Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων
Και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης Και Υπολογιστων

Βελτιστοποίηση Ροών Εργασίας Μηχανικής Μάθησης με την Ενσωμάτωση ενός Αποθετηρίου Χαρακτηριστικών στην Πλατφόρμα Kubeflow

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Κωνσταντίνος Γ. Παπαϊωάννου

Αθήνα, Ιούλιος 2022

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων Και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης Και Υπολογιστων

Βελτιστοποίηση Ροών Εργασίας Μηχανικής Μάθησης με την Ενσωμάτωση ενός Αποθετηρίου Χαρακτηριστικών στην Πλατφόρμα Kubeflow

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Κωνσταντίνος Γ. Παπαϊωάννου

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή τη 15η Ιουλίου 2022.

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

Αθήνα, Ιούλιος 2022

. . . . . .

Κωνσταντίνος Γ. Παπαϊωάννου
Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

Copyright © Κωνσταντίνος Γ. Παπαϊωάννου, 2022
Με επιφύλαξη παντός δικαιώματος. All rights reserved.
Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

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

Optimizing ML Workflows by Integrating a Feature Store into Kubeflow

DIPLOMA THESIS

Konstantinos G. Papaioannou

Athens, July 2022

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

Optimizing ML Workflows by Integrating a Feature Store into Kubeflow

DIPLOMA THESIS

Konstantinos G. Papaioannou

Supervisor: Nectarios Koziris
Professor NTUA

Approved by the three-member examination committee on the 15th of July 2022.

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

Athens, July 2022

Η στοιχειοθεσία του κειμένου έγινε με το XƎTEX 0.999992.

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

Περίληψη

Η μηχανική μάθηση (ΜΜ) είναι σημαντική για τις επιχειρήσεις, γιατί βοηθάει στην επίλυση ποικίλων προβλημάτων, ενώ η ποιότητα των λύσεων που προσφέρει συμβαδίζει με αυτή των χαρακτηριστικών που χρησιμοποιούν τα μοντέλα ΜΜ. Οι επιστήμονες δεδομένων καταπιάνονται με την υλοποίηση ροών εργασίας που υποστηρίζουν ολόκληρο τον κύκλο ζωής της ΜΜ, μέρος του οποίου αφορά τη δημιουργία και τη διάθεση χαρακτηριστικών κατά την εκπαίδευση μοντέλων και την εξαγωγή συμπερασμάτων. Τα χαρακτηριστικά είναι μεταβλητές εισόδου σε ένα μοντέλο και είναι συνήθως ακατέργαστα δεδομένα που αναπαρίστανται ή κωδικοποιούνται σε διαφορετικές μορφές.

Η διαδικασία αυτή απαιτεί πολύ χρόνο και προσπάθεια και συνεπάγεται διάφορες προκλήσεις. Οι πηγές δεδομένων έχουν διαφορετικά χαρακτηριστικά και απαιτούν διαφορετικές προσεγγίσεις στην κατανάλωση και τη διαχείρισή τους. Επιπλέον, η διάθεση χαρακτηριστικών με συνεπή τρόπο στη χρονοβόρα εκπαίδευση μοντέλων και την αστραπιαία εξαγωγή συμπερασμάτων είναι δύσκολη και οδηγεί συχνά στο πρόβλημα της στρέβλωσης μεταξύ εκπαίδευσης και διάθεσης ενός μοντέλου.

Ένα αποθετήριο χαρακτηριστικών, όπως το Feast, λύνει αυτές τις προκλήσεις τυποποιώντας τους ορισμούς των χαρακτηριστικών, καταχωρώντας τα σε ένα κεντρικό αποθετήριο και κάνοντάς τα διαθέσιμα με συνέπεια κατά την εκπαίδευση μοντέλων και την εξαγωγή συμπερασμάτων. Η ενσωμάτωση του στο Kubeflow επιτρέπει την κοινή χρήση και επαναχρησιμοποίηση αυτών από πολλούς χρήστες και ομάδες.

Η λύση μας μετατρέπει το Feast σε ένα νεφο-εγγενές σύστημα πελάτη-εξυπηρετητή. Δημιουργούμε έναν διακομιστή REST API που διαχειρίζεται και αποθηκεύει τους ορισμούς χαρακτηριστικών και τα μεταδεδομένα τους σε ποικίλες βάσεις δεδομένων SQL οι οποίες υποστηρίζονται μέσω του μηχανισμού Object-Relational Mapping (ORM). Επιπλέον, επιβάλλουμε έλεγχο πρόσβασης με τον μηχανισμό RBAC του Κυβερνήτη και επεκτείνουμε τον πελάτη του Feast ώστε να χρησιμοποιεί το νέο API.

Λέξεις-Κλειδιά

Feast, αποθετήριο χαρακτηριστικών, Kubeflow, Kubernetes, διαμοιρασμός χαρακτηριστικών, συνέπεια χαρακτηριστικών, μηχανική μάθηση, νέφος

Abstract

Machine learning (ML) is a core piece of businesses' daily life, as it proves to be valuable for solving various real-world problems. What is more, the quality of solutions it offers goes hand in hand with the quality of data and features used by ML models and applications. Data scientists put a lot of effort in designing and implementing workflows that support the entire ML lifecycle, a large part of which deals with creating and extracting features from raw data, as well as serving these features during model training and inference. Features are variables that act as input in a model. They are usually data represented or encoded in different forms, but we can also find them in the form of raw data.

This part of the process requires a lot of time and effort and entails various challenges. Data sources have different characteristics and require different approaches in their consumption and management. In addition, making features available both for time-consuming model training and lightning-fast model inference in a consistent way is tough and may result in the very common training-serving skew problem.

A feature store such as Feast can solve these challenges by standardizing feature definitions, registering them in a central repository and making them consistently available during model training and inference. Integrating a feature store into Kubeflow, which is a complete ML platform, also allows sharing and reuse of features by multiple users and teams.

The solution we propose turns Feast into a cloud native client-server system. We create a REST API server which manages and stores all feature definitions and their metadata in an SQL database. We support different SQL databases through the use of Object-Relational Mapping (ORM) mechanism and we enforce access control using Kubernetes RBAC mechanism. Finally, we extend the Feast client to use the new API in order to access feature definitions through the server.

Keywords

Feast, feature store, Kubeflow, Kubernetes, features sharing, features consistency, machine learning, cloud

Αντί Προλόγου

Σε αυτό το σημείο θα ήθελα να ευχαριστήσω όλους όσους συνέβαλαν στην εκπόνηση αυτής της διπλωματικής εργασίας, η οποία και σηματοδοτεί το τέλος των προπτυχιακών σπουδών μου στη Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών. Παράλληλα, θα ήθελα να ευχαριστήσω όλους όσους ήταν δίπλα μου σε αυτή την πενταετή πορεία και με βοήθησαν να εξελιχθώ ως επιστήμονας, ως μηχανικός και κυρίως ως άνθρωπος.

Αρχικά, ευχαριστώ τον επιβλέποντα της διπλωματικής εργασίας μου, Καθηγητή Νεκτάριο Κοζύρη, που με εισήγαγε για πρώτη φορά στον κόσμο των υπολογιστικών συστημάτων και με ώθησε να ασχοληθώ με αυτά. Ένα μεγάλο ευχαριστώ οφείλω και στον Αναπληρωτή Καθηγητή Γεώργιο Γκούμα που κατάφερε ακόμα και με την εξ αποστάσεως διδασκαλία, στον δύσκολο καιρό της πανδημίας, να ενισχύσει το ενδιαφέρον μου για τα υπολογιστικά συστήματα και να με κάνει να αγαπήσω ιδιαίτερα τα λειτουργικά συστήματα.

Ξεχωριστή αναφορά θέλω να κάνω στον Διδάκτορα Βαγγέλη Κούκη που αποτέλεσε σημείο καμπής στη φοιτητική μου πορεία. Όχι μόνο με έκανε να λατρέψω τα λειτουργικά συστήματα, αλλά λειτούργησε και ως μέντορας μεταλαμπαδεύοντας το πάθος του για αυτά σε μένα και διαμορφώνοντας καθοριστικά τον τρόπο σκέψης μου ως μηχανικού. Τον ευχαριστώ επίσης που μου έδωσε την ευκαιρία να εκπονήσω τη διπλωματική μου ως μέλος της Arrikto, να έρθω σε επαφή με εξαιρετικούς μηχανικούς και να καταπιαστώ με τεχνολογίες αιχμής. Εκεί, συνεργάστηκα και με τον Stefano Fioravanzo τον οποίο επίσης ευχαριστώ για την υποστήριξη και την καθοδήγηση καθόλη τη διάρκεια

αυτής της διπλωματικής εργασίας, αλλά κυρίως επειδή με έμαθε να εργάζομαι επαγγελματικά ως μέλος μιας ομάδας μηχανικών λογισμικού.

Κλείνοντας, θέλω αρχικά να ευχαριστήσω τους φίλους και συνοδοιπόρους που είχα κατά τη φοιτητική μου ζωή, και ιδιαίτερα τον Βασίλη και τον Ορφέα, με τους οποίους μοιράστηκα πολλές ωραίες στιγμές που θα θυμάμαι για πάντα. Τέλος, ευχαριστώ όλους όσους θεωρώ οικογένειά μου αλλά πρωτίστως τους γονείς μου, Γιώργο και Όλγα, και τα αδέρφια μου, Βύρωνα και Φωτεινή, που ήταν και είναι δίπλα σε κάθε προσπάθειά μου.

Κωνσταντίνος Παπαϊωάννου
Ιούλιος 2022

Περιεχόμενα

Περίληψη **v**

Abstract **vii**

Αντί Προλόγου **ix**

1 Εισαγωγή **1**

1.1 Κίνητρο 1

1.2 Διατύπωση Προβλήματος 3

1.3 Υπάρχουσες Λύσεις 5

1.4 Προτεινόμενη Λύση 8

1.5 Δομή Διπλωματικής Εργασίας 9

2 Υπόβαθρο **11**

2.1 Περιέκτες 11

2.2 Κυβερνήτης 12

2.2.1 Επισκόπηση 13

2.2.2 Αρχιτεκτονική 13

2.2.3 Βασικές Έννοιες 15

2.3 Kubeflow 17

2.3.1 Επισκόπηση 17

2.3.2 Βασικές Έννοιες 18

2.4 Αποθετήριο Χαρακτηριστικών 20

2.4.1 Επισκόπηση 20

2.4.2 Βασικά Στοιχεία 22

3 Σχεδίαση **25**
3.1 Αποθετήριο Χαρακτηριστικών Feast 25
3.1.1 Αρχιτεκτονική 26
3.1.2 Βασικές Έννοιες 31
3.1.3 Χρήση 36
3.2 Αρχιτεκτονική Μητρώου 41
3.2.1 Λειτουργία 41
3.2.2 Πρόσθετο RegistryStore 45
3.2.3 Προβλήματα Σχεδίασης 46
3.3 Νέα Αρχιτεκτονική 47
3.3.1 Αρχική Σχεδίαση 48
3.3.2 Βελτιωμένη Σχεδίαση 50
3.3.3 Υπηρεσία Μητρώου 52
3.3.4 Πελάτης Μητρώου 54

4 Υλοποίηση **55**
4.1 Επισκόπηση 55
4.2 Πυρήνας Feast 56
4.3 Υπηρεσία Μητρώου 58
4.4 Πελάτης Μητρώου 65
4.5 Ρύθμιση Συστάδας 69
4.5.1 Υπηρεσία Μητρώου 69
4.5.2 Υποστήριξη Πιστοποίησης 73
4.5.3 Υποστήριξη Εξουσιοδότησης 74

5 Επίλογος **77**
5.1 Συμπερασματικά Σχόλια 77
5.2 Μελλοντικό Έργο 78

1 Introduction **81**
1.1 Motivation 81
1.2 Problem Statement 83
1.3 Existing Solutions 84
1.4 Proposed Solution 87
1.5 Outline 88

2 Background **89**
2.1 Containers . . . 89
2.2 Kubernetes . . . 90
2.2.1 Overview . . . 90
2.2.2 Architecture . . . 91
2.2.3 Core Concepts . . . 93
2.3 Kubeflow . . . 94
2.3.1 Overview . . . 95
2.3.2 Core Concepts . . . 95
2.4 Feature Store . . . 97
2.4.1 Overview . . . 97
2.4.2 Core Components . . . 98

3 Design **101**
3.1 Feast Feature Store . . . 101
3.1.1 Architecture . . . 102
3.1.2 Core Concepts . . . 107
3.1.3 Usage . . . 110
3.2 Registry Architecture . . . 115
3.2.1 Functionality . . . 116
3.2.2 RegistryStore Plugin . . . 119
3.2.3 Problematic Designs . . . 120
3.3 New Architecture . . . 121
3.3.1 Initial Design . . . 121
3.3.2 Improved Design . . . 124
3.3.3 Registry Service . . . 126
3.3.4 Registry Client . . . 127

4 Implementation **129**
4.1 Overview . . . 129
4.2 The Feast Core . . . 130
4.3 Registry Service . . . 132
4.4 Registry Client . . . 139
4.5 Cluster Configuration . . . 143
4.5.1 Registry Service . . . 143
4.5.2 Authentication Support . . . 146
4.5.3 Authorization Support . . . 147

5 Conclusion **149**

5.1 Concluding Remarks 149

5.2 Future Work 150

Bibliography **153**

1 Εισαγωγή

Το πρώτο κεφάλαιο έχει ως στόχο να θέσει το πλαίσιο της παρούσας διπλωματικής. Αρχικά, παρουσιάζει τα κίνητρα που οδήγησαν στη διερεύνηση των αποθετηριών χαρακτηριστικών (feature stores). Στη συνέχεια, προβάλλει τα προβλήματα που στοχεύει να επιλύσει η όλη προσπάθεια καθώς και τις υπάρχουσες λύσεις. Τέλος, παρουσιάζει με συντομία την προτεινόμενη λύση.

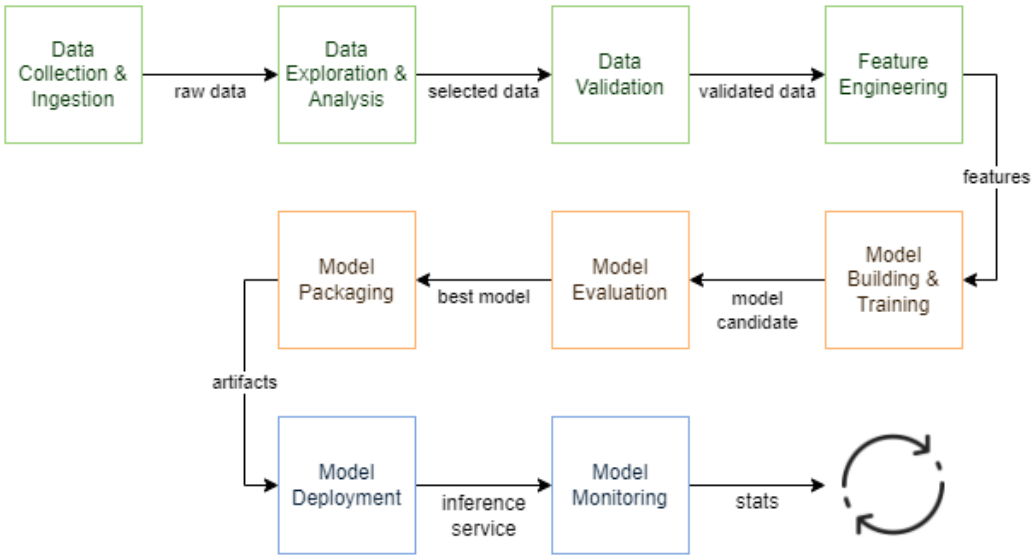
1.1 Κίνητρο

Στις μέρες μας, όλο και περισσότεροι άνθρωποι εμπλέκονται στη διαδικασία ανάπτυξης ή τουλάχιστον χρησιμοποιούν καθημερινά εφαρμογές μηχανικής μάθησης (ΜΜ), καθώς αποδεικνύεται πολύτιμο εργαλείο για την επίλυση σύνθετων προβλημάτων του πραγματικού κόσμου. Ωστόσο, τα φορτία εργασίας και οι διαδικασίες ΜΜ φαίνεται να ποικίλλουν στα διάφορα στάδια μιας εφαρμογής (ανάπτυξη, χρήση, συντήρηση). Συνεπώς, η παροχή των κατάλληλων συστημάτων και εργαλείων για επιστήμονες δεδομένων και μηχανικούς, ώστε να καταστούν τα φορτία εργασίας πιο αποδοτικά και οι διαδικασίες απλούστερες και βιώσιμες, είναι ιδιαίτερης σημασίας.

Κύκλος Ζωής και Ροές Εργασίας Μηχανικής Μάθησης

Εμβαθύνοντας στις καθημερινές διαδικασίες των επιστημόνων και των μηχανικών παρατηρούμε ότι εργάζονται πάντα πάνω στον σχεδιασμό και την υλοποίηση ροών εργασίας που υποστηρίζουν ολόκληρο τον κύκλο ζωής της ΜΜ. Μια υψηλού επιπέδου

επισκόπηση μιας τυπικής ροής εργασίας ΜΜ που προσπαθεί να ταιριάξει στον κύκλο ζωής μιας εφαρμογής ΜΜ είναι η ακόλουθη:



Σχήμα 1.1: *Τυπική Ροή Εργασίας ΜΜ*

Η παραπάνω ροή εργασίας περιλαμβάνει πολλαπλά βήματα και μπορεί να χωριστεί σε τρεις κύριες επιμέρους ροές εργασίας:

- Ροή εργασίας δεδομένων
- Ροή εργασίας μοντέλου
- Ροη εργασίας εξυπηρέτησης

Και οι τρεις επιμέρους ροές εργασίας απαιτούν διαφορετικά είδη τεχνογνωσίας και ειδικών δεξιοτήτων, καθιστώντας την ανάπτυξη και την λειτουργία ολόκληρης της ροής εργασίας δύσκολη και χρονοβόρα. Υποθέτοντας ότι μία ή περισσότερες ομάδες ανθρώπων εργάζονται σε καθένα από αυτά τα τρία μέρη, η ενασχόληση με αυτά ανεξάρτητα είναι απαραίτητη και λειτουργεί ως μεγάλο κίνητρο στην προσπάθεια βελτιστοποίησης των ροών εργασίας ΜΜ.

MLOps και Βέλτιστες Πρακτικές

Τα τελευταία χρόνια έχει εμφανιστεί μια νέα προσέγγιση που ονομάζεται MLOps και προσπαθεί να αντιμετωπίσει τις προκλήσεις στις ροές εργασίας και τα συστήματα ΜΜ

εφαρμόζοντας τις αρχές DevOps [27] σε αυτά. Αυτή η προσέγγιση είναι στην πραγματικότητα μια κουλτούρα και πρακτική μηχανικής στη ΜΜ που στοχεύει στην ενοποίηση της ανάπτυξης (Dev) και της λειτουργίας (Ops) [16] συστημάτων ΜΜ. Η αυτοματοποίηση και παρακολούθηση σε όλα τα στάδια μιας τυπικής ροής εργασίας ΜΜ θεωρούνται μεταξύ των βέλτιστων πρακτικών για την αποφυγή κοινών παγίδων που συνήθως περιγράφονται ως τεχνικό χρέος της ΜΜ [14]. Η ακολούθηση βέλτιστων πρακτικών για την απλοποίηση πολύπλοκων διαδικασιών και την περαιτέρω βελτίωση των ροών εργασίας ΜΜ είναι ένα ακόμα κίνητρο της παρούσας διπλωματικής.

