is not actively working on it, then these resources will not be de-allocated from the system. This could also lead to higher costs if the infrastructure being provided as a Service. To tackle this the user can simply delete their Notebook when they are done with it, but he would have to re-create it every time they would like to come back to it, which also creates a negative User Experience.

In order to reduce this problem, we have come up with the following solutions:

1. **Start/Stop resources**: The user should be able to scale the deployed Pods for a Resource down to zero and back to one. With this behavior the definition and configuration of the tool will be persisted and the user can choose when to initialize it. This would require changes both to the Controller of the Resources as well as in the UIs.

2. **Stop Idle Pods(TTL)**: While giving the user the freedom to start and stop their resources, we would like to further improve the process by having the platform reconcile and stop idle tools. Of course the term idle must be parameterizable

[4]Create, Read, Update, Delete

depending on the tool as well as its details. For example an idle Notebook with GPU should have less idle time than a Notebook that is only using CPU.

The state of a resource that is stopped should be reflected in the Kubernetes Object so that the UIs can expose that state to the user. The controller must watch the resources for a special condition which will trigger the stopping of that resource. This condition will also be set or un-set from the UI in order to start/stop the resources.

4.5 Web UIs for Managing Computation Pods

With the implementation of the mentioned *CRDs* users can now create parameterized instances of their tools by applying YAML files to the cluster. While this workflow offers an abstraction on top of managing the Kubernetes Objects it imposes two main issues in terms of usability and User Experience. Firstly, the end-user must have access to the cluster that Kubeflow is deployed in order to be able to apply these YAMLs. Secondly and most importantly from our perspective, is the this offers a sub-par User Experience.

To counter these two issues, the need of management web applications becomes apparent. The aim of these applications will be to add another layer of abstraction on top of the implemented *CRDs*. The web applications will be the ones to deal with the inner details of the *CRDs* and the end-user will be exposed to parameterize only the important fields.

More specifically, these web applications should provide the following functionalities:

- **CRUD Operations on CRDs**: The end-user must have full control over the instances. The UI should allow them to list the existing instances, modify them if necessary, delete them and also to create new ones. Another necessary requirement would be to fully expose the state of the instances. This could be achieved by showing logs and Kubernetes Events.
- **Hide the Kubernetes details**: The user must not deal with any YAML files. The UI should ask the user to provide only the relevant information regarding the runtime of the tools. In this context, it should aim to hide as much of the underlying Kubernetes concepts as possible from the end-user.

In the next sections we will detail the design of the Frontend and Backend of these web applications.

4.5.1 Permissions and Authorization

Another very important aspect we need to address regarding the web apps is the way they handle user Authentication and Authorization. This will effect the instances that the UIs will have to show to different users, as well as the nature of the permissions the UI should use to CRUD instances. The answer to this depends, once again, on the version of Kubeflow.

Up until version 0.5, Kubeflow did not provide any form of user Authentication. In terms of the Web Apps this would mean that they would expose all the instances from all the Namespaces to everyone that accessed them. In conjunction, the web Apps would be able to create, delete and modify instances in any Namespace.

As of version 0.6, support for user Authentication was added in Kubeflow. The user would first have to login in a central place before accessing any of Kubeflow's resources. The login process depends on the underlying platform and infrastructure, for example *GCP*, *AWS*, *on-prem* to name a few. Every request that is destined for a path on the Kubeflow cluster will first pass from this central authentication point. After logging in, the identity of the user would be set in a configurable header to each request that will pass from this central authentication point. To complement the user Authentication a new component was added to Kubeflow, *Kubeflow Access Management*[37]. In this case, the web apps would have to first query kfam on every REST verb in order to ensure that the user is authorized to perform that action.

4.5.2 Frontend for Exposing the CRDs

To achieve the described behavior, the web applications will have three pages. A *main*, a *new* and a *logs* page. Before jumping on the design choices of each page, emphasis should be given to some common patterns that govern all of the pages.

The main one is the fact that any web app that is exposed with Kubeflow under the central dashboard must not determine itself which Kubernetes Namespace to use at

any point in time. This is because since authorization is in place, different users have different access levels. Thus users shouldn't be able to browse through the entire list of existing Namespaces in the cluster, but rather through the ones that they have permissions in. The central dashboard takes care of showing the users the list of Namespaces that they can see. Each other app will have to communicate with the dashboard to know which Namespace the user wants to use. More on how this is achieved on the 5.2 Management UIs section below.

Another architectural decision worth mentioning is how the frontend, the user's browser, can list and edit the resources in the cluster. The frontend does *not* communicate directly with the Kubernetes API Server at any point in time. In order to get any required information regarding the state of the cluster it will need to query its backend. The backend will be also exposed under the domain of the cluster, most frequently with a */api/* prefix. The backend is deployed inside the cluster and as a result any sensitive data or processing on that data happens inside the cluster, which is more secure than running in the user's browser.

The three pages that the web apps will expose are the following:

- **main**: This page is responsible for exposing to the user the state of the existing instances in the selected Namespace. It should show important fields in the *Spec* of the instances and their current status.

  From this page the user should also be able to perform necessary actions upon the available resources. These actions could be basic CRUD operations or more complex ones, like to connect to a Jupyter Notebook or stop a running one.

- **new**: The user will be redirected to this page once they want to create a new instance. The page should be responsible for exposing only the fields that have meaning in parameterizing, for the respective tool, and not the entire list of supported fields in the *CRDs' Spec*.

  This page can also drastically improves the User Experience if it would support the creation of other supplementary resources that might be needed from the new instance. For example, providing the user the ability to create and attach new *PersistentVolumeClaims* to their Notebooks lowers the barrier significantly for using local disks.

- **logs**: This page will be mostly used for debugging reasons. Its primary purpose is to provide logs and information of the instances. For this, it might even surface low level Kubernetes API concepts.

4.5.3 Stateless REST Backend APIs

The backend will be responsible for feeding the necessary information to the frontend. Also, any interaction with the instances in the cluster will happen through it. The backend will be accessible from outside the cluster for the frontend to use. With a Kubeflow version prior to 0.6 anyone that could access that endpoint would have full control over the *CustomResource* that this API handles. In 0.6 and afterwards, authentication and authorization was added to fine grain the permissions of users.

The backend will be a *stateless* REST API. Stateless because all the necessary information regarding the state is stored on *etcd*, the cluster's Object Store which serves as the single point of truth. So the backend will act as a layer on top of the Kubernetes API, but tailored for managing a specific resource. This simplifies the its structure by a great margin since most of the time it will only be validating, sanitizing and transforming the data it receives and then forwards it to the Kubernetes API. Being stateless also make it really easy to scale, since the cluster administrator can spin up more instances of the stateless backend.

An important feature in such web apps is to give the cluster administrators the freedom to control the fields that the web apps will expose to the users. More precisely the administrator might want the UI to have some specific default values for when the user tries to create a new instance. Also, they might even want to disable the editing of a specific field altogether. To accommodate those needs the web app will have a configuration YAML file that will dictate the default values of the form's fields as well as which fields should not be editable. This configuration will be deployed inside the cluster, so the end-users won't be able to mess around with it and only people with sufficient permissions will be able to do so. The edits of this config will be picked by the web app on the fly and won't need to restart the entire app.

The authentication and authorization checks, of Kubeflow 0.6 and up, happen in the backend. In order to use the web app the user must already be logged in to the cluster.