1.2 Διατύπωση Προβλήματος

Οι ροές εργασίας μηχανικής δεδομένων (ή μηχανικής χαρακτηριστικών) είναι συνήθως ένα σημαντικό κομμάτι μιας ροής εργασίας ΜΜ, καθώς τα χαρακτηριστικά και τα δεδομένα γενικά έχουν εξαιρετικά σημαντική αξία για την επιτυχία ενός μοντέλου. Η δημιουργία νέων χαρακτηριστικών απαιτεί πολλή προσπάθεια και χρόνο, επομένως η επαναχρησιμοποίηση και ο διαμοιρασμός τους είναι ζωτικής σημασίας. Επιπλέον, τα χαρακτηριστικά που παράγονται πρέπει να χρησιμοποιούνται τόσο για την εκπαίδευση ενός μοντέλου (βλ. Ροή εργασίας μοντέλου) όσο και κατά τη διάρκεια της εξαγωγής συμπερασμάτων (βλ. Ροή εργασίας εξυπηρέτησης) με συνεπή τρόπο ώστε να αποφεύγεται η στρέβλωση μεταξύ εκπαίδευσης και διάθεσης ενός μοντέλου.

Προκλήσεις Συστημάτων Μηχανικής Μάθησης

Εξετάζοντας πιο προσεκτικά γιατί η δημιουργία χαρακτηριστικών και στη συνέχεια η εξυπηρέτηση είναι δύσκολη, το πρώτο πράγμα που παρατηρείται είναι η ποικιλία των πηγών δεδομένων από τις οποίες προέρχονται αυτά τα χαρακτηριστικά. Οι πηγές δεδομένων δέσμης (batch) ή ροής (stream) καθώς και ένα μείγμα και των δύο είναι πολύ συνηθισμένο στις εφαρμογές αιχμής στη ΜΜ. Είναι προφανές ότι αυτές οι πηγές έχουν διαφορετικά χαρακτηριστικά όσον αφορά την ποιότητα των δεδομένων ή τη φρεσκάδα τους. Για παράδειγμα, μια πηγή δέσμης, όπως μια αποθήκη δεδομένων (data warehouse), περιέχει ιστορικά δεδομένα χρονοσειρών που δημιουργούν ένα πλήρες ιστορικό ενός χαρακτηριστικού, ενώ μια πηγή ροής διατηρεί μόνο δεδομένα πραγματικού χρόνου. Ως αποτέλεσμα, οι λειτουργίες που μπορούν να εκτελεστούν σε αυτά τα

δεδομένα είναι επίσης εντελώς διαφορετικές.

Συνεχίζοντας, αυτά τα χαρακτηριστικά μπορεί να χρειαστεί να υπολογίζονται σε διαφορετικά χρονικά διαστήματα ή ακόμη και κατά τη στιγμή μιας πρόβλεψης, επομένως το να τρέχει μια διαδικασία παραγωγής χαρακτηριστικών σε διαφορετικές χρονικές στιγμές απαιτεί τεράστια προσπάθεια και μπορεί να αυξήσει σημαντικά το χρόνο πρόβλεψης. Κατά συνέπεια, απαιτείται αποσύνδεση της δημιουργίας χαρακτηριστικών από την κατανάλωση τους στα συστήματα παραγωγής ΜΜ. Τα νέα ερωτήματα που προκύπτουν είναι:

- Πόσο συχνά πρέπει να δημιουργούνται χαρακτηριστικά;
- Ποια είναι η σχέση κόστους-αποτελεσματικότητας;

Ακολουθεί το πολύ συνηθισμένο πρόβλημα της στρέβλωσης μεταξύ εκπαίδευσης και διάθεσης ενός μοντέλου [24]. Αυτό είναι το αποτέλεσμα της αποτυχίας να παρέχονται τα ίδια χαρακτηριστικά τόσο για την εκπαίδευση όσο και για την εξαγωγή συμπερασμάτων λόγω της δημιουργίας δύο διαφορετικών υλοποιήσεων για τη δημιουργία χαρακτηριστικών. Μιας υλοποίησης που συνήθως συμβαίνει σε συγκεκριμένα χρονικά διαστήματα και μιας άλλη που εκτελείται σε πραγματικό χρόνο κατά τη διάρκεια κάθε αιτήματος πρόβλεψης.

Δεδομένου ότι η ανάπτυξη χαρακτηριστικών είναι χρονοβόρα, η διασφάλιση ότι τα χαρακτηριστικά που δημιουργούνται είναι τυποποιημένα και ότι μπορούν να έχουν εύκολη πρόσβαση σε αυτά πολλοί επιστήμονες και μηχανικοί δεδομένων είναι μια άλλη πρόκληση. Η έλλειψη τρόπων κοινής χρήσης χαρακτηριστικών και συνεργασίας έχει ως αποτέλεσμα την υπερβολική επανάληψη των ίδιων χαρακτηριστικών μέσα σε ομάδες και οργανισμούς.

Τέλος, η διασφάλιση της ποιότητας των δεδομένων που παρέχονται στα μοντέλα αποτελεί επίσης πρόκληση. Ερωτήματα όπως "Λαμβάνει το μοντέλο μου τα σωστά δεδομένα και εξακολουθεί να λειτουργεί σωστά;" ή "Υπήρξε απόκλιση στα δεδομένα με την πάροδο του χρόνου;" είναι πολύ συνηθισμένα στα συστήματα παραγωγής ΜΜ.

Συνοψίζοντας, οι 4 κύριες προκλήσεις που στοχεύει να λύσει, ή τουλάχιστον να μειώσει τις αρνητικές επιπτώσεις τους η προσπάθεια αυτής της διπλωματικής είναι οι εξής:

- Δημιουργία ροών χαρακτηριστικών

- Συνεπής πρόσβαση στα δεδομένα
- Διπλότυπα χαρακτηριστικά
- Διασφάλιση της ποιότητας των δεδομένων

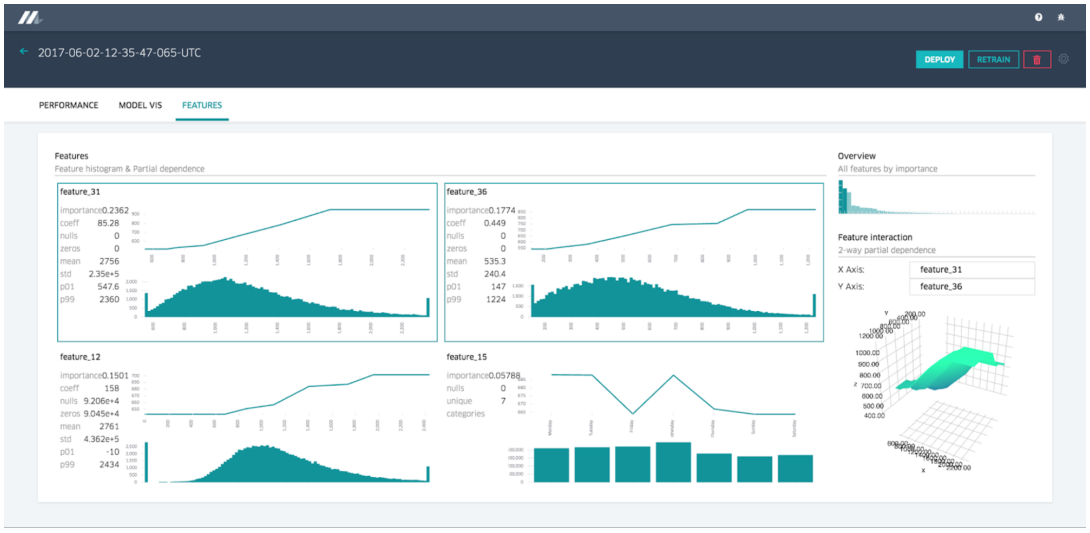
1.3 Υπάρχουσες Λύσεις

Μέχρι σήμερα, μια πληθώρα έργων και οργανισμών έχουν προσπαθήσει να αναπτύξουν τις δικές τους λύσεις για την αντιμετώπιση των προαναφερθέντων προκλήσεων. Οι ιδέες και τα προϊόντα που έχουν δημιουργήσει αποτελούν ένα εξαιρετικό σημείο εκκίνησης για να κατανοήσουμε την αξία που φέρνουν στον κόσμο της ΜΜ.

Πλατφόρμα Μηχανικής Μάθησης Michelangelo

Το Michelangelo [23] της Uber ήταν ουσιαστικά η πρώτη ολοκληρωμένη πλατφόρμα ΜΜ που δημιουργήθηκε γύρω στο 2017 ως ανάγκη για την αντιμετώπιση μιας σειράς προκλήσεων που σχετίζονται με τη δημιουργία και την ανάπτυξη πολυάριθμων μοντέλων μηχανικής μάθησης σε κλίμακα. Κατά τη διάρκεια των πρώτων ετών της Uber οι επιστήμονες δεδομένων χρησιμοποιούσαν διάφορα εργαλεία για τη δημιουργία μοντέλων και οι μηχανικοί κατασκεύαζαν μεμονωμένα συστήματα προσαρμοσμένα για τη χρήση αυτών των μοντέλων στην παραγωγή. Το Michelangelo τυποποίησε τις ροές εργασίας και τα εργαλεία μεταξύ των ομάδων παρέχοντας ένα ολοκληρωμένο σύστημα που βοήθησε στην εύκολη δημιουργία και λειτουργία συστημάτων ΜΜ σε κλίμακα.

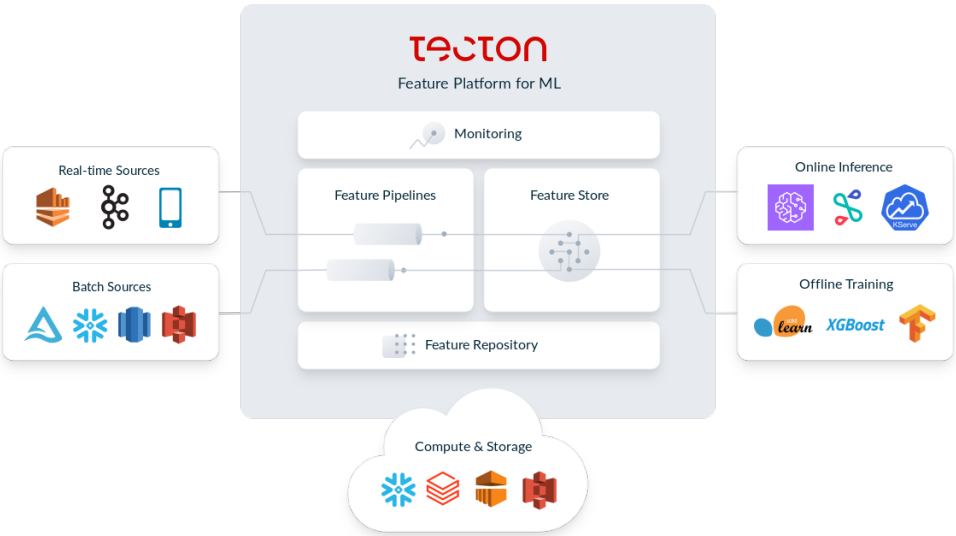
Η δημιουργία καλύτερων συστημάτων για τη διαχείριση και την κοινή χρήση ροών χαρακτηριστικών ήταν ένα σημαντικό μέρος του όλου συστήματος. Έτσι γεννήθηκε ο όρος Feature Store. Οι μηχανικοί της Uber δημιούργησαν ένα πλήρες σύστημα για τη δημιουργία ροών δεδομένων που δημιουργούν χαρακτηριστικά και σετ δεδομένων τόσο για την εκπαίδευση μοντέλων όσο και για προβλέψεις. Πρόσθεσαν επίσης ένα επίπεδο διαχείρισης δεδομένων που επέτρεπε στις ομάδες να μοιράζονται, να ανακαλύπτουν και να χρησιμοποιούν περισσότερα από 10.000 χαρακτηριστικά, τα οποία υπολογίζονται και ενημερώνονται αυτόματα καθημερινά. Τέλος, δημιουργήθηκαν εργαλεία παρακολούθησης χαρακτηριστικών που παρατηρούν τη σημασία ενός χαρακτηριστικού σε ένα μοντέλο μαζί με διαγράμματα μερικής εξάρτησης και ιστογράμματα κατανομής.



Σχήμα 1.2: *Αναφορά Χαρακτηριστικού*

Πλατφόρμα Χαρακτηριστικών Tecton

Η Tecton ιδρύθηκε από μια ομάδα ανθρώπων που δημιούργησαν το Michelangelo της Uber. Αντί να εστιάζει σε ολόκληρη τη διαδικασία ΜΜ, παρέχει μια έτοιμη πλατφόρμα χαρακτηριστικών για επιχειρήσεις, η οποία έχει κατασκευαστεί για να ενορχηστρώνει τον πλήρη κύκλο ζωής των χαρακτηριστικών, από τον μετασχηματισμό έως την απευθείας διάθεσή τους [28]. Η λύση ανοικτού κώδικα που συμβάλλει ενεργά η Tecton είναι το Feast, το οποίο θα αποτελέσει και το επίκεντρο της παρούσας διπλωματικής εργασίας.



Σχήμα 1.3: *Αρχιτεκτονική Tecton*

Η Tecton επιτρέπει στους χρήστες να ορίζουν και να διαχειρίζονται χαρακτηριστικά χρησιμοποιώντας κώδικα σε ένα αποθετήριο git. Τους βοηθάει να υπολογίζουν και να ενορχηστρώνουν αυτόματα μετασχηματισμούς χαρακτηριστικών, καθώς και να αποθηκεύουν με συνέπεια τα online και offline δεδομένα των χαρακτηριστικών. Τέλος, φροντίζει για την αποτελεσματική διάθεση των χαρακτηριστικών τόσο κατά την εκπαίδευση όσο και κατά την εξαγωγή συμπερασμάτων.

Αποθετήριο Χαρακτηριστικών Vertex AI

Το Vertex AI που αναπτύχθηκε από την Google είναι μια άλλη πλατφόρμα ΜΜ που στοχεύει στην ταχύτερη κατασκευή, ανάπτυξη και κλιμάκωση μοντέλων ΜΜ, με προεκπαιδευμένα και προσαρμοσμένα εργαλεία. Μέρος αυτής της πρωτοποριακής πλατφόρμας είναι ένα αποθετήριο χαρακτηριστικών. Το αποθετήριο χαρακτηριστικών της Vertex AI παρέχει ένα κεντρικό αποθετήριο για την οργάνωση, αποθήκευση και διάθεση των χαρακτηριστικών ΜΜ [15].

Μεταξύ άλλων πλεονεκτημάτων, παρέχει ένα διαχειριζόμενο επίπεδο απευθείας διάθεσης χαρακτηριστικών, μηχανισμούς εντοπισμού απόκλισης δεδομένων και αυτόματη συντήρηση δεδομένων, διατηρώντας τις τιμές των χαρακτηριστικών φρέσκες. Επιπλέον, επιβάλλει ποσοστώσεις και όρια για την αποτελεσματική διαχείριση των πόρων τόσο στο επίπεδο διάθεσης χαρακτηριστικών όσο και στο επίπεδο δεδομένων.

Εξατομικευμένες Λύσεις

Αυτή τη στιγμή, όλο και περισσότεροι οργανισμοί και μεγάλες εταιρείες τεχνολογίας έχουν δημιουργήσει τις δικές τους προσαρμοσμένες λύσεις για την επίλυση μέρους ή όλων των προκλήσεων που αναφέρθηκαν προηγουμένως. Ακολουθεί ένας κατάλογος με τις πιο συνηθισμένες λύσεις:

Amazon SageMaker Feature Store	Rasgo
Databricks Feature Store	Scribble Data
Hopswork Feature Store	AirBnB Zipline
Linkedin Feathr	FBLearner Flow

Spotify Jukebox Feature Store	Salesforce Feature Store
Apple Overton	Netflix Metaflow

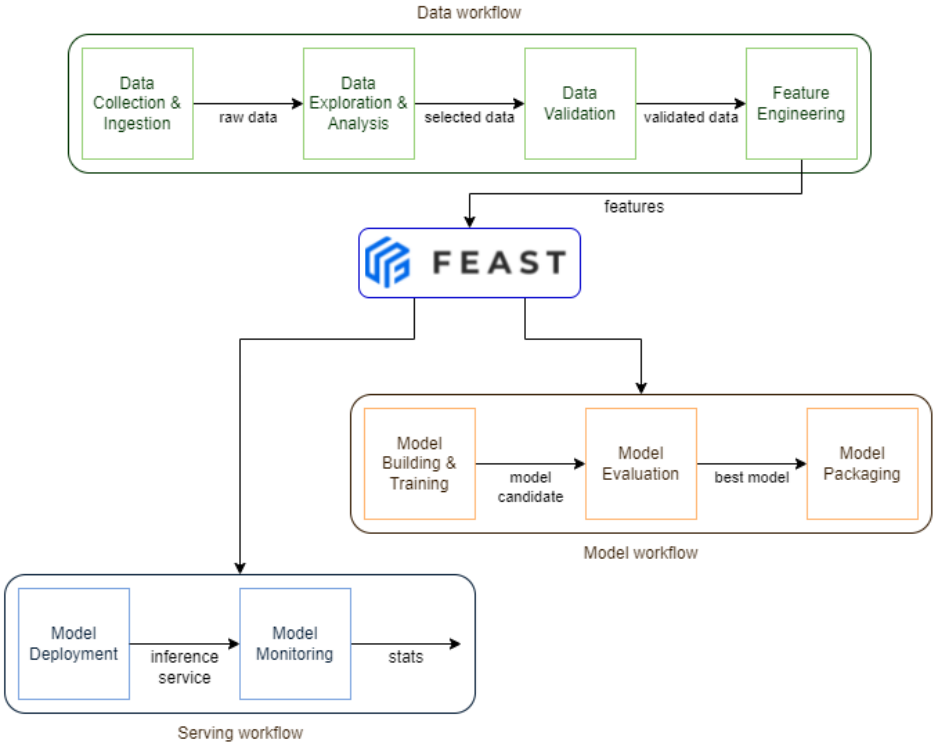
1.4 Προτεινόμενη Λύση

Ενσωμάτωση Feast στην Πλατφόρμα Kubeflow

Σκοπός της παρούσας διπλωματικής είναι η ενσωμάτωση ενός αποθετηρίου χαρακτηριστικών μέσα σε μια υπάρχουσα πλατφόρμα ΜΜ, στην προκειμένη περίπτωση το Kubeflow. Το Kubeflow είναι ήδη μια καλά αναπτυγμένη και συντηρούμενη πλατφόρμα ανοικτού κώδικα, αλλά εξακολουθεί να λείπει ένας τρόπος διαχείρισης δεδομένων και χαρακτηριστικών από αυτή. Η ενσωμάτωσή του με ένα αποθετήριο χαρακτηριστικών θα επιχειρήσει να λύσει όλες τις προκλήσεις που αναφέρθηκαν προηγουμένως και θα προσπαθήσει να αποσυνδέσει ουσιαστικά τη ροή εργασίας της μηχανικής δεδομένων από τις ροές εργασίας εκπαίδευσης μοντέλων και εξυπηρέτησης μοντέλων (βλ. Σχήμα 1.4). Επιπλέον, θα βοηθήσει τους χρήστες να κάνουν τις ροές εργασίας ΜΜ απλούστερες, πιο φορητές και πιο επεκτάσιμες.

Η χρήση του Feast ως το επιλεγμένο αποθετήριο χαρακτηριστικών έχει πολλά πλεονεκτήματα, καθώς είναι ανοικτού κώδικα και δεν βασίζεται σε τεχνολογίες ή υποδομές συγκεκριμένων προμηθευτών. Είναι ακριβώς ο κατάλληλος υποψήφιος καθώς έχει επίσης πολλούς ανθρώπους που δουλεύουν σε αυτό και μια ενεργή κοινότητα.

Για την επιτυχή ενσωμάτωση του Feast στο Kubeflow, το μητρώο (Registry) του θα επεκταθεί. Ο στόχος είναι να εφαρμοστούν ελάχιστες αλλαγές στον υπάρχων κώδικα και να αξιοποιηθεί όσο το δυνατόν περισσότερο ο μηχανισμός πρόσθετων που προσφέρει το Feast. Για το λόγο αυτό, θα δημιουργηθεί ένας πελάτης (client) που θα χειρίζεται όλες τις αλληλεπιδράσεις με το νέο μητρώο. Το μητρώο θα είναι ένας διακομιστής REST API που θα είναι υπεύθυνος για την αποθήκευση των ορισμών των χαρακτηριστικών και των σχετικών μεταδεδομένων σε μια βάση δεδομένων SQL, καθώς και για την επιβολή μηχανισμών ελέγχου πρόσβασης. Αξίζει να αναφερθεί το γεγονός ότι αυτό το νέο είδος μητρώο είναι μέρος των τρεχόντων στόχων του Feast και κάτι που ζητείται ιδιαίτερα από τους ενεργούς χρήστες. Τελικός σκοπός είναι το νέο μητρώο να προσφερθεί στο έργο ανοικτού κώδικα και να το καταστήσει ακόμη πιο ευέλικτο και επεκτάσιμο.



Σχήμα 1.4: *Διασπασμένη Ροή Εργασίας ΜΜ*

1.5 Δομή Διπλωματικής Εργασίας

Το υπόλοιπο της παρούσας διπλωματικής διαρθρώνεται ως εξής:

- **Κεφάλαιο 2**: Επισκόπηση υψηλού επιπέδου χρήσιμων τεχνολογιών, όρων και εννοιών
- **Κεφάλαιο 3**: Πλήρης παρουσίαση της τρέχουσας αρχιτεκτονικής του Feast και της σχεδίασης των νέων στοιχείων του
- **Κεφάλαιο 4**: Ειδικές λεπτομέρειες υλοποίησης σχετικά με τα νέα στοιχεία που θα δημιουργηθούν
- **Κεφάλαιο 5**: Συμπερασματικές παρατηρήσεις και πιθανό μελλοντικό έργο

2 Υπόβαθρο

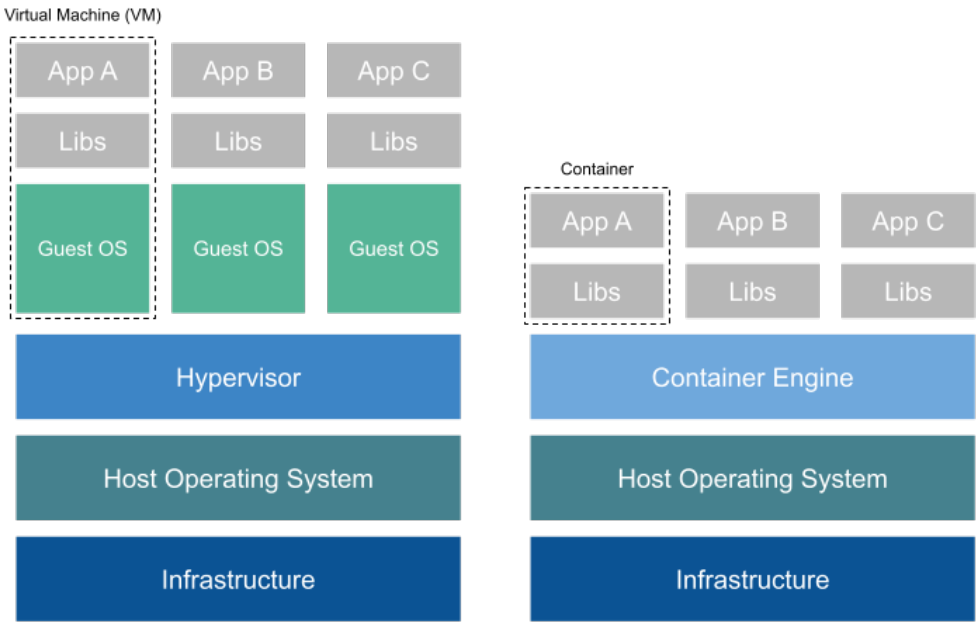
Αυτό το κεφάλαιο παρουσιάζει μια υψηλού επιπέδου επισκόπηση όλων των προαπαιτούμενων γνώσεων. Η βασική κατανόηση των ακόλουθων εννοιών και τεχνολογιών είναι απαραίτητη για την κατανόηση των τμημάτων σχεδιασμού και υλοποίησης αυτής της διπλωματικής. Για αρχή, εξηγεί τους περιέκτες (containers) και τον Κυβερνήτη (Kubernetes), καθώς τα πάντα τρέχουν πάνω σε αυτούς. Στη συνέχεια, παρουσιάζει το Kubeflow, μια πλήρης πλατφόρμα ΜΜ που αξιοποιεί τον Κυβερνήτη για την υποστήριξη ολόκληρου του κύκλου ζωής της ΜΜ. Τέλος, περιγράφει τα βασικά στοιχεία και τις λειτουργίες ενός αποθετηρίου χαρακτηριστικών.

2.1 Περιέκτες

Στις μέρες μας, οι περιέκτες (containers) είναι η πιο δημοφιλής μορφή εικονικοποίησης λειτουργικού συστήματος. Πρόκειται για εκτελέσιμες μονάδες λογισμικού στις οποίες περιέχεται ο κώδικας της εφαρμογής, μαζί με τις βιβλιοθήκες και τις εξαρτήσεις του, έτσι ώστε να μπορεί να εκτελεστεί σε πολλαπλά περιβάλλοντα, όπως το νέφος ή ένας προσωπικός υπολογιστής [18]. Πολλαπλοί περιέκτες μπορούν να εκτελούνται στο ίδιο μηχάνημα και να μοιράζονται τον πυρήνα του λειτουργικού συστήματος με άλλους περιέκτες, καθένας από τους οποίους εκτελείται ως απομονωμένη διεργασία στο χώρο χρήστη. Αυτές οι διεργασίες είναι επίσης ελαφριές, αποδοτικές και φορητές, επιτρέποντας την πλήρη αξιοποίηση των υποκείμενων πόρων του μηχανήματος [1].

Για την καλύτερη κατανόηση των περιεκτών, ας δούμε πώς διαφέρουν από τις παραδοσιακές εικονικές μηχανές (VM). Τα VMs είναι μια αφαίρεση του φυσικού υλικού (CPU,

μνήμη, αποθήκευση κ.λπ.) ενώ οι περιέκτες είναι μια αφαίρεση στο επίπεδο εφαρμογών. Στην πρώτη περίπτωση, αξιοποιούμε έναν υπερεπόπτη για την εικονικοποίηση του φυσικού υλικού, επομένως κάθε VM περιέχει ένα φιλοξενούμενο λειτουργικό σύστημα, ένα εικονικό αντίγραφο του υλικού που απαιτεί το λειτουργικό σύστημα για να τρέξει, μαζί με μια εφαρμογή και τις εξαρτήσεις της. Στη δεύτερη περίπτωση, οι περιέκτες εικονικοποιούν το λειτουργικό σύστημα, ώστε κάθε περιέκτης να περιέχει μόνο την εφαρμογή και τις εξαρτήσεις της. Η απουσία φιλοξενούμενου λειτουργικού συστήματος έχει ως αποτέλεσμα την εξοικονόμηση πόρων και την βελτίωση των επιδόσεων.



Σχήμα 2.1: *Εικονικές Μηχανές - Περιέκτες*

2.2 Κυβερνήτης

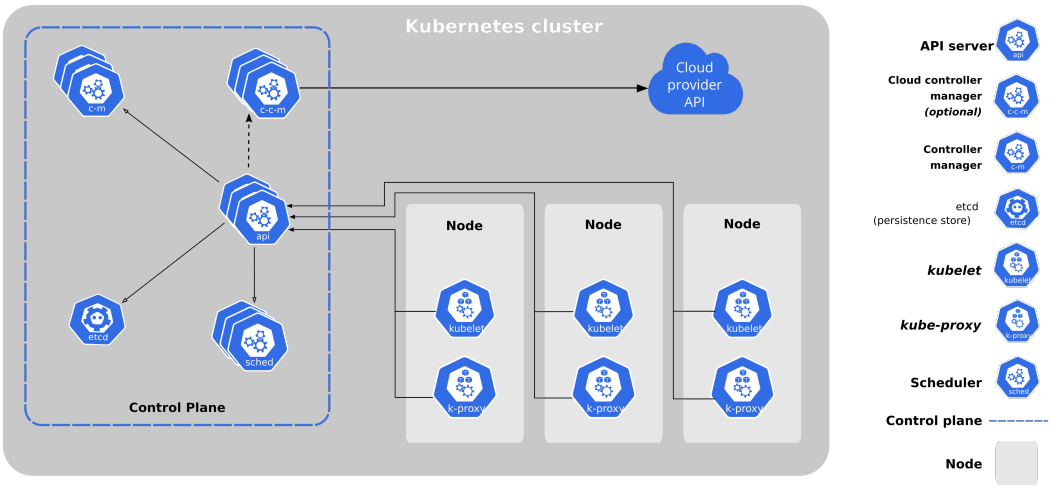
Έχοντας ήδη εξηγήσει τι είναι οι περιέκτες, είναι σαφές ότι η ενσωμάτωση εφαρμογών σε αυτούς φέρνει πολλά οφέλη. Ωστόσο, η τυποποίηση της ανάπτυξης εφαρμογών και η χρήση πόρων αποτελεσματικά σε πολλαπλά περιβάλλοντα είναι μόνο η αρχή. Σε μεγάλης κλίμακας περιβάλλοντα όπου πολλαπλές εφαρμογές εκτελούνται σε εκατοντάδες κόμβους υπάρχει ζωτική ανάγκη για την ενορχήστρωση και τη διαχείριση αυτών των εφαρμογών. Εδώ είναι που ο Κυβερνήτης (Kubernetes ή K8s) δίνει την λύση.

2.2.1 Επισκόπηση

Ο πυρήνας του Κυβερνήτη είναι ένας ενορχηστρωτής εφαρμογών μέσα σε περιέκτες. Πρόκειται για ένα σύστημα που αναπτύσσει και διαχειρίζεται εφαρμογές δυναμικά χωρίς την ανάγκη παρέμβασης του χρήστη. Για το σκοπό αυτό, παρέχει έναν δηλωτικό τρόπο προσδιορισμού μιας επιθυμητής κατάστασης και είναι υπεύθυνος για την αντιστοίχισή της με την πραγματική κατάσταση λειτουργίας του συστήματος. Το καλύτερο μέρος του είναι ότι πρόκειται για ένα ταχύτατα εξελισσόμενο έργο ανοικτού κώδικα που μπορεί να τρέξει σε οποιοδήποτε νέφος ή ιδιωτικό κέντρο δεδομένων, αντιμετωπίζοντας την υποκείμενη υποδομή με αφαιρετικό τρόπο [9].

2.2.2 Αρχιτεκτονική

Μετά την ανάπτυξη του Κυβερνήτη δημιουργείται μια συστάδα (cluster). Αυτή η συστάδα αποτελείται από ένα σύνολο κόμβων και ένα επίπεδο ελέγχου. Το επίπεδο ελέγχου θα μπορούσε να θεωρηθεί ως ο εγκέφαλος της συστάδας, ενώ οι κόμβοι ως οι μύες. Πιο συγκεκριμένα, το επίπεδο ελέγχου εκθέτει μια προγραμματιστική διεπαφή (API), διαθέτει έναν χρονοδρομολογητή για την ανάθεση εργασιών στους κόμβους και καταγράφει την κατάστασή του σε έναν μόνιμο αποθηκευτικό χώρο. Οι κόμβοι είναι υπεύθυνοι για την εκτέλεση των εφαρμογών, πράγμα που σημαίνει ότι παρακολουθούν το επίπεδο ελέγχου για νέες εργασίες, τις εκτελούν και αναφέρουν την κατάσταση λειτουργίας σε αυτό. [26]



Σχήμα 2.2: *Στοιχεία Συστάδας Κυβερνήτη*

Επίπεδο Ελέγχου

Το επίπεδο ελέγχου (control plane) περιλαμβάνει 5 κύρια τμήματα:

API Server Εκθέτει το API του Κυβερνήτη που χρησιμοποιείται από όλα τα τμήματα για να επικοινωνούν μεταξύ τους. Οι χρήστες στέλνουν manifests (αρχεία ρυθμίσεων YAML) που περιέχουν την επιθυμητή κατάσταση μιας εφαρμογής στον API server, ο οποίος αρχικά πιστοποιεί και εξουσιοδοτεί τα αιτήματα, στη συνέχεια επικυρώνει και διατηρεί αυτά τα αρχεία στον αποθηκευτικό χώρο της συστάδας (cluster store) και τέλος τα αναπτύσσει στη συστάδα.

Cluster Store Είναι το μόνο τμήμα του επιπέδου ελέγχου που διατηρεί κατάσταση και το οποίο είναι υπεύθυνο για την αποθήκευση των ρυθμίσεων και της κατάστασης της συστάδας. Βασίζεται στο etcd, σε έναν κατανεμημένο αποθηκευτικό χώρο κλειδιών-τιμών, και λειτουργεί ως η μοναδική πηγή της αλήθειας. Το cluster store είναι ζωτικής σημασίας, καθώς η απουσία ή η αποτυχία του σημαίνει ουσιαστικά ότι δεν υπάρχει συστάδα. Η συνέπεια είναι η νούμερο ένα προτεραιότητα και σε περίπτωση που κάτι πάει στραβά το σύστημα θα σταματήσει και θα περιμένει ώστε να τη διατηρήσει.

Scheduler Παρακολουθεί τον API server για νέα Pods ή εργασίες και τα αναθέτει σε υγιείς κόμβους εργασίας. Για να εκτελέσει ένας κόμβος μια εργασία πραγματοποιούνται πολλαπλοί έλεγχοι σχετικά με τους ελεύθερους πόρους, τις διαθέσιμες θύρες δικτύου κ.λπ. και οι κόμβοι κατατάσσονται ανάλογα. Ο κόμβος με την υψηλότερη κατάταξη θα εκτελέσει τη ζητούμενη εργασία. Σε περίπτωση που ένας κατάλληλος κόμβος δεν μπορεί να βρεθεί, μια εργασία θα σημειωθεί ως εκκρεμής και θα περιμένει μέχρι να δρομολογηθεί.

Controller Manager Είναι υπεύθυνος για την εκτέλεση βρόχων ελέγχου που παρακολουθούν την συστάδα και ανταποκρίνονται σε συμβάντα. Στην πραγματικότητα πρόκειται για ένα σύνολο ελεγκτών (controllers) που εκτελούνται στο πλαίσιο μιας ενιαίας διεργασίας. Αυτοί οι ελεγκτές εκτελούν βρόχους που παρακολουθούν συνεχώς τον API server και διασφαλίζουν ότι η τρέχουσα κατάσταση της συστάδας ταιριάζει με την επιθυμητή.

Cloud Controller Manager Σε συστάδες K8s που εκτελούνται σε πλατφόρμες νέφους, όπως AWS, Azure, GCP κ.λπ., χειρίζεται ελεγκτές που αφορούν ειδικά τον πάροχο cloud. Για παράδειγμα, δημιουργεί, ενημερώνει ή διαγράφει τους εξισορροπητές φορτίου του παρόχου του νέφους για εφαρμογές που χρησιμοποιούν εξισορροπητές φορτίου που βλέπουν στο διαδίκτυο.

Κόμβοι

Οι κόμβοι αποτελούνται από τρία κύρια τμήματα:

Kubelet Πρόκειται για έναν πράκτορα που εκτελείται σε κάθε κόμβο και είναι υπεύθυνος για την εγγραφή του στη συστάδα. Παρακολουθεί ενεργά τον API server για νέες εργασίες, τις εκτελεί και αναφέρει την κατάστασή τους πίσω σε αυτόν.

Περιβάλλον Εκτέλεσης Περιεκτών Κάθε κόμβος έχει ένα περιβάλλον εκτέλεσης περιεκτών το οποίο χρησιμοποιείται από το kubelet για την έναρξη/διακοπή περιεκτών, εξαγωγή εικόνων κ.λπ. Το K8s παρέχει έναν μηχανισμό επέκτασης που ονομάζεται Container Runtime Interface (CRI) ο οποίος εκθέτει μια διεπαφή για όλα τα είδη περιβάλλοντος εκτέλεσης περιεκτών ώστε να μπορούν να συνδεθούν με αυτόν. Τη στιγμή που γράφεται αυτό το κείμενο, το containerd είναι το πιο δημοφιλές πρόγραμμα εκτέλεσης περιεκτών που χρησιμοποιείται στον Κυβερνήτη.

Kube-Proxy Πρόκειται για έναν διαμεσολαβητή δικτύου που χειρίζεται την τοπική δικτύωση της συστάδας διατηρώντας κανόνες δικτύου, φροντίζοντας για την απόκτηση μοναδικών διευθύνσεων IP και την ανακατεύθυνση της κυκλοφορίας κ.λπ.

2.2.3 Βασικές Έννοιες

Ας συνεχίσουμε ρίχνοντας μια ματιά στα πιο συνηθισμένα αντικείμενα API του K8s που θα μας βοηθήσουν να κατανοήσουμε καλύτερα τον τρόπο με τον οποίο αναπτύσσονται και εκτελούνται οι εφαρμογές σε ένα περιβάλλον K8s.