After being logged in then each following request will have a specific header set, non forge-able, which will contain the identity of the logged in user. The backend can check this header for the user identity and if not present then abort the request altogether with a respective error code. But, an interesting issue is how will the web app respect the authorization that is in place. More specifically, the problem is permissions should the web app use to make the final requests to the Kubernetes API Server. There are two options:

1. **Web app will have elevated permissions**: In this case the web app will have permissions to perform any actions on the specific resource in any Namespace. Thus, for any request to the Kubernetes API it will use these permissions. But this solution would render authentication and authorization useless, since anyone can again create the instances. To cope with this, the backend will first make a check if the user has access and permissions in the selected Namespace. It gets this information by querying *kfam* [5], which is also the service that the central dashboard queries to determine which Namespaces to show the logged in user. Then, if the user has permissions on this Namespace, then the web app will make the request with its own permissions. Else it will abort and the frontend will show an authorization error.

2. **Impersonate the logged in user**: In this case now the web app will have no permissions by itself. Instead, it will be making all the requests to the Kubernetes API Server by impersonating [18] the logged in user. This way the authorization would be handled by Kubernetes itself since the request will be essentially happening from the end-user. The issue with this implementation is that in case the web app malfunctions or get hijacked then it will be able to make requests to the API Server by Impersonating any user it is allowed to. In such case it could event have admin level permissions.

The main goal of both solutions is to reduce, as much as possible, the blast radius of a web app's malfunction. A first step for this would be to establish that it will have the least needed permissions. This way the worst possible scenario would be that the tool instances would be deleted from the cluster.

[5] Kubeflow Access Management component

Up until Kubeflow 0.6.2 the first solution was preferred, although discussions[19] were being made to figure out if it would make more sense to go with the second implementation.

5 Implementation

5.1 Software Stack

The proposed workflow in the 2.4 Proposed Solution section is a fairly complex one that has many moving parts. It requires code for Kubernetes Controller, frontend as well as backend implementations. In this section we will have a deep dive on the inner details regarding the different building blocks for creating such a Workflow.

5.1.1 Frontend Technologies

For the frontend part we had a great number of options. The first iteration that we published on Kubeflow was written in plain JavaScript and HTML. This was mainly to take advantage of already existing code that was adding extra functionality on JupyterHub for creating customized resources on Kubernetes. But, as the web app grew more in complexity and required more advanced features we decided that moving to a frontend framework would be a better solution in the long run.

In the end the frontend framework we went with was Angular[46]. The main reason for this was due to its modular nature as well as the way it handles a global state in the app with the use of Services. Not that it wouldn't be effective to write it in another framework, but I was mostly familiar with Angular at the time.

5.1.2 Backend Technologies

For the backend of the web apps we had again a lot of different options, from languages to frameworks. In the end we chose Python[47] as a language and the Flask[48] web framework. Once again being familiar enough with Python and Flask was the primary reason for going this road. Also, since the backend would only be stateless REST API we thought that a micro web framework would suffice. Up to this point Flask had sufficient enough mechanisms for implementing the features that we wanted.

5.1.3 Controller Libraries and Frameworks

Kubernetes and its ecosystem is written in Go[49], a programming language developed by Google. Because of this a lot of tools around Kubernetes have been build with Go. Kubernetes client libraries have been auto-generated for other languages like Python, Java and JavaScript but the main development of Kubernetes itself happens in Go.

In case of Controllers the Kubebuilder[50] project that provided an SDK for creating Kubernetes Controllers. It was mature enough, had support for multiple versions of the *CRD* and also could scaffold the boilerplate code so that we could only focus with the logic of the Controllers. This framework was used to create all of the Controllers for the mentioned *CRDs*.

5.2 Management UIs

5.2.1 Overview

A top-down view of the design of the web apps as well as the rationale behind the architectural decisions is detailed in the previous section, *Design*. In this section we aim to describe the low level implementation of theses choices and how they all glue together to provide a cohesive User Experience.

5.2.2 Build & Deploy

We will start by discussing how the frontend and the backend are build, how are they bundled together in a single web app and, in the end, how is this app deployed in the cluster.

The **frontend**, as already mentioned, is written in Angular. Angular has a specific way of organizing the TypeScript code into folders that correspond to Modules, Components and Services. To ease the developer of having to manually structure their source code the Angular community has created an NPM[1] package, *ng*, which provides functions for creating correctly structured boilerplate code which the user can the fill with their own logic.

But in the end, after we have finished with the source cod, most of the time some extra processing needs to take place. Specifically we want to deal with *transpiling, bundling, minifying* and *packaging*. Luckily for the developers the *ng* has a *build* command which applies these functionalities on the source code. The result is the parsed code translated into a *.html, .js, .css* and maybe some *.ttf* or other support files. These are the frontend artifacts that we want to bundle together with the backend.

The **backend** will be subjected to different process. All of the backend code will be a Python Module[51] which would encapsulate a Flask App. This app will be serving the REST API as well as the frontend's static files of the web app. These static file are the result of *ng build* as described above. So in order for the backend and the frontend to be bundled together as a single app it is required to take the frontend artifacts and add them to the backend Python Module.

But this web app will need to be deployed in Kubernetes so it must be containerized. To achieve this a *Dockerfile* is created which will automate the mentioned process. It will build the frontend code, bundle it together with the backend module and finally run the app so it can start serving requests. The Dockerfile for the web app for Jupyter Notebooks looks like this:

For the **Controllers** we will also be building Docker Images. The kubebuilder project that we used to build the controllers creates a Dockerfile alongside the boilerplate code, so we can use that to launch our controllers. We will also need to create yaml files

[1]Node Package Management

that will grant the controller *Pods* the necessary permissions to modify the specific resources.

```
# Stage 0: UI Build Stage
FROM node:10 as build-stage

WORKDIR /app

COPY ./frontend/package*.json /app/
RUN npm install

COPY ./frontend/ /app/

# Build the default and rok frontends
RUN npm run build frontend -- --output-path=./dist/out/default --configuration=production
RUN npm run build frontend -- --output-path=./dist/out/rok --configuration=rok-prod

# Stage 2: Backend code and UI serving
FROM ubuntu:18.04

RUN apt-get update -y && \
    apt-get install -y apt-utils  build-essential curl \
    python-dev python3-pip \
    libssl-dev libffi-dev python3-bcrypt

# We copy just the requirements.txt first to leverage Docker cache
COPY ./backend/requirements.txt /app/requirements.txt

RUN pip3 install -r /app/requirements.txt

# Backend code
COPY ./backend/kubeflow_jupyter /app/kubeflow_jupyter
COPY ./backend/main.py /app/main.py

# Frontend code
COPY --from=build-stage /app/dist/out/default /app/kubeflow_jupyter/default/static/
COPY --from=build-stage /app/dist/out/rok /app/kubeflow_jupyter/rok/static/

WORKDIR /app/

ENTRYPOINT ["python3"]
CMD ["main.py"]
```

Figure 5.1: *Jupyter WebApp Dockerfile. In this Dockerfile the web app actually contains two flavors. A default one which is used for vanilla Kubeflow and one for specialized vendor infrastructure, Arrikto's Rok in this case.*

The final Docker Image will be based on the ubuntu image and will only have the least required packages for running a Python Flask app. The size of the Image is around 600MBs, which is relatively lightweight. Now that the web app is encapsulated and modularized as a Docker Image it can be deployed on the Kubernetes cluster.

5.2.3 Frontend

Now that the building process of the web app is clear, we can move with the implementation details of the underlying parts. The frontend has been created with composabil-

ity in mind. By that we mean that it is broken down to smaller reusable components that could be assembled to create a frontend with custom logic. These components range from high level ones like the *main, new* and *logs* pages to the different input fields in the submit form.

In the Jupyter web app there are two flavors of the app that are being developed simultaneously. The default one which is using the basic components and a customized one that uses most of the default functionality but also introduces some more fields in the submit form.

When the *ng build* command is applied both of the flavors are compiled together in the source code. In order for the JavaScript to determine which flavor of the page it has to load it utilizes the Angular Environments[52]. There will be a variable in the Angular Environment that holds the information regarding the flavor of the UI. Then, each of the *main, new* and *logs* pages will be a component that checks this variable and then loads the respective code. The variable itself will be set when running the *ng build* or the *ng serve* commands. Also, when compiling the code the compiler will remove the unreachable code so the produced artifacts will not have bloated code.

At this point we can start discussing the details that each of the pages will require.

5.2.3.1 Main Page

As stated in the 4.5.2 Frontend for Exposing the CRDs the web apps must let the user to select the Kubernetes Namespace for which they want to manage the tools instances. But, they have to respect the existing authorization and not allow them to tinker with Namespaces that they don't have permissions to. In order to reduce code duplication every UI will be receiving the list of available Namepsaces directly from the central dashboard. This list will be complying with the existing authorization rules.

Once the user is logged in to Kubeflow they will be viewing the central dashboard for the entirety of the time. When they select to go to one of the underlying web apps what happens under the hood is that the central dashboard will load the web page requested inside an iframe[53]. Thus, the central dashboard will be emitting the value of the selected Namespace to the web apps via cross-document messaging[54]. In order to reduce the code duplication, since this logic will be required by Kubeflow web apps,

this functionality is packaged in a JavaScript library that the web apps can dynamically use.

Once the web app knows which Namespace the user wants to view it can then query the backend and get a list of the existing tools instances. But the web app will need to be able to respond to changes to the instances' state. This means that it can't just query the backend once but rather it needs to be able to continuously know the state of the instances. For this issue there are mainly two options

- **Polling**: In this case the frontend will be querying periodically the backend in order to get the latest state of the instances. One enhancement that could be applied is to not query with a fixed period but use Exponential Backoff Polling[55]. We could reduce the number of requests to the Kubernetes API Server in the backend even more if the backend would cache the results of the API Server and then make requests with a higher period.

- **WebSockets**: In case of WebSockets[56] the frontend would need to constantly have an open connection with the backend. The backend in turn would also need to have some form of caching which the websockets would be syncing with. While this way the frontend would immediately notice any difference in the state the implementation would be significantly more complex since both front and back-end would need to be stateful.

Up to Kubeflow version 0.6.2 our web apps use Exponential Backoff Polling.

With the mechanism for constantly having an up to date state in place the next decision to be taken in the frontend's main page is what information regarding the instances it should surface, as well as which actions to allow.

Each web app will be visualizing a different set of attributes to the user, depending on the underlying tool that gets deployed. But, the frontend should always reflect the state of the Pod that is responsible for deploying the tool. This status lives in the *status* field of each *CustomResource*. The UI will need to be able to distinguish between the different states and show a relevant icon as well as a basic informative message as a tooltip.

The actions the frontend will allow for each instance are the same for all of the web apps. The basic ones are the ability to *Create* and instance, *Delete* an existing one, *Start/Stop* idle instances and *Connect* to an instance in order to use the deployed tool.

5.2.3.2 New Instance Page

This page will be used to create new instances of a tool in the cluster. The user should be able to customize the necessary parameters that the tool can support. After the user has filled in their configuration then by clicking *Launch* the frontend will send a POST request to the backend. The backend will then process the input data, create any other Kubernetes objects if necessary, and then send a POST request to the Kubernetes API Server to create the new instance.

The frontend will need to have some default values for the fields that it exposes. Also, the cluster admin might not want users to be able to configure some of the tool parameters, for example CPU usage allocated for the tool instance. To cope with both of these needs the frontend will be querying the backend for a configuration that would control these two. What default values to put on the form and which of the fields the user should not be allowed to edit.

Another important aspect is how the page handles errors in the backend. These errors could range from problem with the Kubernetes API Server or even straight connection loss with the Backend. In these cases the web app will show a warning to the user. In case the error was caused in the backend then the backend will provide a helpful message explaining the problem. Else it will show a more generic error message. In either case, if this was caused when it tried to create a new instance then it won't try to recreate it. The user will need to click the *Launch* button once

5.2.3.3 Logs Page

This page's main goal is to provide all the necessary debug information regarding the life-cycle of the alive instances. When designing the User Experience of these web apps we decided that the end-user should be exposed as less as possible to any of the underlying system infrastructure. This would also include any information that Kubernetes itself would be exposing. With this rationale the main page only shows as little as an

icon and a sort message which comes from the *CustomResource*'s *status* field. But there are cases where this information might not be enough to get to the root of the problem. To tackle this the page will be exposing more messages, events and logs from the underlying infrastructure.

Under the hood the UI will have to tabs for an instance. One tab for *Events* and another one for *Logs*. More specifically:

- **Events**: This tab will be a list of all of the *Kubernetes Events* that each instance will be emitting. The nature of these events will be discussed in the following sections regarding the *CRD Controllers*.
- **Logs**: This tab will contain the logs of all the containers in the instance's Pod. These logs can be found via a GET to the Kubernetes API Server's Pod endpoint[2].

5.2.4 Backend

As noted in the 4.5.3 Stateless REST Backend APIs the backend will be a stateless REST API. Its primary goal is to add another layer on top of the Kubernetes API that will process the data in the request and then perform the necessary actions.

One of the first design choices we had to take was how would the responses be structured. Since most of the time the data that would be passed through would be JSON Objects the *Content-Type: application/json* header would be used. The next step would be to decide on the specific fields a response request would have. One of the main constraints was the fact that the backend might not be able to successfully complete the request's logic. This could be for a plethora of reasons like malformed data to a problem with the connection to the Kubernetes API Server that the backend needs to have. This created the need for a field in each request that the backend would have to send in order indicate to the frontend if it could successfully complete the needed task of the request, a *success* boolean field. When this field is set to *false* the backend will always set a string field, the *log*, which will contain a descriptive message of the error that the frontend can then show to the user. Also, each endpoint that will be returning a list of Kubernetes resources will be using a specific field to set this value like *Pods, PVCs, PodDefaults* etc.

[2] /api/v1/namespaces/<namespace>/pods/<name>/log?container=<container>

In order for the backend to be robust it will also need to be able to handle data that is not correctly formatted or even missing altogether. These can be handled by using boilerplate code for handling such cases and properly setting the *success* and *log* fields of the response respectively. Although there might be cases that the backend will not make the checks for the correctness of the data. This mostly happens with the fields that the backend might use to make requests to the Kubernetes API Server. In such cases the backend will simply format the request, without checking if the data is properly formatted, and it will depend on the API Server to handle the incorrect data. In case the API Server returns an error then the backend will again surface this information to the frontend.

The backend is also responsible for configuring the frontend's *new* page form fields. For this the backend will expose an endpoint that will return a JSON object describing the default values of each field in the form, as well as if it should be read-only or not. Also for robustness the frontend can handle missing data in that configuration object. But we want the administrator to be able to dynamically update the values of this configuration without having to re-deploy the application. The mentioned behavior can be achieved by creating a Kubernetes *ConfigMap* which will hold this configuration object. This *ConfigMap* can then be mounted to the Pods as a file in the container filesystem. The administrator can now edit this *ConfigMap* and Kubernetes will take care of updating the files in all the Pods that use it.