Pods

Ένα Pod είναι η ατομική μονάδα χρονοδρομολόγησης στον Κυβερνήτη, με τον ίδιο τρόπο που είναι ένα VM στην εικονικοποίηση ή ένας περιέκτης στο Docker. Ένα μεμονωμένο Pod λειτουργεί διαφανώς και αποτελείται από έναν ή περισσότερους περιέκτες που μοιράζονται το ίδιο περιβάλλον (χώρος ονομάτων IPC, κοινή μνήμη, volumes, δίκτυο κ.λπ.). Η ανάπτυξη ενός Pod είναι μια ατομική λειτουργία που απαιτεί όλους τους περιέκτες του να γίνουν έτοιμοι πριν θεωρηθεί έτοιμο. Όλοι οι περιέκτες αναπτύσσονται στον ίδιο κόμβο για προφανείς λόγους. Γενικά, τα Pods αντιμετωπίζονται ως αναλώσιμα, προορίζονται να δημιουργηθούν, να "ζήσουν", να "πεθάνουν" και στη συνέχεια να αντικατασταθούν απρόσκοπτα από νέα.

Deployments

Από μόνα τους τα Pods δεν αρκούν σε ένα περιβάλλον μεγάλης κλίμακας, καθώς δεν προσφέρουν δυνατότητες αυτοΐασης (self-healing) ή επεκτασιμότητας (scalability). Το Deployment είναι υπεύθυνο για αυτό, αντικαθιστώντας τα Pods που αποτυγχάνουν με νέα ή αυξάνοντας/μειώνοντας τα Pods με βάση μια καθορισμένη μετρική. Επιπλέον, παρέχει κυλιόμενες ενημερώσεις και επαναφορές σε προηγούμενη έκδοση που επιτρέπουν στο σύστημα αλλαγές μεταξύ εκδόσεων εφαρμογών με μηδενικό χρόνο καθυστέρησης. Αυτό το κάνει αυξάνοντας/μειώνοντας σταδιακά τα Pods των νεότερων ή παλαιοτέρων εκδόσεων, ενώ παράλληλα διατηρεί το ιστορικό όλων των ρυθμίσεων που χρησιμοποιήθηκαν.

Stateful Sets

Τα Deployments προσθέτουν μεγάλη αξία στις εφαρμογές που δεν χρειάζονται να διατηρούν την κατάστασή τους. Ωστόσο, όταν τα Pods αποτυγχάνουν αντικαθίστανται από εντελώς νέα (νέο όνομα, hostname, volume bindings) με αποτέλεσμα την απώλεια της κατάστασης του Pod. Οι εφαρμογές που χρειάζονται να διατηρούν την κατάστασή τους απαιτούν να διατηρείται η κατάσταση του Pod ακόμα και αν ένα Pod αποτύχει. Για να γίνει αυτό, ένα StatefulSet εξασφαλίζει προβλέψιμα και μόνιμα ονόματα Pod και DNS ονόματα, καθώς και ένα μοναδικό σύνολο volumes που παραμένει με το Pod για ολόκληρο τον κύκλο ζωής του.

Services

Ένα Service είναι μια αφαίρεση που επιτρέπει τους χρήστες να εκθέτουν εφαρμογές που εκτελούνται σε ένα σύνολο Pods ως υπηρεσία δικτύου. Για να γίνει αυτό, βρίσκεται μπροστά από τα Pods και παρέχει αξιόπιστο όνομα, IP διεύθυνση και θύρα. Οι πελάτες μπορούν να συνδεθούν στα υποκείμενα Pods χρησιμοποιώντας το Service, το οποίο εξισορροπεί το φορτίο των αιτημάτων προς τα Pods στόχους.

Namespaces

Οι χώροι ονομάτων είναι ένας μηχανισμός που παρέχει απομόνωση των πόρων σε μια ενιαία συστάδα K8s. Είναι ένας τρόπος για την κατάτμηση της συστάδας σε πολλαπλές εικονικές συστάδες και είναι χρήσιμος όταν εφαρμόζουμε ποσοστώσεις πόρων ή πολιτικές ελέγχου πρόσβασης. Είναι σημαντικό να αναφέρουμε ότι δεν μπορεί κάθε μεμονωμένο αντικείμενο να ανήκει σε ένα χώρο ονομάτων. Για παράδειγμα, οι κόμβοι ή τα PersistentVolumes δεν μπορούν να ανήκουν σε κάποιο χώρο ονομάτων και είναι καθολικά.

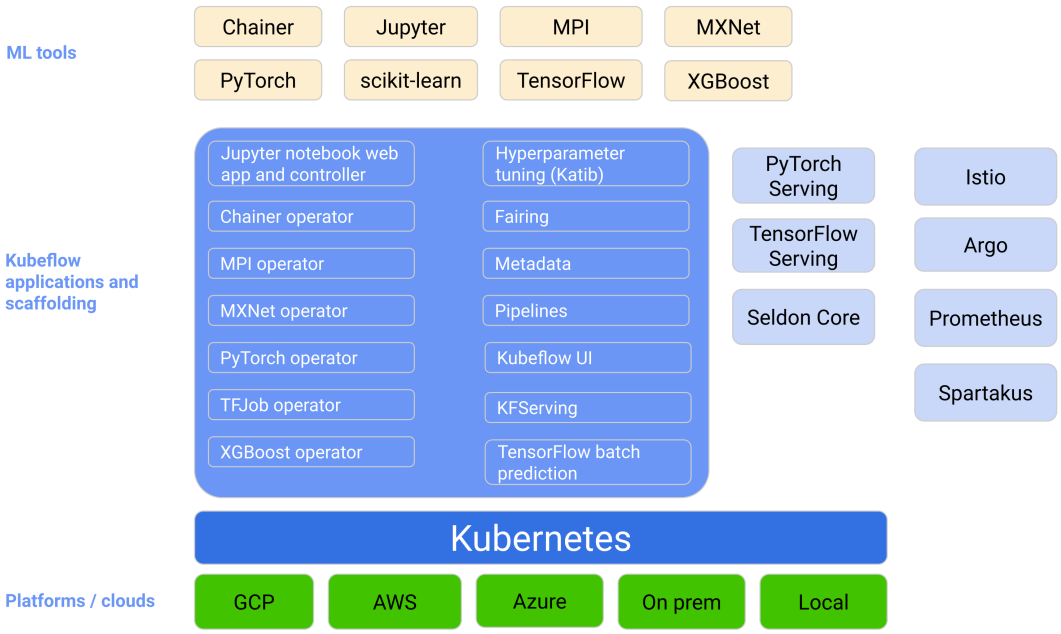
2.3 Kubeflow

Αφού μελετήσαμε τα βασικά στοιχεία του Κυβερνήτη, είναι προφανές ότι προσφέρει διάφορες δυνατότητες και εξαιρετική ευελιξία. Έτσι, έχει αναδειχθεί ως η ντε φάκτο πλατφόρμα για την ανάπτυξη και τη διαχείριση των φορτίων εργασίας σε υποδομές νέφους. Εν τω μεταξύ, τα φορτία εργασίας μηχανικής μάθησης έχουν εξελιχθεί ως τα πιο συνηθισμένα φορτία εργασίας που εκτελούνται σε υποδομές νέφους. Η ανάγκη δημιουργίας ενός οικοσυστήματος που να υποστηρίζει τέτοιου είδους φορτία εργασίας σε ένα περιβάλλον νέφους οδήγησε στο Kubeflow.

2.3.1 Επισκόπηση

Το Kubeflow είναι μια πλατφόρμα σχεδιασμένη για την δημιουργία και την ανάπτυξη ολοκληρωμένων συστημάτων μηχανικής μάθησης. Είναι κατασκευασμένο για επιστήμονες δεδομένων που θέλουν να δημιουργήσουν και να πειραματιστούν με σωληνώσεις

ΜΜ (ML pipelines), καθώς και για μηχανικούς ΜΜ και ομάδες Ops που θέλουν να αναπτύξουν συστήματα ΜΜ σε πολλαπλά περιβάλλοντα (ανάπτυξης, δοκιμών, παραγωγής) [7]. Επιπλέον, είναι ανοιχτού κώδικα, νεφο-εγγενές, καθώς τρέχει πάνω στον Κυβερνήτη, και υποστηρίζει ολόκληρο τον κύκλο ζωής της ΜΜ. Για τον σκοπό αυτό, παρέχει ένα σύνολο εργαλείων ζωτικής σημασίας για την ΜΜ, ενώ παράλληλα τα ενσωματώνει σε ένα κοινόχρηστο περιβάλλον συνεργασίας, το οποίο διαχειρίζεται τα δικαιώματα και εφαρμόζει τον έλεγχο πρόσβασης.



Σχήμα 2.3: *Επισκόπηση Kubeflow*

2.3.2 Βασικές Έννοιες

Όπως αναφέρθηκε προηγουμένως, το Kubeflow αποτελείται από ένα σύνολο νεφο-εγγενών εργαλείων που υποστηρίζουν ολόκληρο τον κύκλο ζωής της ΜΜ. Ας ρίξουμε μια πιο προσεκτική ματιά στα πιο συνηθισμένα από αυτά με σκοπό την καλύτερη κατανόηση της αξίας της πλατφόρμας.

Notebook Servers

Οι Notebook Servers ή Kubeflow Notebooks, όπως επίσης ονομάζονται, παρέχουν διάφορα διαδικτυακά περιβάλλοντα ανάπτυξης, όπως το JupyterLab, το VS Code ή το R

Studio, τα οποία τρέχουν μέσα σε Pods. Αυτοί οι διακομιστές υποστηρίζουν ολόκληρη τη διαδικασία πειραματισμού, ανάπτυξης και εκτέλεσης κώδικα. Για το λόγο αυτό, είναι εξαιρετικά παραμετροποιήσιμοι επιτρέποντας στους χρήστες να καθορίζουν την εικόνα που θέλουν να χρησιμοποιήσουν ή τους πόρους που ανήκουν στο Pod (CPU, μνήμη, τόμοι κ.λπ.).

Pipelines

Το Kubeflow Pipelines (KFP) είναι μια πλατφόρμα σχεδιασμένη για την κατασκευή και την ανάπτυξη φορητών, επεκτάσιμων ροών εργασίας ΜΜ που βασίζονται σε περιέκτες Docker [5]. Είναι ικανή να ενορχηστρώνει σύνθετες σωληνώσεις μηχανικής μάθησης, να εκτελεί και να παρακολουθεί πειράματα καθώς και να να προγραμματίζει επαναλαμβανόμενες εκτελέσεις αυτών των σωληνώσεων. Διαθέτει ένα περιβάλλον εργασίας χρήστη και ένα πλούσιο πελάτη SDK που αλληλεπιδρούν με την πλατφόρμα, καθιστώντας εύκολη για τους χρήστες τη δημιουργία ροών εργασίας από άκρο σε άκρο χρησιμοποιώντας διάφορα εργαλεία.

Serving

Αφού πειραματιστούν με τα Kubeflow Notebooks και Pipelines, οι χρήστες χρειάζονται έναν τρόπο για την αποτελεσματική διάθεση και παρακολούθηση των εκπαιδευμένων μοντέλων τους. Το Kubeflow παρέχει έναν τρόπο διάθεσης των μοντέλων χρησιμοποιώντας τα TFServing, PyTorch, Triton κ.λπ. Σε γενικές γραμμές, μια υπηρεσία εξαγωγής συμπερασμάτων (inference service) δημιουργείται από ένα αποθηκευμένο μοντέλο και γίνεται διαθέσιμη για προβλέψεις. Συμπληρωματικά σε αυτό, μεταδεδομένα και μετρικές διατηρούνται από το σύστημα έτσι ώστε να παρακολουθείται η απόδοση του μοντέλου και η ποιότητα των προβλέψεων. Σε αυτή την προσπάθεια, το KServe είναι η κύρια λύση που ενθυλακώνει περαιτέρω την πολυπλοκότητα της αυτόματης κλιμάκωσης, της δικτύωσης και του ελέγχου υγείας, καθώς και άλλων χαρακτηριστικών αιχμής [6].

Multi-Tenancy

Στο περιβάλλον του Kubeflow όπου πολλοί χρήστες συνεργάζονται στο ίδιο σύνολο πόρων, υπάρχει εγγενής ανάγκη για απομόνωση και ομαδοποίηση αυτών των χρηστών. Η απομόνωση πολλαπλών χρηστών (multi-user isolation) του Kubeflow επιτρέπει στους χρήστες να βλέπουν και να επεξεργάζονται μόνο τους δικούς τους πόρους και τις ρυθμίσεις. Σε υψηλό επίπεδο, οι διαχειριστές δημιουργούν χρήστες και ρυθμίζουν τα δικαιώματα τους σε διαφορετικούς χώρους ονομάτων (namespaces). Στη συνέχεια εφαρμόζεται έλεγχος ταυτότητας με τη χρήση του Istio και του OIDC και η εξουσιοδότηση παρέχεται από το K8s RBAC. Για την επιτυχή πιστοποίηση και εξουσιοδότηση των χρηστών, ο Κυβερνήτης τους παρέχει ένα διακριτικό πρόσβασης μικρής διάρκειας, το οποίο χρησιμοποιούν για να εκτελέσουν τις ενέργειές τους.

2.4 Αποθετήριο Χαρακτηριστικών

Για να γίνει το Kubeflow μια ολοκληρωμένη πλατφόρμα ΜΜ απαιτείται ένα στοιχείο που να είναι υπεύθυνο για τα δεδομένα. Η μηχανική μάθηση είναι ουσιαστικά δεδομένα και κώδικας. Μέχρι τώρα το Kubeflow έχει κάνει βήματα στην παροχή εργαλείων που χειρίζονται κώδικα και κατ' επέκταση τα μοντέλα, αλλά δεν έχει συμβεί το ίδιο για τα δεδομένα. Το αποθετήριο χαρακτηριστικών στοχεύει στη γεφύρωση αυτού του κενού και στην παροχή λύσεων σε προβλήματα δεδομένων που συναντώνται συνήθως στην εφαρμοσμένη ΜΜ.

2.4.1 Επισκόπηση

Ένα αποθετήριο χαρακτηριστικών [25] είναι ένα σύστημα δεδομένων ειδικό για την ΜΜ που στοχεύει να κάνει λειτουργικά και διαθέσιμα στην παραγωγή τα δεδομένα και τα χαρακτηριστικά. Για το λόγο αυτό, προσφέρει διάφορες ενδιαφέρουσες λειτουργίες:

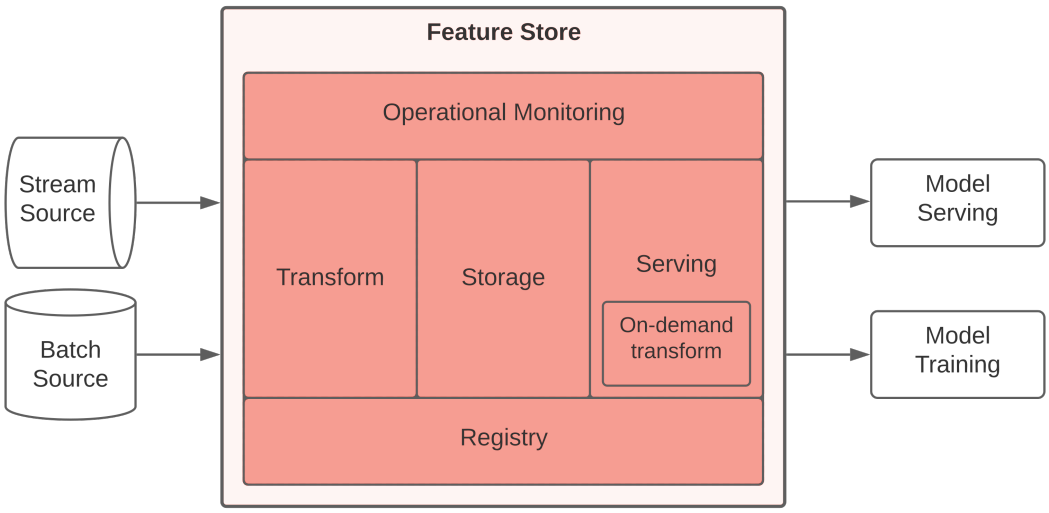
1. Είναι υπεύθυνο για τη διαχείριση και την εκτέλεση σωληνώσεων δεδομένων που μετατρέπουν τα ακατέργαστα δεδομένα σε πολύτιμα χαρακτηριστικά. Αυτό περιλαμβάνει αυτοματοποιημένο υπολογισμό χαρακτηριστικών, backfills και καταγραφή χαρακτηριστικών.

2. Αποθηκεύει και διαχειρίζεται τα δημιουργημένα χαρακτηριστικά και τα σύνολα δεδομένων και παρακολουθεί τις εκδόσεις, τη γενεαλογία και τα σχετικά μεταδεδομένα τους.

3. Παρέχει ένα στρώμα εξυπηρέτησης, ώστε να διαθέτει με συνέπεια τα δεδομένα χαρακτηριστικών τόσο κατά την εκπαίδευση όσο και για τους σκοπούς εξαγωγής συμπερασμάτων. Αυτό το επιτυγχάνει χρησιμοποιώντας δύο συστήματα βάσεων δεδομένων, μια μεγάλη SQL βάση δεδομένων που προορίζεται για εκπαίδευση κατά δέσμες και μια άλλη χαμηλής καθυστέρησης που περιέχει μόνο τα πιο πρόσφατα δεδομένα χαρακτηριστικών, τα οποία προορίζονται για αστραπιαία εξαγωγή συμπερασμάτων.

4. Προσφέρει ένα κεντρικό μητρώο όπου διατηρούνται οι ορισμοί χαρακτηριστικών και μπορούν να διαμοιραστούν και να ανακαλυφθούν από πολλούς χρήστες.

5. Παρακολουθεί τα χαρακτηριστικά και την ποιότητά τους στα συστήματα παραγωγής διασφαλίζοντας ότι δεν υπάρχει απόκλιση (data drift). Η ποιότητα των δεδομένων έχει εξαιρετική αξία στα συστήματα ΜΜ, καθώς είναι άρρηκτα συνδεδεμένη με την απόδοση ενός μοντέλου.



Σχήμα 2.4: *Επισκόπηση Αποθετηρίου Χαρακτηριστικών*

Το τελικό αποτέλεσμα είναι ότι τα χαρακτηριστικά μπορούν πλέον να ορίζονται με τυποποιημένο τρόπο, να καταχωρούνται σε ένα κεντρικό αποθετήριο, να αποθηκεύονται και να είναι προσβάσιμα για την εκπαίδευση και την εξαγωγή συμπερασμάτων από πολλούς χρήστες.

2.4.2 Βασικά Στοιχεία

Προχωρώντας, είναι σημαντικό να κατανοήσουμε λεπτομερώς τον σκοπό των επιμέρους στοιχείων ενός αποθετηρίου χαρακτηριστικών και τις λειτουργίες που αυτά υλοποιούν. Ας τα εξετάσουμε ένα προς ένα!