At last but not least the backend should also be able to support customized flavors with, if necessary, extra endpoints. This can be achieved by using Flask's Blueprints[57] feature. We will first create a *default app* that will have the basic set of endpoints. Then developers can create another Flask app that will inherit the default routes from this *default app* and then add any specialized endpoint.

5.2.5 User Identity & Permissions

While the ingress gateway in-front of Kubeflow is responsible for adding information to each request with the logged in user info it is up to the specific web app to use this information for authorization. For the web apps we discussed the backend is going to use the user identity information alongside *kfam* [3] in order to perform authorization

[3]Kubeflow Access Management API

checks. If the user is authorized then the web app will make the request using the credentials of its own privileged *ServiceAccount*.

More specifically at the start of each request, for any endpoint, the backend will run a function to check if the user identity is provided in the headers of the request. If its not set then it will set the *success* and *log* fields of the response with an unauthorized message. Else it will use this information and make a GET request to *kfam*. The request is essentially asking the Access Management API to give it a list of the users that have access to the Namespace the request was referring to. The API will the return a list of *bindings* that reflect the users that have access to the Namespace as well as the Kubernetes *RoleRef* that is giving them these permissions. With this list the backend will then check if the user identity of the request exists in that list and proceed or abort.

This implementation needs further improvements since at this point, version 0.6.2, the backends will not use the *RoleRef* information when deciding if the user is authorized or not. This means that a user might only be given viewing permissions in the Namespace but the web app will omit this altogether. It is being discussed weather the backends should keep on using *kfam* or make some form of impersonation to actually let Kubernetes itself handle the authorization permissions.

5.2.6 Jupyter Notebooks Management UI

Also referred as the *jupyter-web-app* it is the web app for managing Jupyter Notebook instances in the cluster. It's main goal will be to let users handle their Notebooks inside different Namespaces by taking care of the infrastructure details and allowing the users to customize the important parameters of their Notebooks.

For this the web app will be working with *Notebook CRs*. Under the hood for every CR there will be a *Pod* created in the cluster to launch the Jupyter Notebook's code. With this behavior the parametrization of Notebooks boils down to the parametrization of the *Pod* that will be created. This then raises the question of which fields of the *Pod* should the web app be exposing to the user. Since one of the main goals of the app was to hide as much as possible the Kubernetes concepts from the Data Scientists we decided not to expose the entire set of customizations that can be made on a *Pod Spec*. The web app would allow the user to edit:

```python
def is_authorized(user, namespace):
    '''
    Queries KFAM for whether the provided user has access
    to the specific namespace
    '''
    if user is None:
        # In case a user is not present, preserve the behavior from 0.5
        # Pass the authorization check and make the calls with the webapp's SA
        return True

    try:
        resp = requests.get("http://{}/kfam/v1/bindings?namespace={}".format(
            KFAM, namespace)
        )
    except Exception as e:
        logger.warning("Error talking to KFAM: {}".format(parse_error(e)))
        return False

    if resp.status_code == 200:
        # Iterate through the namespace's bindings and check for the user
        for binding in resp.json().get("bindings", []):
            if binding["user"]["name"] == user:
                return True

        return False
    else:
        logger.warning("{}: Error talking to KFAM!".format(resp.status_code))
        return False
```

Figure 5.2: *Authorization-checking code*

- **Name**: The name of the Notebook instance. This will be the name of the created *Notebook CR* as well as the name of the underlying *Pod* [4].

- **Image**: The Docker Image that the *Pod* will use. Kubeflow maintains a list of custom Jupyter Notebook Instances that can be used with the web app. They have a lot of commonly used python packages that are used from Data and Machine Learning Scientists pre-installed. They can also be run as is, meaning that they can be run locally from the laptop in order to provide the same development environment across locations.

  The web app also allows the user to select their own custom Docker Images. But the images will need to be configured to be able to work under a specific prefix and not always under root "/". For 0.7 one of the goals of the web app will be to support arbitrary jupyter images by configuring them on the fly to work under a specific prefix.

[4]Since the Pod will be created from a StatefulSet, the name of the pod will be <name>-0

- **CPU and RAM**: This will inform Kubernetes that the Notebook will be requesting at least this amount of CPU and RAM.

- **Workspace and Data Volumes**: Since we want to embed the use of local disks and data to the Data and Machine Learning Scientists we need to make it easy for them to use them with their tooling. For this reason the web app will allow the users to select or create a list of *PersistentVolumeClaims* that will be mounted to their Notebooks.

  In the Rok[5] flavor of the web app the user will also be able to select a snapshot of their data. The web app will communicate with the required software and create a *PersistentVolumeClaim* with the data of this snapshot. With this feature the ML Scientists can also have immutability and reproducibility as well as versioning of their data and work.

- **Configurations**: Since we don't allow the users to fully customize the spec of their Notebook *Pods* Kubeflow has created a mechanism for specifying special configurations that can be applied to the created Pods. For this we created a *PodDefaults CR*. These *CustomResources* describe what further parametrizations to make to pods that have some specific labels. These parametrizations could be the injection of specific credentials to the *Pod*, set environment variables etc. The web app will get a list of the *PodDefault* instances in the Namespace and allow the user to select which ones they want to apply to their Notebooks. Then it will set the labels for the selected *PodDefaults* so that the WebHook responsible for apply the configurations can detect it.

These will be the fields that the user will be able to configure and customize for a new Notebook Server. A subset of these will also be exposed to the user in the main page of the Notebooks Management UI.

5.2.7 Tensorboard Management UI

The Tensorboard web app is essentially a modified version of the Jupyter web app that works with *Viewer CRs* instead of *Notebook CRs*. The end result, again, will be a de-

[5]Vendor software provided by Arrikto Inc. that provides many storage functionalities such as dynamic provisioning of disks, snapshots etc.

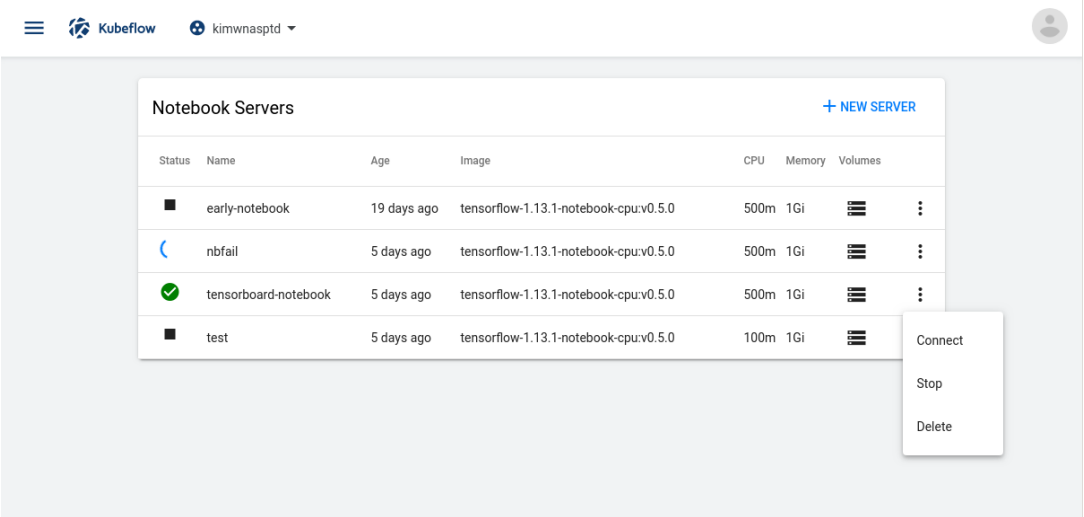

Figure 5.3: *Jupyter web app main page*

ployed *Pods* that will launch the code for Tensorboard. But, in this case the Tensorboard will need to have access to the data that it will visualize. So now the options to expose to the user are more limited. We mainly need to give them an easy way to specify where their data lives.

Before this web app Tensorboard was only launched through the Kubeflow Pipelines UI, but with some important limitations. The Pipelines UI would not be able to use *PersistentVolumeClaims* for the Tensorboard instances. Instead, the data had to live **only** in an Object Store that the *Pod* would have access to.

The Tensorboard web app will give the users two options, for now, to select their data from

- **Object Store**: In this case a link to the data will be provided such as *gs:, s3: etc*.
- **PersistentVolumeClaim**: The user will be able to select a PVC that holds the data they want to visualize. This behavior still needs some improvements as to which PVCs the web app should show to the user and how to check if the user should be authorized to mount this disk.

Once the user will specify their data the web app will create a *Viewer CR* that will in turn create the necessary Kubernetes Objects to instantiate it and make it accessible. The user will the be able to delete or connect to the Tensorboard instances they have created.

5.2.8 PVCs Management UI

This web app main goal is to smooth the experience of working with *PersistentVolumeClaims* on Kubernetes and Kubeflow. In contrast with the two previous web apps this one will actually manage two distinct Kubernetes Resources. The *PersistentVolumeClaims* and the *PVCViewers CRs*.

On the one hand with this web app the user will be able to manage the *PersistentVolumeClaims* in the Namespaces they have access to. This would include deleting and creating PVCs. Once these PVCs are created they can be used from the previous web apps. When creating new ones, the users will be able to configure its *size, StorageClass* and *Read-Write mode*.

This web app will also serve as a central implementation point that different storage vendors can include their own logic. For example, Arrikto's Rok software allows the user to have snapshots of their data. To support such features this web app will also have different flavors. In Rok's case when creating a PVC the user will have the basic options that Kubernetes has to customize but the users will also be able to easily create new PVCs with data from a snapshot they took with Rok.

But being able to create and delete *PersistentVolumeClaims* by itself is not that useful since there is still now way to view or modify the data that lives inside them. To improve this User Experience the web app will also handle *PVCViewer CRs*. These *CustomResources* have one field on their *spec* which spefifies the name of a PVCs. Then their controller will deploy a *Pod* with that hosts a web application[6] which provides a clean UI for editing the files in the filesystem of that app. This *Pod* will be mounted with the PVC that was specified. The Controller will also create a *Service* in order to expose this *Pod* from outside the cluster.

This way the user can edit the files of their PVCs without having to know any details of the underlying infrastructure. The user can even copy huge amounts of data via these *PVCViewer Pods*. Although the data rate will be limited by the network's bandwidth since the user will be uploading the data to the *PersistentVolumeClaims*.

[6]The image of the *PVCViewer Pods* is *coderaiser/cloudcmd* on DockerHub

5.3 TTL for Idle CRDs

The meaning of idle time can vary depending on the Kubernetes Object, or even between Objects of the same Kind. Also, some applications might already have their own definition of idle time embedded in the and expose that information. Also, the action that needs to be taken is, if he object is idle, is not the same between Objects. For example, we might want to *stop* a Notebook Server if it is idle but we want to *delete* a PVCViewer Pod that is not being used.

To be able to adjust to these needs we will need to have our culling logic to be modular and composable, in order to reduce code duplication, as well as to support a base case for determining if a resource is idle. Since the culling code will be used mostly from the Controllers of the *CRDs* we decided to implement it in Go as all of the them are also written in Go. It will be a go module[58] library.

Before diving into the interactions between this library and the Controllers we will describe how to determine if an Object is idle and how to mark it as idle. First of with Object, in this case, we mean the Kubernetes *Pod* where each tool instance lives. Also, the term idle will be from the prism of network traffic rather than CPU usage or some other metric. With this definition we have two options for checking if a Pod is idle:

- **Traffic Statistics**: Firstly by storing statistics of the *Pod*'s traffic in a central place and periodically checking if it hasn't received traffic up to a threshold time.
- **Specific Tool Endpoints**: Secondly, some tools like Jupyter Notebooks[7] expose this information themselves so we can simply query the instance and determine if it is idle based on that information.

Since version 0.6 Kubeflow has a hard dependency on ISTIO. ISTIO will be attaching an init and a sidecar container to all the *Pod* in the cluster. The init container will change the network traffic rules for the Pod and forward all the traffic first to the sidecar container and then to the actual destination. These sidecars will also be sending their requests metrics to a Prometheus[59] database.

[7]Each Notebook Server has an */api/status* endpoint that exposes its idle time in terms of requests and kernel activity

These two methods will be used to determine if a *Pod* is idle, but we also need to depict that information on the *CustomResources* responsible for it. There are again two options for this, either have a special field on each *CustomResourceDefinition* or set a specific annotation. In the end chose the annotation way since it can be used on any Kubernetes Object without modifications to the existing code-base. This annotation will have a key *kubeflow-resource-stopped* and as value the timestamp of when the resource was marked as idle.

5.3.1 Culling Library

This library will aim to decouple the culling logic from the controllers and centralize it in one place. All the logic for checking if an instance is idle as well as how to set the annotations will be a part of this library. This way the controllers will only need to use the functions of this library to check if an instance is idle and then act based on this information.

The library will have all the possible methods for checking if an instance is idle. This includes the default procedure of querying ISTIO's Prometheus database as well as specialized implementations like Jupyter Notebooks. In case the library could not determine if an instance is idle or not, due to communication problems etc, it will always report that the instance should not be culled.

The library should also expose some environment variables that will configure the culling behavior, like default idle time and checking period. The list of supported configuration environment variables is the following:

- **IDLE_TIME**: The time threshold after which an instance will be marked as idle. If not specified this value will default to one day.
- **CULLING_CHECK_PERIOD**: The time interval that the controller will be checking the status of each instance to determine if they are idle. If not specified this value will default to one minute.
- **ENABLE_CULLING**: This variable will dictate if the Controller should cull the idle resources. If not specified it will default to FALSE.

5.3.2 Culling of Notebooks

There are two phases for the culling of Notebooks *CustomResources*, how to determine if an instance is idle and what action to take if it is.

- **Idle Check**: Each Notebook Pod will be a server which the user connects to in order to edit and run their Python code. Each of these Notebook Servers exposes a REST API, with a */api/status* [60] endpoint which exposes information regarding the idle status of the Notebook. This endpoint takes into consideration both the network traffic on this REST API as well as the server's Python kernel activity.

  If the Notebook is idle, then the Controller will add the culling annotation the that Notebook *CustomResource*.

- **Culling**: With Notebooks, culling means that the Notebook CR should still exist but it shouldn't have a live Pod to back it up. Once a Notebook is marked as idle with the culling annotation, the Controller will then update the definition of the underlying *StatefulSet* to have zero replicas.

5.3.3 Culling of Tensorboards, PVCVIewers

- **Idle Check**: These two resources will use the default checking method based on the metrics that are saved on Prometheus.

- **Culling**: In this case, if an instance of these *CustomResources* is marked as idle then the *CustomResource* will be deleted altogether.