Μετασχηματισμός

Η πλειονότητα των έργων ΜΜ εκτελεί κάποιου είδους ροή εργασίας δημιουργίας χαρακτηριστικών. Αυτό σημαίνει συλλογή ακατέργαστων δεδομένων, επικύρωση και μετατροπή τους σε πολύτιμα χαρακτηριστικά. Τα ακατέργαστα δεδομένα μπορούν να συλλεχθούν από διάφορους τύπους πηγών, όπως πηγές δέσμης (αποθήκη δεδομένων, βάση δεδομένων κ.λπ.), πηγές ροής (Kafka, Kinesis κ.λπ.) ή ακόμη και δεδομένα κατά τον χρόνο αιτήματος (δεδομένα που συλλέγονται κατά τη στιγμή της πρόβλεψης). Εν τω μεταξύ, τα μοντέλα πρέπει να έχουν πρόσβαση σε νέες τιμές χαρακτηριστικών ώστε να βελτιώνουν τις προβλέψεις τους.

Οι μετασχηματισμοί που διαχειρίζεται και εκτελεί το αποθετήριο χαρακτηριστικών διασφαλίζουν ότι τα νέα δεδομένα επεξεργάζονται και μετατρέπονται σε νέες φρέσκες τιμές χαρακτηριστικών. Αυτό μπορεί να συμβαίνει σε υπάρχοντα δεδομένα που βρίσκονται σε μια αποθήκη δεδομένων ως μέρος μιας εργασίας backfill (μετασχηματισμός batch) ή σε πηγές ροής που πρέπει να συγκεντρώσουν τιμές για μια χρονική περίοδο (μετασχηματισμός streaming). Μπορούν επίσης να εφαρμοστούν σε δεδομένα που ζητούνται σε χρόνο αιτήματος (μετασχηματισμός on-demand), π.χ. μετατροπή των συντεταγμένων GPS ενός χρήστη κατά τη στιγμή του αιτήματος σε πραγματική τοποθεσία. Η ύπαρξη ενός συνεπούς τρόπου ορισμού και επαναχρησιμοποίησης των μετασχηματισμών έχει μεγάλη αξία, καθώς διασφαλίζει ότι τα μοντέλα λαμβάνουν συνεπή δεδομένα χαρακτηριστικών.

Αποθήκευση

Όπως υποδηλώνει το όνομά του, ένα αποθετήριο χαρακτηριστικών (feature store) περιλαμβάνει ή τουλάχιστον ενσωματώνει ένα σύστημα αποθήκευσης. Η αποθήκευση χαρακτηριστικών υποστηρίζει την ανάκτηση τους μέσω του στρώματος διάθεσης. Όπως αναφέρθηκε προηγουμένως, ένα τυπικό αποθετήριο χαρακτηριστικών περιέχει ένα

offline και ένα online επίπεδο αποθήκευσης.

Ο offline αποθηκευτικός χώρος χρησιμοποιείται για την αποθήκευση μεγάλων ποσοτήτων ιστορικών δεδομένων χρονοσειράς (δεδομένα με χρονοσφραγίδες) και είναι συνήθως μια γνωστή αποθήκη ή λίμνη δεδομένων, όπως το S3 ή το BigQuery. Αυτός ο τεράστιος όγκος δεδομένων χρησιμοποιείται αργότερα για την εκπαίδευση μοντέλων.

Από την άλλη πλευρά, το online επίπεδο αποθήκευσης περιέχει τις τελευταίες τιμές χαρακτηριστικών, οι οποίες θα χρησιμοποιηθούν για εξαγωγή συμπερασμάτων με χαμηλή καθυστέρηση. Συνήθως υλοποιείται από έναν αποθηκευτικό χώρο κλειδιών-τιμών όπως το DynamoDB ή το Redis.

Διάθεση

Ένα αποθετήριο χαρακτηριστικών είναι υπεύθυνο για τη διάθεση των χαρακτηριστικών σε εφαρμογές και μοντέλα με συνεπή τρόπο κατά τη διάρκεια της εκπαίδευσης και της εξαγωγής συμπερασμάτων. Προκειμένου ένα μοντέλο να αξιοποιήσει πλήρως τις δυνατότητές του και να αποφύγει το πολύ κοινό πρόβλημα της στρέβλωσης μεταξύ εκπαίδευσης και διάθεσης ενός μοντέλου [24], τα χαρακτηριστικά που χρησιμοποιούνται για την εκπαίδευση ενός μοντέλου πρέπει να ταιριάζουν ακριβώς με τα χαρακτηριστικά που παρέχονται για την εξαγωγή συμπερασμάτων.

Το στρώμα διάθεσης αφαιρεί τα πολύπλοκα ερωτήματα που απαιτούνται για την εξαγωγή τιμών χαρακτηριστικών με ορθότητα συγκεκριμένης χρονικής στιγμής (point-in-time correctness) και παρέχει έναν ομοιόμορφο τρόπο πρόσβασης σε αυτά τα χαρακτηριστικά από οπουδήποτε χρησιμοποιώντας ένα SDK. Με βάση τις απαιτήσεις επιδόσεων χρησιμοποιούνται διαφορετικά APIs για την επίτευξη των απαιτούμενων αποτελεσμάτων. Υλοποιήσεις του ίδιου API σε διαφορετικές γλώσσες ή πλαίσια (framework) βελτιώνουν επίσης τις επιδόσεις.

Μητρώο

Ένα κεντρικό μητρώο είναι βασικό συστατικό ενός αποθετηρίου χαρακτηριστικών. Αποθηκεύει τους ορισμούς των χαρακτηριστικών και τα σχετικά μεταδεδομένα με τυποποιημένο τρόπο και λειτουργεί ως η πηγή της αλήθειας. Οι επιστήμονες δεδομένων και

οι ομάδες το χρησιμοποιούν για να μοιράζονται και να ανακαλύπτουν νέα χαρακτηριστικά, καθιστώντας το ένα εργαλείο συνεργασίας. Όλες οι ρυθμίσεις αποθηκεύονται στο μητρώο, επομένως όλες οι λειτουργίες συμβουλεύονται το μητρώο πριν από την εκτέλεση οποιασδήποτε ενέργειας. Για παράδειγμα, τα API διάθεσης χαρακτηριστικών χρησιμοποιούν το μητρώο για να καταλάβουν ποιες τιμές χαρακτηριστικών πρέπει να επιστρέψουν ή πού να τις βρουν σε μια αποθήκη δεδομένων.

Όσον αφορά τα αποθηκευμένα μεταδεδομένα, το μητρώο διατηρεί διάφορες πληροφορίες που σχετίζονται με τους ορισμούς των χαρακτηριστικών, όπως ο ιδιοκτήτης, η περιγραφή, οι πληροφορίες σχετικές με τον τομέα του χαρακτηριστικού και η γενεαλογία. Αυτό ανοίγει ένα ολόκληρο χώρο παρακολούθησης και ελέγχου δεδομένων, καθώς και εντοπισμού της γενεαλογίας και ιδιοκτησίας του. Γενικά, τα αποθετήρια χαρακτηριστικών έχουν σχεδιαστεί για να αλληλεπιδρούν με εξωτερικά συστήματα ή στοιχεία που αξιοποιούν τα μεταδεδομένα και παρέχουν χρήσιμες πληροφορίες.

Παρακολούθηση

Ένα αποθετήριο χαρακτηριστικών φαίνεται να βρίσκεται στην ιδανική θέση για την παρακολούθηση των δεδομένων. Δεδομένου ότι είναι το ενδιάμεσο επίπεδο μεταξύ δεδομένων και κώδικα, μπορεί να υπολογίζει μετρικές και να παρακολουθεί την ορθότητα και την ποιότητα, ειδοποιώντας τους επιστήμονες δεδομένων εάν εμφανιστεί ανεπιθύμητη συμπεριφορά. Τα προβλήματα των δεδομένων είναι στις περισσότερες περιπτώσεις ο λόγος που τα συστήματα ΜΜ αποτυγχάνουν να αποδώσουν σύμφωνα με τις προσδοκίες, επομένως η παρακολούθηση της απόκλισης δεδομένων, της στρέβλωσης μεταξύ εκπαίδευσης και διάθεσης ενός μοντέλου, καθώς και η επικύρωση των δεδομένων των χαρακτηριστικών μπορεί να ελαχιστοποιήσει αυτά τα προβλήματα.

Επιπλέον, ένα αποθετήριο χαρακτηριστικών μπορεί να παρακολουθεί μετρικές λειτουργίας όπως η καθυστέρηση, η διεκπεραιωτικότητα ή ο χρόνος επεξεργασίας. Όλες αυτές οι μετρήσεις μπορούν στη συνέχεια να χρησιμοποιηθούν από εξωτερικά εργαλεία παρακολούθησης ή παρατηρησιμότητας και να παρέχουν επιπλέον ορατότητα στα χαρακτηριστικά και τα μοντέλα που τα χρησιμοποιούν. Και πάλι, είναι πολύ συνηθισμένο για τα αποθετήρια χαρακτηριστικών να αλληλεπιδρούν εύκολα με εξωτερικά συστήματα ή στοιχεία, όπως τα προαναφερθέντα.

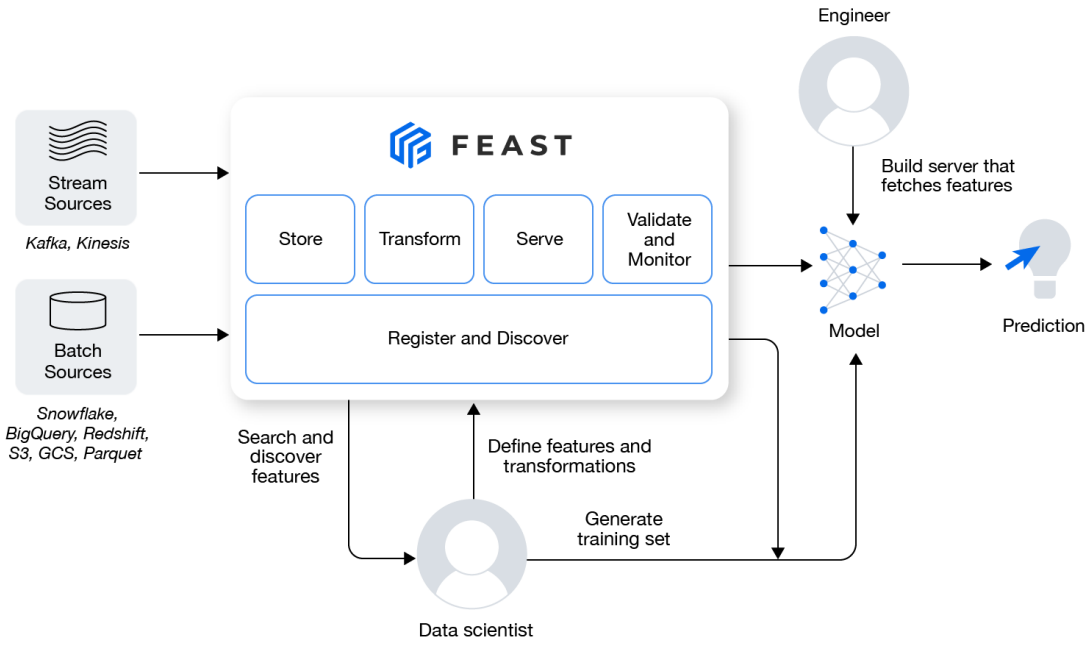
3 Σχεδίαση

Το παρόν κεφάλαιο επικεντρώνεται στην παροχή μιας πλήρους εικόνας του νέου σχεδιασμού με βάση την προτεινόμενη λύση (βλ. ενότητα 1.4). Αρχικά, περιγράφει μια υψηλού επιπέδου επισκόπηση των στοιχείων, των βασικών εννοιών και λειτουργιών του Feast. Στη συνέχεια, παρουσιάζει τεχνικές λεπτομέρειες σχετικές με το μητρώο του Feast και εκθέτει τις υπάρχουσες προβληματικές σχεδιάσεις. Τέλος, παρουσιάζει λεπτομερώς τη νέα προτεινόμενη αρχιτεκτονική, εστιάζοντας στις αλλαγές και τα νέα στοιχεία.

3.1 Αποθετήριο Χαρακτηριστικών Feast

Το Feast είναι ένα αποθετήριο χαρακτηριστικών ανοικτού κώδικα. Είναι ένα λειτουργικό σύστημα δεδομένων για τη διαχείριση και την παροχή χαρακτηριστικών μηχανικής μάθησης σε μοντέλα σε περιβάλλον παραγωγής. Το Feast είναι σε θέση να διαθέτει δεδομένα χαρακτηριστικών στα μοντέλα από ένα online store χαμηλής καθυστέρησης (για προβλέψεις σε πραγματικό χρόνο) ή από ένα offline store (για εκπαίδευση μοντέλων) [3]. Εκτός αυτού, είναι ένα σύστημα που καθιστά τα δεδομένα χαρακτηριστικών συνεπή και εύκολα διαθέσιμα σε πολλαπλά περιβάλλοντα (ανάπτυξη, παραγωγή) και ομάδες χρηστών, ενώ παράλληλα παρέχει ένα κεντρικό μητρώο που επιτρέπει την επαναχρησιμοποίηση χαρακτηριστικών.

Το Feast φαίνεται να μοιάζει πολύ με ένα κοινό αποθετήριο χαρακτηριστικών, αλλά τη στιγμή της συγγραφής λείπουν ακόμη σημαντικά μέρη του (μετασχηματισμοί, επικύρωση, παρακολούθηση, ανακάλυψη χαρακτηριστικών). Δεδομένου ότι πρόκειται για ένα



Σχήμα 3.1: *Επισκόπηση Feast*

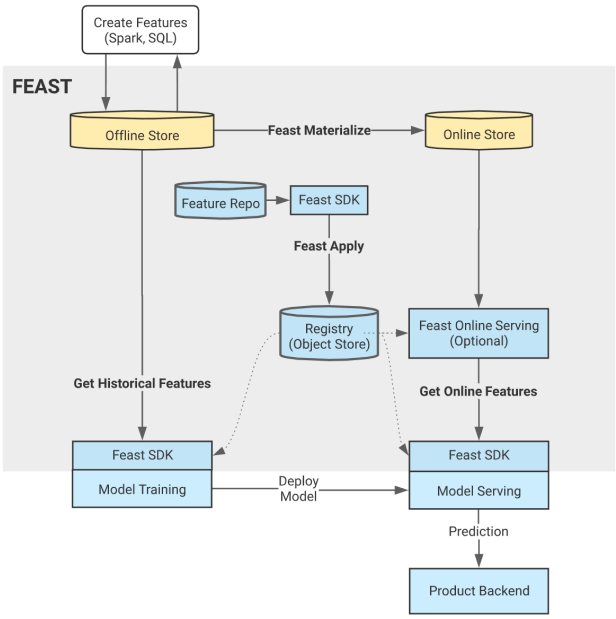
έργο ανοικτού κώδικα, στόχος του δεν είναι η δημιουργία λύσεων από το μηδέν, αλλά η ανάπτυξη αφαιρέσεων και μηχανισμών προσθήκης (plugin) που θα του επιτρέψουν να ενσωματωθεί σε άλλα συστήματα απρόσκοπτα. Αυτή η προσέγγιση το καθιστά μοναδικό και εξαιρετικά ευέλικτο μεταξύ άλλων λύσεων, καθώς δεν είναι συγκεκριμένο για κάποιον προμηθευτή υπηρεσιών νέφους ή κάποιο πλαίσιο και ούτε απαιτεί ειδική υποδομή για να λειτουργήσει.

3.1.1 Αρχιτεκτονική

Η αρχιτεκτονική του Feast είναι αρκετά απλοϊκή και περιέχει 3 κύρια στοιχεία:

1. Offline Store
2. Online Store
3. Registry

Εκτός από αυτά, υπάρχουν στοιχεία ή στρώματα που ενισχύουν τη χρήση του όπως ένα feature repository τύπου git, ένας feature server ή ένας provider. Συνεχίζοντας, όλα τα μέρη της αρχιτεκτονικής του θα παρουσιαστούν λεπτομερώς.



Σχήμα 3.2: *Αρχιτεκτονική Feast*

Offline Store

Ένα Offline Store αναφέρεται στην πραγματική πηγή δεδομένων που περιέχει ιστορικές τιμές χαρακτηριστικών μαζί με την χρονοσφραγίδα τους. Αυτή η πηγή δεδομένων είναι συνήθως μια αποθήκη δεδομένων/λίμνη δεδομένων (BigQuery της Google, Redshift της Amazon κ.λπ.) ή απλώς μια βάση δεδομένων (π.χ. PostgreSQL). Υποστηρίζει επίσης πηγές ροής δεδομένων (π.χ. ροές Kafka ή Kinesis). Πρόκειται για το στοιχείο του οποίου σκοπός είναι να προσφέρει δεδομένα χαρακτηριστικών για εκπαίδευση μοντέλων και να παρέχει τις τελευταίες τιμές χαρακτηριστικών στο Online Store κατά τη διαδικασία materialization.

Γενικά, το Feast δεν είναι ένα σύστημα αποθήκευσης, αλλά αντίθετα χρησιμοποιεί το Offline Store ως διεπαφή για την αναζήτηση των υπαρχουσών τιμών χαρακτηριστικών. Απαιτείται ελάχιστη ρύθμιση για τον ορισμό του προτιμώμενου Offline Store. Ακολουθεί ένα παράδειγμα:

```
offline_store:
    type: bigquery
    dataset: feast_bq_dataset
    project_id: feast-oss
    location: EU
```

Online Store

Ένα Online Store είναι συνήθως μια γρήγορη και κλιμακώσιμη βάση δεδομένων ή μια αποθήκη κλειδιών-τιμών (Datastore της Google, DynamoDB της Amazon, Redis κ.λπ.) που διαχειρίζεται το Feast. Είναι το στοιχείο που είναι υπεύθυνο για την παροχή μόνο των πιο πρόσφατων τιμών χαρακτηριστικών και χρησιμοποιείται κατά την εξαγωγή συμπερασμάτων.

Οι πιο πρόσφατες τιμές χαρακτηριστικών φορτώνονται στο Online Store από τις πηγές δεδομένων των Feature Views κατά τη διαδικασία materialization. Το σχήμα του Online Store αντικατοπτρίζει το σχήμα των πηγών δεδομένων που χρησιμοποιούνται για τον εμπλουτισμό του. Θα κάνουμε μια πιο λεπτομερή περιγραφή του τρόπου λειτουργίας αυτής της διαδικασίας στη συνέχεια (βλ. ενότητα 3.1.3). Και πάλι, ο ορισμός ενός Online Store απαιτεί ελάχιστη προσπάθεια:

```
online_store:
  type: redis
  redis_type: redis_cluster
  connection_string: "redis1:6379,redis2:6379,password=my_password"
```

Registry

Ένα Registry (μητρώο) είναι ένα πλήρως διαχειριζόμενο στοιχείο του Feast, που σημαίνει ότι το Feast είναι υπεύθυνο για τη δημιουργία, την ενημέρωση και την καταστροφή του. Παραπέμπει σε έναν χώρο αποθήκευσης αντικειμένων (π.χ. GCS bucket, S3 bucket) που διατηρεί το αρχείο του μητρώου σε μορφή proto. Είναι το στοιχείο που περιέχει όλους τους ορισμούς χαρακτηριστικών και τα σχετικά μεταδεδομένα τους. Είναι σαν ένας κατάλογος χαρακτηριστικών και λειτουργεί ως η μοναδική πηγή αλήθειας για αυτά. Κάθε ανάπτυξη (deployment) του Feast έχει ένα μόνο Registry και κάθε φορά που ανακτώνται τιμές χαρακτηριστικών είτε από το Offline ή από το Online Store ή εκτελείται κάποια άλλη λειτουργία, το Registry ενεργεί ως σύμβουλος που παρέχει χρήσιμα μεταδεδομένα.