5.4 Custom Resource Controllers

All of the Controllers for *Jupyter Notebooks, Tensorboards* and *PVC Viewers* are written in Go. Notebooks and PVCViewers specifically are written with the kubebuilder project. While each provide a unique functionality and User Experience, they all follow the same pattern. They have a reconciliation loop watching for changes on a *CustomResource* and will keep up to date a Pod, a Service and a VirtualService for each

instance. The main differences between them are the Pods that will be created for each *CustomResource*.

The Services and the VirtualServices that will be created for each Notebook, Tensorboard or PVC Viewer will be almost identical. Their only difference will be the in-cluster endpoint in which they will forward the traffic.

The Controllers will also be responsible for periodically checking for idle resources and stop or delete the idle ones. This will be part of the reconciliation loop and the culling logic will be imported and used from the culling library we discussed in the previous chapter.

The main points we need to address for each controller are the *CustomResourceDefinition*, which is the API Spec of the new Object we want to create, and the logic that takes place inside the reconciliation loop.

5.4.1 Jupyter Notebooks

This controller will be responsible for creating and configuring the resources needed for seamlessly launching and accessing Jupyter Notebooks on a Kubernetes cluster. For this, a *Notebook CustomResourceDefinition* is created for managing Jupyter Notebooks as first class Kubernetes citizens.

5.4.1.1 Notebook Spec

The Notebook Spec has only a *template* field which is a *PodSpec*. This was because we wanted to give full control of all the Pod fields when creating a Notebook. Since this controller is just an abstraction on top of Pods and doesn't have any Notebook specific logic, except from the culling, all the weight for ensuring a smooth User Experience is lifted from the Notebooks web app which abstract away most of the Pod's details that a user might not need to configure.

Each Notebook CR will also have a status field that will be providing information regarding the state of the Notebook in the cluster. This field will have a list of Kubernetes Pod Conditions, number of created replicas for the Notebook and a ContainerState.

5.4.1.2 Notebooks Reconciliation Loop

The Notebooks Controller is essentially an abstraction on top of Pods. The Controller will be responsible for creating a Pod using the PodSpec from the Notebook Spec as well as a Service and VirtualService for making each Notebook accessible from outside and inside the cluster. A Jupyter Notebook most of the time will be a stateful application since it will be having disks to back up its underlying filesystem. For this reason, the Notebooks Controller will not be managing the Pod directly but it will be deploying a StatefulSet using the PodSpec in the spec of the Notebook CR. Despite keeping up to date the child resources for the *Notebook CustomResource* the reconciliation loop will be responsible for updating the status on the CR as well as to emit important events.

Regarding the status of the *CustomResource* the Controller will be updating it only when the underlying Pod's definition will be created on the API Server. The ContainerState will be the ContainerStatus of the first container in the Pod and the conditions will mirror the Pod's conditions. One issue with this behavior is the when the StatefulSet can not create the requested Pod. In that case the status of the Notebook is never updated and as a result the UI won't be able to surface to the user any error message. Also the StatefulSet, in contrast with a Deployment, does not depict the status of the underlying Pods in its own status. So in order to expose information regarding the StatefulSet the Controller will have to use the Events that are emitted from the Notebooks Stateful set and either update the status of the CR based on that information or promote the Events as Notebook Events.

Each Notebook Server will be exposed under a unique URL for the user to connect to. This URL for each Notebook will be */notebook/<namespace>/<notebook-name>*. The endpoint will be created from the VirtualService of each CR and the traffic will end up to the Pod via the created Service. The requests that will end up to the backend of the Notebook Servers will have the mentioned prefix. But this would be a problem since by default the backend will only respond to traffic that is destined for root /. The Controller will need to configure each Pod to be able to receive and respond to traffic that is for another URL than root. Jupyter Notebooks have a *–NotebookApp.base_url* flag for configuring the base URL of the Notebook Server. The Kubeflow Jupyter Notebook docker images set this flag with the value of the *NB_PREFIX* ENV Variable of the Pod which will have a default value of /. The Controller will also be setting this

ENV Var all of the Notebook Pods with the value of the prefix that the Notebook will be exposed under. This way if someone would like to create their own Jupyter Notebook images they could make them work with Kubeflow by utilizing this variable.

For kubeflow 0.7 the Controller will be checking for the Events of the StatefulSets and simply echoing them as Notebook Events. These Events will be then shown to the user in the *logs* page of the Notebooks Management UI. An admin who will have access to the Kubernetes cluster can now also run a *kubectl describe notebooks <notebook-name* in order to debug problems that might arise when deploying Notebooks on the cluster. Also in 0.7 the Controller was migrated to use *kubebuilder v2* in order to support multiple versions of Notebook *CustomResource* to be deployed in the cluster at the same time

5.4.2 Metric Viewers and Tensorboard

This controller will be responsible for creating and configuring the resources needed for seamlessly launching a Tensorboard environment on a Kubernetes cluster for visualizing metrics. For this, a *Viewer CustomResourceDefinition* is created for managing Tensorboard instances as first class Kubernetes citizens.

5.4.2.1 Viewer Spec

Up to version 0.7 the Viewer CRD only supported the visualization of data that would live in GCS Buckets[8]. Because of this the spec only had one field, called *log* that would contain the URL of the GCS folder. This information would then be used in the container's *entrypoint* when launching Tensorboard inside a Pod.

Our proposed implementation is to add the option for the Viewer CRD to natively support PersistentVolumeClaims. For this we will need to extend the Viewer's spec and add a *pvc-name* field. Then the Controller could use this new information and launch Tensorboard on the directory where the data inside the disks live.

Last but not least, this Controller will need to expose some status of the created resources on the status field of each live *CustomResource*.

[8]Google Cloud Storage

5.4.2.2 Viewers Reconciliation Loop

The Viewers Controller is still in alpha version. Just like the Notebooks Controller it will be creating a Deployment and a Service. This implementation is sufficient with the Pipelines UI, as we will discuss in the next paragraph, but imposes some serious limitations regarding the use cases in which it can be leveraged.

Up until version 0.7 this *CustomResource* was only used from the *Pipelines* UI. More specifically the user would state in one of the Pipeline's step that Tensorboard should be used with a specific GCS Bucket as input. Then the Pipelines UI would create a Viewer CR with the respective *log* field and then proxy any traffic directly to the created Pod via its own backend Server. Since this was the only way they had designed to create and use Viewer CRs the Controller would not need to create a *VirtualService* for making the Pod accessible from outside the cluster. But this would limit the use of Tensorboard only from inside the Pipelines UI. Even if the user would create a Viewer CR themselves and POST it to the API Server they wouldn't be able to access it from outside the cluster. To tackle with this it is evident that the Controller will need to also create a VirtualService for each CR.

The next step we will make in this implementation will be to add support for PersistentVolumeClaims on the Tensorboard Pods. As described in the *Viewer Spec* section the first step will be to add the *pvc-name* field on the Viewer CR's spec. Then the controller will have to use these fields accordingly. First of, if the *pvc-name* field is set, then the Controller will attach this PersistentVolumeClaim on the definition of Tensorboard's Deployment. Next the Controller will need to modify the entrypoint of the Tensorboard Container in order to load the data from the attached PersistentVolumeClaim.