Το αρχείο του μητρώου που ουσιαστικά αποθηκεύεται, είναι μια αναπαράσταση των μεταδεδομένων του Feast σε Protobuf μορφή [17]. Σε μια φιλική προς το χρήστη μορφή το μητρώο μοιάζει με το ακόλουθο:

```
{
  "entities": [
    {
      "spec": {
        "name": "driver",
        "valueType": "STRING",
        "joinKey": "driver_id"
      },
      "meta": {}
    }
  ],
  "featureViews": [
    {
      "spec": {
        "name": "driver_activity",
        "entities": [ "driver" ],
        "features": [
          {
            "name": "trips_today",
            "valueType": "INT64"
          },
          {
            "name": "rating",
            "valueType": "FLOAT32"
          }
        ],
        "ttl": "3600s",
        "batchSource": {
          "type": "BATCH_BIGQUERY ",
          "eventTimestampColumn": "timestamp",
          "bigQueryOptions": {
            "table": "feast-oss.demo_data.driver_activity"
          },
          "dataSourceClassType":
            "feast.infra.offline_stores.bigquery_source.BigQuerySource"
        },
        "online": true
      },
      "meta": {
        "materializationIntervals": [
          {
            "startTime": "2021-11-17T15:00:00Z",
            "endTime": "2021-11-17T17:01:00Z"
          }
        ]
      }
  }
}
```

Το παραπάνω παράδειγμα παρουσιάζει μια ελαχιστοποιημένη έκδοση των περιεχομένων του αρχείου του μητρώου. Ωστόσο, η πλήρης έκδοσή του περιέχει τα ακόλουθα αντικείμενα:

- Data Sources
- Entities
- Feature Services
- Feature Views
- On Demand Feature Views
- Request Feature Views
- Saved Datasets

Θα εξηγήσουμε περισσότερα για αυτά τα αντικείμενα στην επόμενη ενότητα (βλ. ενότητα 3.1.2).

Provider

Ο Provider είναι ένα διαφανές στοιχείο του Feast. Είναι προαιρετικό και οι χρήστες μπορούν να το ορίσουν ρητά στις ρυθμίσεις. Σκοπός του είναι να ενεργεί σαν ένα είδος μεσάζοντα μέσω του οποίου εκτελούνται όλες οι λειτουργίες του Feast. Λειτουργίες όπως η ανάκτηση χαρακτηριστικών ή η διαχείριση της υποδομής κ.λπ. περνούν πρώτα από τον Provider που εκτελεί οποιαδήποτε προσαρμοσμένη λογική. Επιπλέον, στοχεύει σε συγκεκριμένα περιβάλλοντα (GCP, AWS, Local) και είναι υπεύθυνος για την απρόσκοπτη ενορχήστρωση των στοιχείων του Feast σε αυτά. Για παράδειγμα, ένας AWS Provider διασφαλίζει ότι το Registry θα δημιουργηθεί σε ένα S3 bucket και πουθενά αλλού.

Feature Repository

Πρόκειται για ένα αποθετήριο git που περιέχει ένα αρχείο ρυθμίσεων feature_store.yaml και όλους τους ορισμούς χαρακτηριστικών (.py αρχεία). Η ύπαρξη ενός κεντρικού αποθετηρίου όπου οι χρήστες μπορούν να διαχειρίζονται τις ρυθμίσεις και τους ορισμούς

των χαρακτηριστικών, γραμμένους δηλωτικά και αποθηκευμένους σε κώδικα Python, δημιουργεί μια ισχυρή πηγή αλήθειας. Το Feature Repository υποδεικνύει την επιθυμητή κατάσταση του αποθετηρίου χαρακτηριστικών και χρησιμοποιείται για τη ρύθμιση, την ανάπτυξη και τη διαχείρισή του. Τελικά, όλες οι ρυθμίσεις και οι ορισμοί μεταφέρονται και αποθηκεύονται στο Registry, το οποίο είναι προγραμματιστικά προσβάσιμο από άλλα στοιχεία του Feast. Ακολουθεί ένα παράδειγμα δομής του:

```
$ tree -a
.
├── data
│   └── driver_stats.parquet
├── driver_features.py
├── feature_store.yaml
└── .feastignore
```

Feature Server

Το Feast επιτρέπει τη δημιουργία Feature Servers που δημιουργούν τερματικά σημεία HTTP τα οποία παρέχουν τιμές χαρακτηριστικών σε μορφή JSON. Αυτή η λειτουργικότητα επιτρέπει στους χρήστες να διαβάζουν χαρακτηριστικά από το Online Store χρησιμοποιώντας οποιαδήποτε γλώσσα προγραμματισμού που μπορεί να εκτελέσει αιτήματα HTTP. Στην πραγματικότητα, ένας Feature Server διαβάζει περιοδικά το αρχείο του μητρώου και το διατηρεί στην κρυφή μνήμη (cache) του. Με αυτόν τον τρόπο είναι σε θέση να ανταποκρίνεται γρήγορα στα εισερχόμενα αιτήματα, εκτελώντας ερωτήματα απευθείας στο Online Store.

3.1.2 Βασικές Έννοιες

Ακολουθεί μια επισκόπηση των βασικών εννοιών και της ειδικής ορολογίας του Feast. Αυτή είναι μια σημαντική ενότητα που παρέχει λεπτομέρειες για τα περισσότερα από τα αντικείμενα πρώτης κατηγορίας του μητρώου και για το ποιος είναι ο σκοπός τους.

Project

Τα Projects είναι ένας τρόπος διαχωρισμού ονομάτων ή ομαδοποίησης διαφορετικών ειδών αντικειμένων του μητρώου. Ένα Project σχετίζεται λογικά με ένα ή περισσότερα Feature Views, Data Sources, Entities κ.λπ. Παρέχει επίσης ένα πλήρες επίπεδο απομόνωσης σε επίπεδο υποδομής με την ονοματοθεσία πόρων, όπως στους πίνακες, οι οποίοι χρησιμοποιούν το όνομα του Project ως πρόθεμα. Με άλλα λόγια, δημιουργεί ένα εντελώς ξεχωριστό σύμπαν αντικειμένων.

Data Source

Ένα Data Source είναι ένα αντικείμενο που αναφέρεται σε ακατέργαστα υποκείμενα δεδομένα, όπως ένας πίνακας SQL, μια ροή δεδομένων ή ένα μεμονωμένο αρχείο. Δεδομένου ότι το Feast απαιτεί δεδομένα με χρονοσφραγίδες για την εκτέλεση των λειτουργιών του, όλες οι υποστηριζόμενες πηγές δεδομένων πρέπει να περιέχουν ένα πεδίο χρονοσφραγίδας μαζί με κάθε γραμμή, καταχώρηση ή συμβάν ροής. Ήδη διάφορες πηγές δεδομένων δέσμης (Snowflake, BigQuery, Redshift κ.λπ.) και πηγές ροής (Kafka, Kinesis κ.λπ.) υποστηρίζονται από το Feast, αλλά επιτρέπει επίσης στους χρήστες να προσθέσουν υποστήριξη για προσαρμοσμένες πηγές χρησιμοποιώντας το μηχανισμό πρόσθετων στοιχείων του (plugin mechanism).

```
# Data Source definition
  driver_stats_source = BigQuerySource(
    table_ref="feast-oss.demo_data.driver_hourly_stats",
    event_timestamp_column="datetime",
    created_timestamp_column="created",
  )
```

Entity

Γενικά, μια οντότητα είναι μια συλλογή από σημασιολογικά συναφή χαρακτηριστικά. Στον κόσμο του Feast ένα Entity είναι μέρος ενός Feature View και ορίζεται από ένα όνομα, έναν τύπο τιμής και ένα σύνολο κλειδιών. Τα κλειδιά είναι σημαντικά, καθώς το Feast τα χρησιμοποιεί κατά την αναζήτηση τιμών χαρακτηριστικών από το Online Store και κατά τη διαδικασία συνένωσης σε point-in-time joins. Μπορούν εύκολα να

συγκριθούν με τα πρωτεύοντα κλειδιά στις βάσεις δεδομένων SQL και περιγράφουν μοναδικά μια εγγραφή ενός Feature View.

```
# Entity definition
  driver = Entity(
    name="driver",
    value_type=ValueType.STRING,
    join_keys=["driver_id"]
  )
```

Feature View

Ένα Feature View είναι μια λογική ομαδοποίηση χαρακτηριστικών (fields ή features), οντοτήτων (entities) και πηγών δεδομένων (data sources). Πιο συγκεκριμένα, αποτελείται από μηδέν ή περισσότερα Entities, ένα ή περισσότερα Features (Fields) και ακριβώς ένα Data Source. Περιγράφει μια προβολή ή ένα υποσύνολο της υποκείμενης πηγής δεδομένων. Είναι το αντικείμενο που στην πραγματικότητα επιτρέπει τη μοντελοποίηση των υφιστάμενων χαρακτηριστικών με συνεπή τρόπο. Τα Features που σχετίζονται με το Feature View είναι ουσιαστικά ιδιότητες ή χαρακτηριστικά του Entity που τα συνοδεύει.

Το Feast χρησιμοποιεί τα Feature Views κατά την ανάκτηση χαρακτηριστικών είτε από το Online ή από το Offline Store. Καθορίζουν επίσης το σχήμα αποθήκευσης στο Online Store, το οποίο είναι ζωτικής σημασίας κατά τη διαδικασία materialization και κατά την εξαγωγή συμπερασμάτων.

```
# Feature View definition
  driver_stats_fv = FeatureView(
    name="driver_activity",
    entities=["driver"],
    schema=[
      Field(name="trips_today", dtype=Int64),
      Field(name="rating", dtype=Float32),
    ],
    source=BigQuerySource(
      table="feast-oss.demo_data.driver_activity"
    )
  )
```

Ένα άλλο ενδιαφέρον είδος Feature View είναι το On Demand Feature View. Επιτρέπει τη χρήση υφιστάμενων χαρακτηριστικών και request time data (δεδομένα διαθέσιμα μόνο κατά το χρόνο του αιτήματος πρόβλεψης) για το μετασχηματισμό και τη δημιουργία νέων χαρακτηριστικών εκείνη την στιγμή. Αυτό συμβαίνει με τον προσδιορισμό μιας συνάρτησης μετασχηματισμού που το Feast εκτελεί τόσο κατά τις ιστορικές όσο και κατά τις ταχύτατες λειτουργίες ανάκτησης χαρακτηριστικών.

```
# On Demand Feature View definition
  @on_demand_feature_view(
    sources=[
      driver_hourly_stats_view,
      input_request
    ],
    schema=[
      Field(name="conv_rate_plus_val1", dtype=Float64),
      Field(name="conv_rate_plus_val2", dtype=Float64)
    ]
  )
  def transformed_conv_rate(features_df: pd.DataFrame)
  -> pd.DataFrame:
    df = pd.DataFrame()
    df["conv_rate_plus_val1"] =
      (features_df["conv_rate"] + features_df["val_to_add"])
    df["conv_rate_plus_val2"] =
      (features_df["conv_rate"] + features_df["val_to_add_2"])
    return df
```

Feature Service

Ένα Feature Service είναι ένα αντικείμενο που αντιπροσωπεύει μια λογική ομάδα χαρακτηριστικών από ένα ή περισσότερα Feature Views. Συνήθως είναι ένας καλός τρόπος για να αντιστοιχήσουμε τα μοντέλα με μια ομάδα χαρακτηριστικών που χρειάζονται για να εκπαιδευτούν ή για να κάνουν μια πρόβλεψη. Αυτό έχει επίσης ως αποτέλεσμα την καλύτερη παρακολούθηση των διαθέσιμων μοντέλων σε περιβάλλον παραγωγής.

```
# Feature Service definition
  driver_stats_fs = FeatureService(
    name="driver_activity",
    features=[
      driver_stats_fv,
      driver_ratings_fv[["lifetime_rating"]]
    ]
  )
```

Dataset

Ένα Dataset επιτρέπει την αποθήκευση ενός συνόλου δεδομένων που περιέχει τόσο οντότητες όσο και δεδομένα χαρακτηριστικών, τα οποία μπορούν να χρησιμοποιηθούν αργότερα για την εκπαίδευση του μοντέλου. Επιπλέον, η ύπαρξη ενός στιγμιότυπου δεδομένων για ένα συγκεκριμένο χρονικό διάστημα βοηθά στην ανάλυση δεδομένων και στην παρακολούθηση της ποιότητας τους. Στο παρασκήνιο, τα μεταδεδομένα του Dataset αποθηκεύονται στο Registry και τα πραγματικά δεδομένα αποθηκεύονται στο Offline Store.

```
# Dataset creation
historical_job = store.get_historical_features(
  features=["driver:avg_trip"],
  entity_df=["driver"],
)

dataset = store.create_saved_dataset(
  from_=historical_job,
  name='training_dataset',
  storage=SavedDatasetBigQueryStorage(
    table_ref='feast-oss.demo_data.driver_activity'
  )
)

dataset.to_df()
```

3.1.3 Χρήση

Σε αυτό το σημείο, όλα είναι έτοιμα για να προχωρήσουμε με τις λειτουργίες του Feast και τους τρόπους με τους οποίους το χρησιμοποιούμε γενικά. Αρχικά, περιγράφουμε την αρχικοποίηση και την καταστροφή του. Στη συνέχεια, παρουσιάζουμε τις τυπικές λειτουργίες ανάκτησης χαρακτηριστικών και τέλος εξηγούμε την διαδικασία materialization λεπτομερώς. Για να γίνει κατανοητός ο τρόπος με τον οποίο συντονίζονται όλα τα στοιχεία του Feast κατά τη διάρκεια αυτών των λειτουργιών θα χρησιμοποιήσουμε ως αναφορά τον πελάτη SDK γραμμένο σε Python.

Συναρτήσεις apply() - teardown()

Όπως αναφέρθηκε προηγουμένως, μπορούμε να βρούμε όλους τους ορισμούς χαρακτηριστικών στο Feature Repository. Για να είναι προσβάσιμοι από άλλα στοιχεία του Feast πρέπει να αποθηκευτούν στο μητρώο. Η διαδικασία ανάλυσης (parsing) του Feature Repository, η μετατροπή του σε αναπαράσταση protobuf και η αποθήκευση του στο μητρώο πραγματοποιείται με την εντολή *feast apply* του CLI. Στα παρασκήνια, μετά την επικύρωση και την ανάλυση του Feature Repository, το Feast καλεί την μέθοδο *apply()* της κλάσης Feature Store.

```
def apply(
  self,
  objects: Union[
    DataSource,
    Entity,
    FeatureView,
    OnDemandFeatureView,
    RequestFeatureView,
    FeatureService,
    List[FeastObject],
  ],
  objects_to_delete: Optional[List[FeastObject]] = None,
  partial: bool = True,
): ...
```

Ουσιαστικά λαμβάνει δύο ορίσματα: αντικείμενα προς δημιουργία ή ενημέρωση και αντικείμενα προς διαγραφή. Αρχικά, αποθηκεύει τα νέα ή προς ενημέρωση αντικείμενα

στο μητρώο, στη συνέχεια διαγράφει τα αντικείμενα που έχουν καταργηθεί και τέλος ενημερώνει την υποδομή. Η τελευταία ενέργεια αφορά κυρίως τη δημιουργία και τη διαγραφή δομών του Online Store, όπου το Feast αποθηκεύει τις πιο πρόσφατες τιμές χαρακτηριστικών.

Η εκτέλεση της *teardown()* πραγματοποιεί την αντίστροφη διαδικασία. Αρχικά, καταστρέφει όλη την υπάρχουσα υποδομή και στη συνέχεια διαγράφει όλα τα αντικείμενα του μητρώου.

Συνάρτηση get_offline_features()

Η ανάκτηση ιστορικών χαρακτηριστικών είναι ίσως η πιο συνηθισμένη λειτουργία του Feast. Ο στόχος της είναι να ενώσει πολλαπλά χαρακτηριστικά από ένα ή περισσότερα Feature Views σε ένα entity data frame με ορθό τρόπο. Ένα entity data frame που λειτουργεί ως είσοδος της συνάρτησης είναι στην πραγματικότητα ένα σύνολο εγγραφών που περιέχει ένα κλειδί (ή κλειδιά) και μια χρονοσφραγίδα για κάθε εγγραφή.

Entity DataFrame

row	event_timestamp	driver_id
0	2021-04-16 20:29:28+00:00	1001
1	2021-04-15 12:29:28+00:00	1003
2	2021-04-17 04:29:28+00:00	1002
3	2021-02-16 10:29:28+00:00	1000

Η έξοδος είναι μια αναπαραγώγιμη κατάσταση των χαρακτηριστικών σε μια συγκεκριμένη χρονική στιγμή.

Joined Training DataFrame

row	event_timestamp	driver_id	conv_rate	acc_rate
0	2021-04-16 20:29:28+00:00	1001	0.675539	0.657475
1	2021-04-15 12:29:28+00:00	1003	0.128302	0.913942
2	2021-04-17 04:29:28+00:00	1002	0.313097	0.770170
3	2021-02-16 10:29:28+00:00	1000	NULL	NULL

Η διαδικασία συμπλήρωσης του data frame εισόδου με τις σωστές τιμές χαρακτηριστικών είναι πολύ απλή. Υποθέστε ότι η υποκείμενη πηγή δεδομένων είναι η ακόλουθη:

row	event_timestamp	driver_id	conv_rate	acc_rate
1804	2021-04-12 07:00:00+00:00	1001	0.373866	0.896520
1443	2021-04-15 07:00:00+00:00	1003	0.675539	0.657475
1082	2021-04-16 07:00:00+00:00	1001	0.128302	0.913942
721	2021-04-16 07:00:00+00:00	1002	0.802812	0.410884
456	2021-03-16 07:00:00+00:00	1000	0.402842	0.510674
360	2021-04-17 07:00:00+00:00	1002	0.313097	0.770170

Για κάθε γραμμή εντός του entity data frame το Feast σαρώνει χρονικά προς τα πίσω από τον χρόνο του γεγονότος μέχρι το μέγιστο του χρόνου TTL. Για παράδειγμα, για τη σειρά 0 (driver_id 1001) σαρώνει την υποκείμενη πηγή δεδομένων και παίρνει τις γραμμές 1804 και 1082. Στη συνέχεια, ενώνει τη σειρά με την πλησιέστερη χρονοσφραγίδα, άρα τη σειρά 1082. Η σειρά 3 (driver_id 1000) έχει παλαιότερη χρονοσφραγίδα από την πιο πρόσφατη εγγραφή του driver_id 1000 στην υποκείμενη πηγή δεδομένων, επομένως συμπληρώνει το αποτέλεσμα με NULL τιμές. Συνοψίζοντας, η διαδικασία join είναι πολύ παρόμοια με μια διατεταγμένη αριστερή ένωση (join), "χαλαρή" όμως, που βασίζεται στις πλησιέστερες χρονοσφραγίδες.