This implementation seems viable on a first glance but there is another serious issue lying in Kubernetes' architectural decisions. The problem lies on the fact that if the requested PersistentVolumeClaim had *ReadWriteOnce* Access Mode and is already used by another Pod, then the scheduler will schedule the Tensorboard Pod and it will wait for the other Pod to terminate. Note that this problem won't exist if the PVC has *ReadWriteMany* Access Mode, but in most cases it will have *ReadWriteOnce*. In order to deal with the bad User Experience that would follow from this pattern we will take advantage of the *ReadWriteOnce*'s definition[61]. More specifically we *can* have different

Pods to use the same *ReadWriteOnce* PVC *if* these Pods will be scheduled on the same Node. The PodSpec has a *nodeName* field which will request from the scheduler to launch the Pod on the requested Node. To wrap up, on the reconciliation loop if the Pod for the CR is not created then the Controller will:

1. Get a list of all the Pods in the Namespace of the new Viewer CR
2. Check if there is a Pod that uses the requested PVC and that this Pod running
3. Get the Node name from the Pod that is already using the PVC
4. Set the nodeName field on the Tensorboard deployment with the name from the previous step

The Tensorboard Pod will have minimal CPU and RAM requirements so it should be able to launch on the same Node most of the times. This way the users will be able to have a Notebook running that will be updating the logs in a PersistentVolumeClaim and at the same time be able to view live the metrics with Tensorboard.

The last changes we will need to make to the Controller will be regarding the status of the Viewer CRs. Up to version 0.7 of Kubeflow the status of the Viewer CR would not be updated from the underlying resources. As a first step we will mimic the behavior of the Notebook Controller and will expose the state of the Tensorboard's Container to the status field of the CR as well as Emit the Deployment's Events as Viewer Events.

5.4.3 PVC Viewers

The goal of this *CustomResource* will be to make it easier for the user to modify and view the files inside their PersistentVolumeClaims. This will require a Pod with the desired PVC attached to it and a web app inside it that will provide a UI for editing the files inside that Container's filesystem. This web app will be the Cloud Commander[62] which have a very intuitive User Interface.

5.4.3.1 PVCViewers Spec

The PVCViewer spec has only one field, *pvc*. This field denotes the PVC that the user wants to edit its files and will be used by the reconciliation loop in order to mount the

requested PersistentVolumeClaim and properly use it with the right Container entrypoint.

The PVCViewer's status will have the same update logic with Notebooks and Viewers Controllers. The state of the Cloud Commander container will be exposed to the CR once the Pod is created and also the Controller will be echoing the Events of the underlying resources as PVCViewer Events.

5.4.3.2 PVCViewers Reconciliation Loop

Once again the implementation of the PVCViewers Controller will be very similar with the Notebooks Controller. This time, thought, the Controller will be creating Deployments instead of StatefulSets. It will be also creating a Service and VirtualService for proper traffic routing, both in-cluster and outer-cluster.

One important aspect of the implementation is how it will mount the PersistentVolumeClaim that the user has requested. The biggest obstacle in terms of User Experience will be the use of PVCs that have *ReadWriteOnce* Access Mode and are already used by another Pod, which will be the case most of the time. To tackle this and allow the users to view and edit the data of the PersistentVolumeClaim live even when it is being used from another Pod the Controller will influence the scheduling of the Pod as described in the Viewers CR reconciliation loop section.

Once the PVC is mounted on the Pod and the Pod can be successfully scheduled the next step is to configure the Cloud Commander web app to properly use the mounted disk. For this the Controller would need to pass the directory of the volume to the container's entrypoint. Also, just like the other Controllers, the Cloud Commander would need to be configured to work under a prefix which in this case would be */volumes/devices/<namespace>/<pvc-name>*. This would be the endpoint that the VirtualService will be exposing for each PVCViewer. The Cloud Commander web app has a *–prefix* and *–prefix-socket* option for configuring it to respond to traffic in a different endpoint than root.

Lastly, again just like the Notebooks and Viewers Controller, this controller will be using the ContainerState of the Cloud Commander container to set the status of the PVCViewer CR. Also it will be emitting all of the events from the created Deployment

as PVCViewer Events.

```yaml
apiVersion: kubeflow.org/v1alpha1
kind: Notebook
metadata:
  annotations:
    kubeflow-resource-stopped: "2019-09-03T16:30:59Z"
  creationTimestamp: "2019-08-25T10:23:11Z"
  generation: 3
  labels:
    app: tensorboard-notebook
  name: tensorboard-notebook
  namespace: kimwnasptd
  resourceVersion: "13430115"
  selfLink: /apis/kubeflow.org/v1alpha1/namespaces/kimwnasptd/notebooks/tensorboard-notebook
  uid: 56ea5996-c722-11e9-9068-42010a9c01ac
spec:
  template:
    spec:
      containers:
      - env:
        - name: NB_PREFIX
          value: /notebook/kimwnasptd/tensorboard-notebook
        image: gcr.io/kubeflow-images-public/tensorflow-1.13.1-notebook-cpu:v0.5.0
        imagePullPolicy: IfNotPresent
        name: tensorboard-notebook
        ports:
        - containerPort: 8888
          name: notebook-port
          protocol: TCP
        resources:
          requests:
            cpu: 500m
            memory: 1Gi
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
        volumeMounts:
        - mountPath: /home/jovyan
          name: workspace-tensorboard-notebook
        - mountPath: /dev/shm
          name: dshm
        workingDir: /home/jovyan
      serviceAccountName: default-editor
      volumes:
      - name: workspace-tensorboard-notebook
        persistentVolumeClaim:
          claimName: workspace-tensorboard-notebook
      - emptyDir:
          medium: Memory
        name: dshm
status:
  conditions:
  - lastProbeTime: "2019-09-03T16:31:05Z"
    type: Terminated
  containerState:
    terminated:
      exitCode: 0
      finishedAt: null
      startedAt: null
  readyReplicas: 0
```

Figure 5.4: *Stopped Notebook CustomResource Object*

6 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.

6.1 Concluding Remarks

All in all, the primary goal of this thesis was to implement a design that would improve the User Experience from the workflows that Kubeflow supported by taking under consideration their collaborative nature as well as the gravity of the data. 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.

More specifically, regarding the main goal, the described workflow implementations were met with positive feedback from both the Kubeflow community as well as end users. The Jupyter Notebooks workflow became part of Kubeflow as of version 0.5 and Tensorboard is set to be included for 0.8. As a proof of concept for our implemented rationale, a number of organizations are using the workflow we implemented, such as Cisco, IBM and Google, as well as private users. In addition, more and more people get to use it day by day.

As far as the second goal is concerned, there where times that, mostly due to my inexperience, felt really stressful. But, with constant effort and communication we managed to deliver a competent end result and rose to the expectations.

6.2 Future Work

Although we implemented a couple of enhancements on Kubeflow's UX with the introduction of Management UIs, *CustomResourceDefinitions* and Controllers there is still room for development.

First of all since this is an iterative process we will need to constantly observe and communicate with the end users to make sure that our suggested workflows cover their needs. Also, while the Management UIs allow the users to parameterize a subset of the possible fields of the underlying Objects more advanced use cases will require the UIs to expose more and more options. While this has not been the case up to now, we will need to provide support and further development when such needs become a necessity.

Other possible improvements, in terms of PVCs support, would be in the Model Serving component of Kubeflow, KFServing[63]. KFServing is used for inference, which is to expose the trained model to new data and return back the prediction. But, currently KFServing has no support for *PersistentVolumeClaims* and can only load data from Object Stores.

Bibliography & References

[1] CloudNative Whitepaper, *Arrikto Rok*, https://www.arrikto.com/wp-content/uploads/2018/10/20180206-2-CloudNative-Whitepaper.pdf, accessed on the November 10, 2018.

[2] Software Development Kit, *Wikipedia*, https://en.wikipedia.org/wiki/Software_development_kit, accessed on the November 10, 2018.