Για την εκτέλεση αυτής της λειτουργίας, το Feast χρησιμοποιεί την *get_historical_features()* μέθοδο της κλάσης FeatureStore.

```
def get_historical_features(
  self,
  entity_df: Union[pd.DataFrame, str],
  features: Union[List[str], FeatureService],
  full_feature_names: bool = False,
) -> RetrievalJob: ...
```

Αυτή η μέθοδος ακολουθεί μια τεμπέλικη προσέγγιση επιστρέφοντας ένα RetrievalJob αντί για τα πραγματικά δεδομένα. Η λήψη των χαρακτηριστικών απαιτεί την εκτέλεση των μεθόδων *to_df()* ή *to_arrow()*.

```
entity_df = pd.read_csv("entity_df.csv")

training_job = store.get_historical_features(
  entity_df=entity_df,
```

```
  features = [
    'driver_hourly_stats:conv_rate',
    'driver_hourly_stats:acc_rate'
  ],
)

training_df = training_job.to_df()
```

Συνάρτηση get_online_features()

Η απευθείας λήψη χαρακτηριστικών είναι η λειτουργία που χρησιμοποιεί το Feast κατά τη διάρκεια της εξαγωγής συμπερασμάτων για την ανάκτηση των τελευταίων τιμών χαρακτηριστικών από το Online Store. Η μέθοδος *get_online_features()* της κλάσης FeatureStore χρησιμοποιείται από τον πελάτη SDK για το λόγο αυτό.

```
def get_online_features(
  self,
  features: Union[List[str], FeatureService],
  entity_rows: List[Dict[str, Any]],
  full_feature_names: bool = False,
) -> OnlineResponse: ...
```

Αυτή τη φορά δεν υπάρχει ανάγκη για χρονοσφραγίδες, οι χρήστες παρέχουν ένα σύνολο κλειδιών και λαμβάνουν πίσω ένα νέο data frame με τις απαιτούμενες τιμές χαρακτηριστικών. Η λειτουργία δεν είναι τεμπέλικη όπως με την *get_historical_features()*. Επιστρέφει τις τιμές των χαρακτηριστικών σε μορφή protobuf και στη συνέχεια ο πελάτης χρησιμοποιεί τις μεθόδους *to_df()* ή *to_dict()* για να τις μετατρέψει σε εύχρηστη μορφή.

```
online_response = fs.get_online_features(
  features=[
    "driver_hourly_stats:conv_rate",
    "driver_hourly_stats:acc_rate"
  ],
  entity_rows=[{"driver_id": 1001}, {"driver_id": 1002}],
)
online_response_dict = online_response.to_dict()
```

Συνάρτηση materialize()

Τέλος, υπάρχει η διαδικασία materialization. Είναι η διαδικασία που είναι υπεύθυνη για την ανανέωση των τιμών των χαρακτηριστικών του Online Store. Ένας χρήστης παρέχει ένα χρονικό διάστημα και καθορίζει ένα σύνολο από Feature Views που θέλει να ανανεώσει.

```
def materialize(
  self,
  start_date: datetime,
  end_date: datetime,
  feature_views: Optional[List[str]] = None,
) -> None: ...
```

Το Feast ζητάει από όλες τις πηγές δέσμης των Feature Views τις πιο πρόσφατες τιμές χαρακτηριστικών μέσα στο καθορισμένο χρονικό διάστημα και τις φορτώνει στο Online Store. Το ερώτημα τεμαχίζει την υποκείμενη πηγή δεδομένων διατηρώντας μόνο το ζητούμενο χρονικό διάστημα, στη συνέχεια ταξινομεί τις εναπομείναντες εγγραφές με βάση τη χρονοσφραγίδα και τέλος κρατάει την πιο πρόσφατη εγγραφή για κάθε μοναδικό κλειδί.

Step 1: Τεμαχισμένες Εγγραφές

row	event_timestamp	driver_id	conv_rate	acc_rate
1804	2021-04-12 07:00:00+00:00	1001	0.373866	0.896520
1082	2021-04-16 07:00:00+00:00	1001	0.128302	0.913942
1443	2021-04-15 07:00:00+00:00	1003	0.675539	0.657475
360	2021-04-17 07:00:00+00:00	1002	0.313097	0.770170
721	2021-04-16 07:00:00+00:00	1002	0.802812	0.410884

Step 2: Ταξινομημένες Εγγραφές

row	event_timestamp	driver_id	conv_rate	acc_rate
1804	2021-04-12 07:00:00+00:00	1001	0.373866	0.896520
1443	2021-04-15 07:00:00+00:00	1003	0.675539	0.657475
1082	2021-04-16 07:00:00+00:00	1001	0.128302	0.913942
721	2021-04-16 07:00:00+00:00	1002	0.802812	0.410884
360	2021-04-17 07:00:00+00:00	1002	0.313097	0.770170

Step 3: Πιο Πρόσφατες Τιμές

driver_id	conv_rate	acc_rate
1003	0.675539	0.657475
1001	0.128302	0.913942
1002	0.313097	0.770170

3.2 Αρχιτεκτονική Μητρώου

Για να προσδιορίσουμε τι λείπει και τι πρέπει να κάνουμε από άποψη σχεδίασης και αρχιτεκτονικής, απαιτείται πλήρης κατανόηση της τρέχουσας αρχιτεκτονικής του μητρώου. Σε αυτή την ενότητα θα επικεντρωθούμε κυρίως σε δύο κλάσεις Python, τις κλάσεις Registry και RegistryStore, του SDK. Αυτές οι κλάσεις είναι υπεύθυνες για τη διαχείριση των αντικειμένων του μητρώου και την αλληλεπίδραση με το επίπεδο αποθήκευσης.

3.2.1 Λειτουργία

Το μητρώο χρησιμοποιείται σχεδόν σε κάθε λειτουργία του Feast. Κάθε φορά που ένας χρήστης χρειάζεται να πάρει δεδομένα χαρακτηριστικών είτε από το Offline ή από το Online Store χρησιμοποιεί έναν ορισμό χαρακτηριστικού για να καθορίσει το χαρακτηριστικό που θέλει να πάρει. Στο παρασκήνιο:

1. Το Feast εκτελεί ένα ερώτημα προς το μητρώο για να ληφθούν όλα τα απαιτούμενα μεταδεδομένα (π.χ. πού βρίσκονται τα πραγματικά δεδομένα του χαρακτηριστικού, ο τύπος του χαρακτηριστικού κ.λπ.).

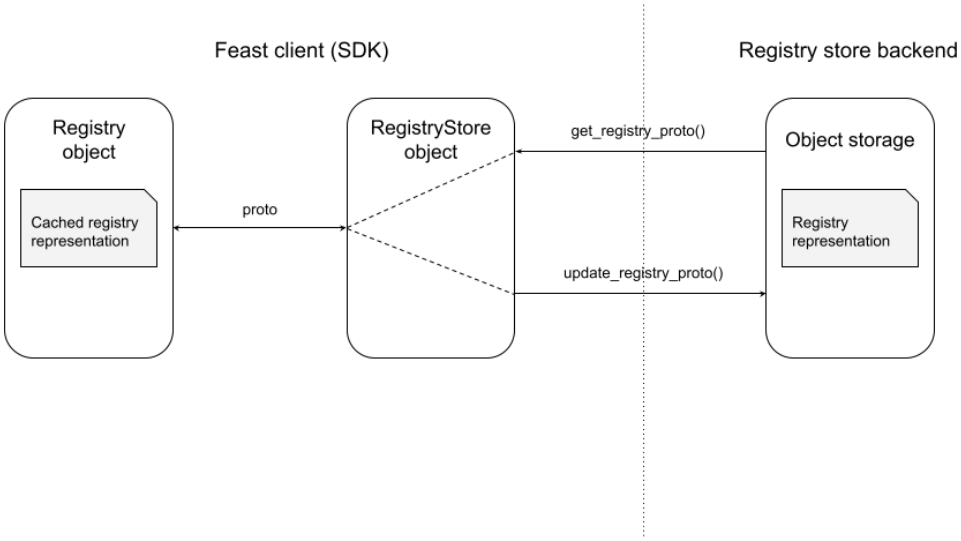
2. Εκτελεί ένα άλλο ερώτημα για την λήψη των πραγματικών δεδομένων χαρακτηριστικών.

Ανεξάρτητα από το αν ένας χρήστης θέλει να πάρει ιστορικά δεδομένα χαρακτηριστικών για να δημιουργήσει ένα σύνολο δεδομένων ή να πάρει online δεδομένα χαρακτηριστικών για να εκτελέσει μια πρόβλεψη, ο πελάτης Feast (SDK) απαιτεί αυτά τα μεταδεδομένα (1). Ωστόσο, αυτές οι δύο λειτουργίες είναι εντελώς αντίθετες:

- Η λήψη ιστορικών δεδομένων χαρακτηριστικών είναι μια λειτουργία που καθορίζεται από το I/O και ξοδεύει τον περισσότερο χρόνο για τη λήψη πραγματικών δεδομένων

- Η λήψη online δεδομένων χαρακτηριστικών είναι μια γρήγορη λειτουργία που είναι ζωτικής σημασίας για υψηλές επιδόσεις στην εξαγωγή συμπερασμάτων

Προκειμένου το Feast SDK να υποστηρίζει εξαγωγή συμπερασμάτων υψηλής απόδοσης, χωρίς να χρειάζεται να αντλεί αυτά τα μεταδεδομένα κάθε φορά, διατηρεί μια προσωρινή αναπαράσταση μητρώου στη μνήμη και φροντίζει να την ανανεώνει όταν λήγει, φέρνοντας την πιο πρόσφατη αναπαράσταση του μητρώου. Για να το κάνει αυτό, χρησιμοποιεί έναν μηχανισμό commit & refresh σε όλες τις λειτουργίες του.



Σχήμα 3.3: *Μηχανισμός Commit & Refresh*

Πριν εξηγήσουμε τον μηχανισμό commit & refresh, ας δούμε τι είναι ένα Registry και ένα RegistryStore αντικείμενο. Είναι αντικείμενα των δύο ακόλουθων κλάσεων που το Feast SDK υλοποιεί:

- **Registry**: βασική κλάση που είναι υπεύθυνη για τη διαχείριση του μητρώου (προσθήκη νέων χαρακτηριστικών, διαγραφή παλιών, κ.λπ.) και τη διατήρηση της προσωρινής αναπαράστασης του μητρώου σε συγχρονισμό με την πραγματική. Υλοποιεί το μηχανισμό commit & refresh.

- **RegistryStore**: επεκτάσιμη κλάση υπεύθυνη για την αλληλεπίδραση με ένα συγκεκριμένο χώρο αποθήκευσης του μητρώου.

Η κλάση Registry περιέχει μεθόδους που εφαρμόζουν αλλαγές στο αρχείο μητρώου. Αυτές παρέχουν ένα boolean όρισμα που ονομάζεται commit. Για παράδειγμα:

```
def apply_entity(
  self,
  entity: Entity,
  project: str,
  commit: bool = True
): ...
  if commit:
    self.commit()
  return

def delete_entity(
  self,
  name: str,
  project: str,
  commit: bool = True
): ...
  if commit:
    self.commit()
  return
```

Αυτό το όρισμα καθορίζει αν μια αλλαγή θα παραμείνει στην κρυφή μνήμη ή θα μεταφερθεί στον αποθηκευτικό χώρο αντικειμένων από το Feast. Στην περίπτωση commit = True, το αντικείμενο τύπου Registry θα στείλει ολόκληρη την αποθηκευμένη αναπαράσταση του μητρώου από την προσωρινή μνήμη στο αντικείμενο τύπου RegistryStore, το οποίο στη συνέχεια θα την προωθήσει στον πραγματικό αποθηκευτικό χώρο αντικειμένων και θα αντικαταστήσει την παλιά αναπαράσταση του μητρώου.

Για τον ίδιο λόγο, οι μέθοδοι της κλάσης Registry που είναι υπεύθυνες για λειτουργίες "ανάγνωσης" παρέχουν την επιλογή allow_cache. Εάν allow_cache = True, το αντικείμενο τύπου Registry επιτρέπεται να χρησιμοποιήσει την αποθηκευμένη στην μνήμη αναπαράσταση του μητρώου. Εάν όχι, πρέπει πρώτα να την ανανεώσει ζητώντας από το RegistryStore να την ανακτήσει από τον αποθηκευτικό χώρο αντικειμένων.

```
def get_entity(
  self,
  name: str,
  project: str,
  allow_cache: bool = False
):
  registry_proto = self._get_registry_proto(allow_cache=allow_cache)
  ...

def list_entities(
  self,
  project: str,
  allow_cache: bool = False
):
  registry_proto = self._get_registry_proto(allow_cache=allow_cache)
  ...
```

Και τα δύο αυτά ορίσματα μαζί με τις μεθόδους commit() και refresh() αποτελούν τον μηχανισμό commit & refresh. Ας ρίξουμε μια πιο προσεκτική ματιά σε αυτές τις μεθόδους.

```
def commit(self):
  if self.cached_registry_proto:
    self._registry_store.update_registry_proto(
      self.cached_registry_proto
    )

def refresh(self):
  self._get_registry_proto(allow_cache=False)

def _get_registry_proto(self, allow_cache: bool = False)
-> RegistryProto:
  ...
  registry_proto = self._registry_store.get_registry_proto()
  ...
```

Και οι δύο αυτές μέθοδοι ουσιαστικά χρησιμοποιούν ένα αντικείμενο τύπου RegistryStore καλώντας τις ακόλουθες δύο μεθόδους που ανήκουν σε αυτό:

- update_registry_proto
- get_registry_proto

3.2.2 Πρόσθετο RegistryStore

Η κλάση RegistryStore είναι ο προγραμματιστικός πελάτης που χρησιμοποιεί το Feast για να αλληλεπιδράσει με τον υποκείμενο αποθηκευτικό χώρο του μητρώου του. Αποτελεί μέρος του μηχανισμού των πρόσθετων που επιτρέπει σε διαφορετικά είδη χώρων αποθήκευσης να λειτουργούν σε συνεργασία με το Feast. Για να γίνω πιο συγκεκριμένος πρόκειται για μια αφηρημένη κλάση που περιέχει τις ακόλουθες 3 μεθόδους:

- get_registry_proto(self) -> RegistryProto
- update_registry_proto(self, registry_proto: RegistryProto)
- teardown(self)

Οι προγραμματιστές που θέλουν να υποστηρίξουν ένα νέο χώρο αποθήκευσης πρέπει να δημιουργήσουν μια νέα κλάση που υλοποιεί αυτές τις μεθόδους. Ακολουθεί ένα παράδειγμα:

```
class GCSRegistryStore(RegistryStore):
  ...
  def get_registry_proto(self):
    ...
    if storage.Blob(bucket=bucket, name=self._blob).
        exists(self.gcs_client):
      self.gcs_client.download_blob_to_file(
        self._uri.geturl(), file_obj, timeout=30
      )
      file_obj.seek(0)
      registry_proto.ParseFromString(file_obj.read())
      return registry_proto
    ...


  def update_registry_proto(self, registry_proto: RegistryProto):
    self._write_registry(registry_proto)
```

```
def teardown(self):
  ...
  gs_bucket = self.gcs_client.get_bucket(self._bucket)
  try:
    gs_bucket.delete_blob(self._blob)
  ...

def _write_registry(self, registry_proto: RegistryProto):
  ...
  gs_bucket = self.gcs_client.get_bucket(self._bucket)
  blob = gs_bucket.blob(self._blob)
  file_obj = TemporaryFile()
  file_obj.write(registry_proto.SerializeToString())
  file_obj.seek(0)
  blob.upload_from_file(file_obj)
```

Είναι σαφές ότι και στις τρεις υλοποιημένες μεθόδους υπάρχει ένας πελάτης GCS ο οποίος είναι υπεύθυνος για τη δημιουργία, ενημέρωση, διαγραφή ή ανάκτηση του αρχείου του μητρώου από τον χώρο αποθήκευσης αντικειμένων (object storage).

3.2.3 Προβλήματα Σχεδίασης

Αφού είδατε πώς λειτουργούν τα πάντα, ήρθε η ώρα να παρουσιάσουμε την σχεδίαση που κάνει την τρέχουσα αρχιτεκτονική δύσκολη στη χρήση μέσα σε περιβάλλον πολλαπλών χρηστών, όπου οι χρήστες πρέπει να μοιράζονται συγκεκριμένα τμήματα του μητρώου. Πριν προχωρήσουμε, είναι σημαντικό να έχουμε κατά νου ότι η αναπαράσταση του μητρώου την οποία μεταφέρει και αποθηκεύει το Feast σε μορφή protobuf περιέχει έναν κατάλογο για κάθε ένα από τα αντικείμενα πρώτης κατηγορίας του μητρώου (βλ. ενότητα 3.1.1).

Τώρα, η αποθήκευση ολόκληρου του μητρώου ως ενιαίο αρχείο δημιουργεί προβλήματα με την κοινή χρήση των αντικειμένων πρώτης κατηγορίας του μητρώου σε περιβάλλον πολλαπλών χρηστών. Δεν υπάρχει σαφής τρόπος ρύθμισης των δικαιωμάτων σε συγκεκριμένα αντικείμενα, καθώς μόνο δικαιώματα ανάγνωσης ή εγγραφής μπορούν να ρυθμιστούν σε επίπεδο αρχείου. Επιπλέον, η μετακίνηση ολόκληρων αρχείων μεταξύ του πελάτη και του χώρου αποθήκευσης αντικειμένων υποβαθμίζει την απόδοση.

Μια άλλη πτυχή του μητρώου που θα μπορούσε ενδεχομένως να προκαλέσει προβλήματα είναι η προσωρινή αναπαράσταση του μητρώου. Σε σενάρια όπου πολλαπλοί χρήστες αλληλεπιδρούν με το ίδιο μητρώο, η διατήρηση του συγχρονισμού στην πλευρά του πελάτη (Feast SDK) διασφαλίζοντας παράλληλα ότι οι αλλαγές που πραγματοποιούνται τοπικά από τους χρήστες βρίσκουν το δρόμο τους προς τον αποθηκευτικό χώρο, αποτελεί πρόκληση. Σε αυτό το σημείο, δεν υπάρχει μηχανισμός κλειδώματος στο "απομακρυσμένο" αρχείο του μητρώου, καθιστώντας αδύνατη την εγγύηση ατομικής πρόσβασης σε ταυτόχρονες εγγραφές.