[3] *Open-Source Software* https://en.wikipedia.org/wiki/Open-source_software, accessed on the November 12, 2018.

[4] Production-Grade Container Orchestration, *Kubernetes*, https://kubernetes.io/, accessed on the November 12, 2018.

[5] *Kubeflow*, https://www.kubeflow.org/, accessed on the November 18, 2018.

[6] Kubeflow, *v0.4*, https://v0-4.kubeflow.org/, accessed on the November 18, 2018.

[7] Data wrangling, *Wikipedia - The Free Encyclopedia*, https://en.wikipedia.org/wiki/Data_wrangling, accessed on the November 21, 2018.

[8] Data warehousing, *Wikipedia - The Free Encyclopedia*, https://en.wikipedia.org/wiki/Data_warehouse, accessed on the November 21, 2018.

[9] Data integration, *Wikipedia - The Free Encyclopedia*, https://en.wikipedia.org/wiki/Data_integration, accessed on the November 21, 2018.

[10] Data management, *Wikipedia - The Free Encyclopedia*, https://en.wikipedia.org/wiki/Data_management, accessed on the November 21, 2018.

[11] Data Science, *Wikipedia - The Free Encyclopedia*, https://en.wikipedia.org/wiki/Data_science, accessed on the November 21, 2018.

[12] Data set, *Wikipedia - The Free Encyclopedia*, https://en.wikipedia.org/wiki/Data_set, accessed on the November 21, 2018.

[13] Web Services Architecture, *REST API*, https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#relwwwrest, accessed on the December 2, 2018.

[14] Object-Based Storage, *IEEE*, https://ieeexplore.ieee.org/document/1222722, accessed on the December 3, 2018.

[15] *Arrikto*, https://www.arrikto.com/, accessed on the December 3, 2018.

[16] Jupyter Notebook, *Extensions*, https://jupyter-contrib-nbextensions.readthedocs.io/en/latest/, accessed on the December 8, 2018.

[17] *KubeSpawner* https://github.com/jupyterhub/kubespawner, accessed on the December 8, 2018.

[18] Impersonation, *Authenticating - Kubernetes*, https://kubernetes.io/docs/reference/access-authn-authz/authentication/#user-impersonation, accessed on the December 10, 2018.

[19] GitHub - Kubeflow, *Notebook Spawner backend should impersonate as k8s user account*, https://github.com/kubeflow/kubeflow/issues/2981, accessed on the April 10, 2019.

[20] *Docker Hub*, https://hub.docker.com/, accessed on the December 8, 2018.

[21] *Open Container Initiative*, https://www.opencontainers.org/, accessed on the December 8, 2018.

[22] *Open Container Initiative, spec*, https://github.com/opencontainers/image-spec/blob/master/spec.md, accessed on the December 8, 2018.

[23] *Containerd*, https://containerd.io/, accessed on the December 8, 2018.

[24] *CRI-o*, https://cri-o.io/, accessed on the December 8, 2018.

[25] *rktlet*, https://github.com/kubernetes-incubator/rktlet, accessed on the December 8, 2018.

[26] *YAML*, https://yaml.org/, accessed on the December 2, 2018.

[27] *A Reliable Key-Value Store*, https://etcd.io/, accessed on the November 4, 2018.

[28] *TensorFlow*, https://www.tensorflow.org/, accessed on the December 10, 2018.

[29] *TFX: A TensorFlow-Based Production-Scale Machine Learning Algorithm*, http://stevenwhang.com/tfx_paper.pdf, accessed on the December 10, 2018.

[30] *Jupyter Notebooks* https://jupyter.org/, accessed on the November 21, 2018

[31] *JupyterHub* https://jupyter.org/hub, accessed on the November 21, 2018

[32] *Custom Resource Definition | Kubernetes* https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/, accessed on the November 21, 2018

[33] *Tensorboard* https://www.tensorflow.org/guide/summaries_and_tensorboard, accessed on the December 10, 2018

[34] Get stuff done with Kubernetes, *Argo* https://argoproj.github.io/, accessed on the January 9, 2018

[35] *ksonnet* https://ksonnet.io/, accessed on the November 21, 2018

[36] *Ambassador* https://www.getambassador.io, accessed on the November 23, 2018

[37] Kubeflow Access Management, *kfam* https://github.com/kubeflow/kubeflow/tree/master/components/access-management, accessed on the June 23, 2019

[38] *Istio* https://istio.io/, accessed on the June 23, 2019

[39] Identity-Aware Proxy, *Google* https://cloud.google.com/iap/, accessed on the June 28, 2019

[40] *Dex* https://github.com/dexidp/dex, accessed on the July 2, 2019

[41] *OpenID Connect* https://openid.net/connect/, accessed on the July 2, 2019

[42] Lightweight Directory Access Protocol, *LDAP* https://ldap.com/, accessed on the July 4, 2019

[43] Distributed Training with Tensroflow, *Tensroflow Core* https://www.tensorflow.org/guide/distributed_training, accessed on the June 24, 2019

[44] Katib, *Kubeflow* https://github.com/kubeflow/katib, accessed on the June 18, 2019

[45] Google Vizier: A Service for Black-Box Optimization https://static.googleusercontent.com/media/research.google.com/ja//pubs/archive/bcb15507f4b52991a0783013df4222240e942381.pdf, accessed on the June 18, 2019

[46] *Angular* https://angular.io/, accessed on the November 17, 2018

[47] *Python* https://www.python.org/, accessed on the November 17, 2018

[48] *Flask* https://flask.palletsprojects.com/en/1.1.x/, accessed on the November 19, 2018

[49] *The Go Project*, https://golang.org/, accessed on the November 19, 2018.

[50] *Kubebuilder - SDK for writing Kubernetes Controllers*, https://github.com/kubernetes-sigs/kubebuilder, accessed on the November 19, 2018.

[51] *Python Modules*, `https://docs.python.org/3/tutorial/modules.html`, accessed on the November 19, 2018.

[52] *Angular Environments*, `https://angular.io/guide/build#configure-environment-specific-defaults`, accessed on the November 17, 2018.

[53] *HTML iFrames*, `https://www.w3schools.com/html/html_iframe.asp`, accessed on the November 28, 2018.

[54] *Cross-Origin Communication*, `https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage`, accessed on the November 28, 2018.

[55] Exponential backoff, *Wikipedia - The Free Encyclopedia*, `https://en.wikipedia.org/wiki/Exponential_backoff`, accessed on the December 12, 2018.

[56] *WebSockets*, `https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API`, accessed on the December 19, 2018.

[57] *Flask Blueprints*, `https://flask.palletsprojects.com/en/1.1.x/blueprints/`, accessed on the November 19, 2018.

[58] *Go modules*, `https://github.com/golang/go/wiki/Modules`, accessed on the November 19, 2018.

[59] *Prometheus - Monitoring system & time series database*, `https://prometheus.io/`, accessed on the July 2, 2018.

[60] *Jupyter Notebook REST API*, `https://github.com/jupyter/jupyter/wiki/Jupyter-Notebook-Server-API`, accessed on the November 19, 2018.

[61] Persistent Volumes, *Kubernetes*, `https://kubernetes.io/docs/concepts/storage/persistent-volumes/#access-modes`, accessed on the November 3, 2018.

[62] *Cloud Commander*, `https://cloudcmd.io/`, accessed on the February 8, 2019.

[63] KFServing, *Kubeflow*, `https://www.kubeflow.org/docs/components/serving/kfserving/`, accessed on the May 19, 2019.