3.3 Νέα Αρχιτεκτονική

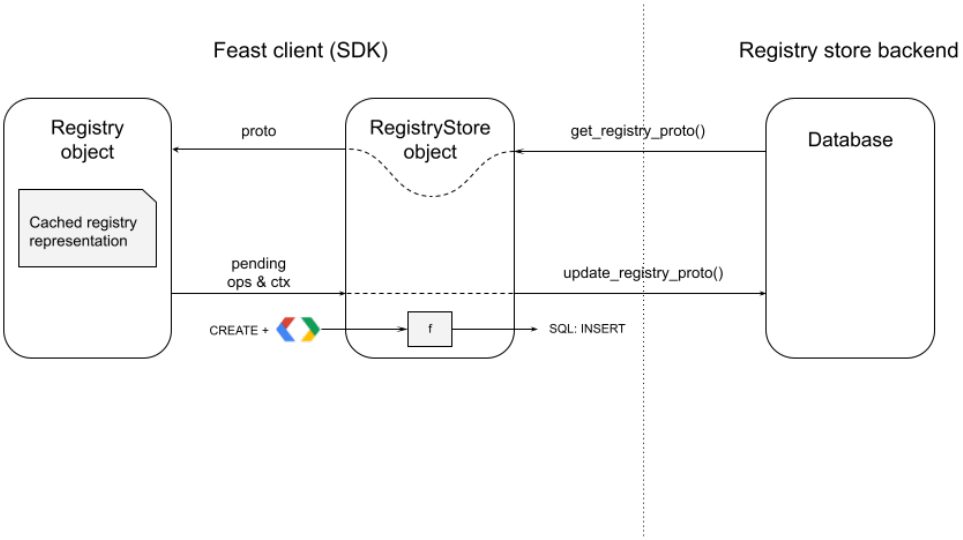
Η επίλυση ή το ξεπέρασμα των παραπάνω προβληματικών σχεδιάσεων είναι απαραίτητη για την ενσωμάτωση του Feast στο Kubeflow. Σε αυτή την ενότητα εκθέτουμε πιθανές νέες σχεδιάσεις και λύσεις. Η εστίαση θα παραμείνει στο μητρώο και τη λειτουργικότητά του.

Αυτή τη στιγμή, το Feast είναι ουσιαστικά ένα Python SDK που διαχειρίζεται και ενορχηστρώνει άλλα στοιχεία, όπως το Offline Store, το Online Store και το Registry. Ωστόσο, και τα τρία αυτά στοιχεία βασίζονται σε υποδομές συγκεκριμένων προμηθευτών, όπως κουβάδες αποθήκευσης ή αποθήκες δεδομένων. Έχοντας ένα έργο ανοιχτού κώδικα όπως το Kubeflow απαιτεί στοιχεία που είναι ανεξάρτητα από τον προμηθευτή και μπορούν να χρησιμοποιηθούν σε διάφορες χρήσεις του Feast.

Ο σκοπός της όλης προσπάθειας είναι να επεκτείνουμε τη λειτουργικότητα και την ευελιξία του Feast για να το ενσωματώσουμε στο Kubeflow. Αυτό θα συμβεί με το σχεδιασμό ενός νέου backend για το μητρώο, ενός πλήρους διακομιστή API, ο οποίος θα προσθέσει ένα επιπλέον στρώμα πάνω από τα αποθηκευμένα αντικείμενα του μητρώου. Προσθήκη του αυτού του στρώματος διαχείρισης θα καταστήσει το Feast ακόμη πιο επεκτάσιμο και θα θέσει τις βάσεις για άλλες βελτιώσεις. Ένα αυτόνομο backend για το μητρώο που υποστηρίζεται από μια βάση δεδομένων SQL, που αποθηκεύει αντικείμενα του και που είναι σε θέση να εκτελεί έλεγχο πρόσβασης είναι ο τελικός στόχος.

3.3.1 Αρχική Σχεδίαση

Η αρχική σχεδίαση αποσκοπεί στην αποφυγή της χρήσης ενός μόνο αρχείου μητρώου ως κεντρικού αποθηκευτικού συστήματος. Έτσι, θα αντικαταστήσουμε τον αποθηκευτικό χώρο αντικειμένων από μια σχεσιακή βάση δεδομένων.



Σχήμα 3.4: *Μητρώο ως Βάση Δεδομένων*

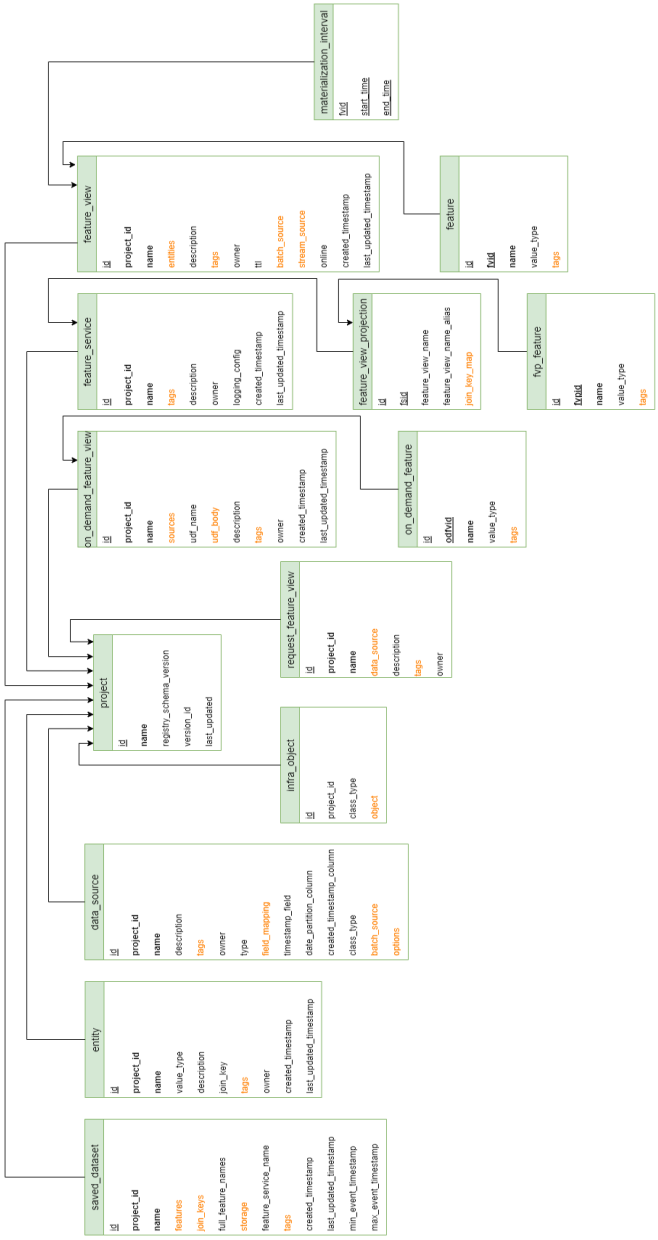
Αυτή η σχεδίαση βελτιώνει επίσης τον τρόπο διαχείρισης της αναπαράστασης του μητρώου, καθώς τα τυπικά ΣΔΒΔ προσφέρουν ιδιότητες ACID χρήσιμες για την αποφυγή συγκρούσεων σε ταυτόχρονες λειτουργίες εγγραφής. Όταν οι χρήστες προσπαθούν να αλλάξουν την κατάσταση του μητρώου ταυτόχρονα, η χρήση των συναλλαγών θα διασφαλίσει ότι κάθε επιτυχής λειτουργία εγγραφής θα παραμείνει στη βάση δεδομένων.

Σε αυτήν την νέα σχεδίαση, η ανάκτηση του μητρώου παραμένει τόσο απλή όσο η εκτέλεση των απαιτούμενων ερωτημάτων SELECT SQL προς τη βάση δεδομένων. Η ενημέρωση του μητρώου είναι λίγο πιο πολύπλοκη. Το αντικείμενο τύπου Registry θα διατηρεί μια λίστα FIFO με τις εκκρεμείς λειτουργίες και το περιεχόμενό τους (σε μορφή protobuf). Αυτή η ουρά περιέχει τις αλλαγές που πρέπει να εκτελεστούν για να ανανεώσει το Feast το μητρώο από το τελευταίο commit. Το περιεχόμενό τους αναφέρεται στο πραγματικό αντικείμενο του μητρώου, σε αναπαράσταση protobuf, πάνω στο οποίο θα εκτελεστεί η λειτουργία. Για παράδειγμα:

```
[("CreateEntity", entityProto), ("DeleteFeatureView", fvProto), ...]
```

Κάθε φορά που το Feast ενεργοποιεί τον μηχανισμό commit, μια παρόμοια λίστα περνάει στο αντικείμενο RegistryStore, το οποίο τη μεταφράζει στα αντίστοιχα ερωτήματα SQL και τα εκτελεί. Η λίστα αδειάζει μετά από κάθε επιτυχές commit.

Σχήμα Βάσης Δεδομένων



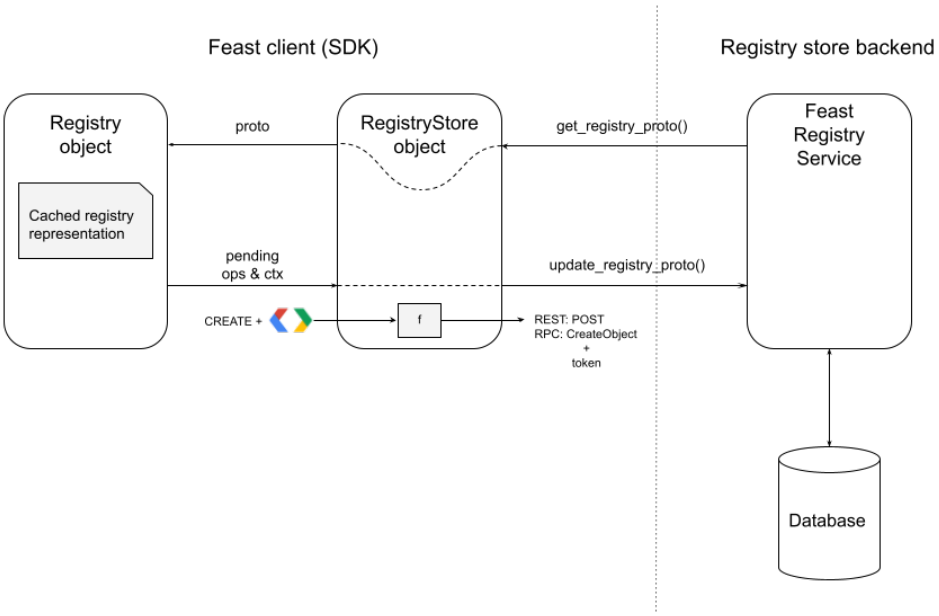
Σχήμα 3.5: *Σχήμα Βάσης Δεδομένων*

Δεδομένου ότι πρόκειται για μια νέα προσέγγιση, υπάρχει ανάγκη να σχεδιαστεί ένα σχήμα βάσης δεδομένων που αποθηκεύει τα αντικείμενα μητρώου. Το νέο σχήμα ακολουθεί τον ορισμό του σχήματος του μητρώου και των στοιχείων του [2], αυτού που χρησιμοποιείται για τα μηνύματα protobuf.

Σχόλια Όλοι οι πίνακες περιέχουν ένα μοναδικό αναγνωριστικό. Επιπλέον, υπάρχουν και άλλοι περιορισμοί μοναδικότητας. Για όλα τα αντικείμενα πρώτης κατηγορίας ο συνδυασμός του project_id και του name πρέπει να είναι μοναδικός. Το ίδιο ισχύει και για τον πίνακα projects, όπου το όνομα του project πρέπει επίσης να είναι μοναδικό. Αποθηκεύουμε τα πεδία που είναι χρωματισμένα με πορτοκαλί χρώμα στην βάση δεδομένων ως bytes καθώς περιέχουν σύνθετα αντικείμενα όπως αντικείμενα JSON ή λίστες.

3.3.2 Βελτιωμένη Σχεδίαση

Έχοντας θέσει ένα στιβαρό υπόβαθρο (μια βάση δεδομένων ως μέσο αποθήκευσης του μητρώου), είναι πλέον δυνατό να δημιουργηθεί ένα άλλο στρώμα πάνω σε αυτό, προκειμένου να διαχειριστούμε τα δικαιώματα σε συγκεκριμένα αντικείμενα του μητρώου.



Σχήμα 3.6: *Νέα Αρχιτεκτονική*

Αυτό λειτουργεί ως ενδιάμεσο στρώμα, μεταξύ της πλευράς του πελάτη και της πραγματικής βάσης δεδομένων, που θα επιβάλλει τον έλεγχο πρόσβασης. Είναι ένας απλός διακομιστής REST API μπροστά από έναν διακομιστή gRPC [13]. Από εδώ και στο εξής αυτό το στρώμα θα ονομάζεται Feast Registry Service (FRS) και έχει τις ακόλουθες τρεις λειτουργίες:

- Έκθεση τελικών σημείων CRUD για όλα τα αντικείμενα πρώτης κατηγορίας του μητρώου.
- Εκτέλεση ελέγχου ταυτότητας και εξουσιοδότησης των αιτημάτων.
- Διαχείριση αλληλεπιδράσεων με τη βάση δεδομένων.

Τα τελικά σημεία CRUD έχουν την ακόλουθη μορφή:

```
POST: /createEntity body: entity_proto
GET: /getEntity body: entity_name, entity_project
POST: /updateEntity body: entity_proto
DELETE: /deleteEntity body: entity_name, entity_project
```

Στην πλευρά του πελάτη, το αντικείμενο τύπου RegistryStore μεταφράζει τον κατάλογο των εκκρεμών λειτουργιών σε κλήσεις API και τις εμπλουτίζει με ένα token. Αυτό το token χρησιμοποιείται στη συνέχεια από το FRS κατά τη διάρκεια ελέγχου ταυτότητας και εξουσιοδότησης.

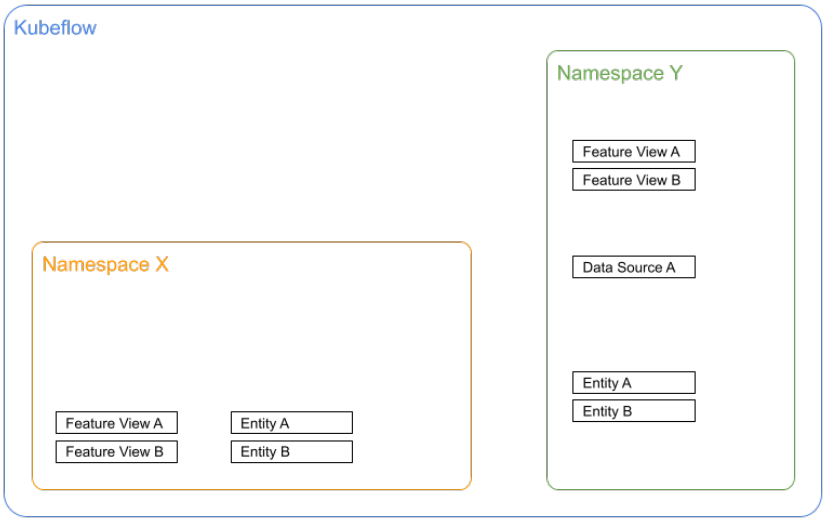
Projects και Απομόνωση

Μέχρι τώρα παρουσιάσαμε όλους τους μηχανισμούς που απαιτούνται για τη διαχείριση των δικαιωμάτων σε αντικείμενα του μητρώου. Ωστόσο, είναι επίσης σημαντικό να εξηγήσουμε τα επίπεδα απομόνωσης που προσφέρουμε σε ένα περιβάλλον Kubeflow. Όπως εξηγήσαμε στο κεφάλαιο Υπόβαθρο (βλ. ενότητα 2.2.3) ο Κυβερνήτης παρέχει έναν μηχανισμό που ονομάζεται Namespaces για την απομόνωση των πόρων. Επιπλέον, το Feast παρέχει τα Projects (βλ. ενότητα 3.1.2) που προσφέρουν ένα απομονωμένο περιβάλλον χαρακτηριστικών και οντοτήτων.

Σε ένα συνεργατικό περιβάλλον όπως το Kubeflow πρέπει να υπάρχουν πολλαπλά αντικείμενα και να μοιράζονται μεταξύ των χρηστών. Τα ονόματα των Projects πρέπει

να είναι μοναδικά, για αυτό το λόγο θα χρησιμοποιήσουμε 1:1 αντιστοιχία μεταξύ ενός Feast project και ενός Kubeflow namespace.

Για την κοινή χρήση αντικειμένων ή ολόκληρων namespaces, οι διαχειριστές δίνουν δικαιώματα σε συγκεκριμένους χρήστες ή ομάδες ώστε να εκτελούν ενέργειες σε πόρους σε namespaces χρησιμοποιώντας K8s Roles (ClusterRoles) και RoleBindings (ClusterRoleBindings).



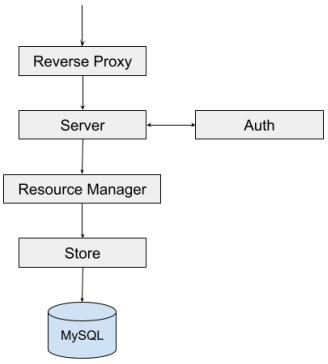
Σχήμα 3.7: *Επίπεδα Απομόνωσης*

3.3.3 Υπηρεσία Μητρώου

Η σχεδίαση και η αρχιτεκτονική του Feast Registry Service είναι σε μεγάλο βαθμό εμπνευσμένη από τον Kubeflow Pipelines (KFP) backend apiserver [4]. Μια πολύ παρόμοια δομή φαίνεται στο Σχήμα 3.8.

Η υπηρεσία του μητρώου διαθέτει 5 κύρια στοιχεία που έχουν διαφορετικούς ρόλους και εκτελούν διαφορετικές ενέργειες κατά τη διάρκεια κάθε αιτήματος.

Reverse Proxy Λαμβάνει αιτήματα HTTP από τους πελάτες, τα μετατρέπει σε αιτήσεις gRPC και τα αποστέλλει στον Server. Η αντίστροφη διαδικασία συμβαίνει μετά την παραγωγή μιας απάντησης gRPC από τον Server.



Σχήμα 3.8: *Αρχιτεκτονική Υπηρεσίας Μητρώου*

Server Πρόκειται για έναν τυπικό διακομιστή gRPC [20]. Αποτελεί το σημείο εισόδου των αιτημάτων gRPC και υλοποιεί συναρτήσεις που τα χειρίζονται. Εκτελεί ελέγχους επικύρωσης, χρησιμοποιεί το Auth στοιχείο για να εκτελέσει τον έλεγχο ταυτότητας και την εξουσιοδότηση και μεταβιβάζει το αίτημα στον υποκείμενο Resource Manager.

Auth Λαμβάνει ένα token και ένα σύνολο δικαιωμάτων από το στοιχείο Server. Στη συνέχεια μεταβιβάζει το token σε έναν πελάτη TokenReview, ο οποίος εκτελεί έλεγχο ταυτότητας. Ο πελάτης TokenReview είναι ένας πελάτης K8s [22] που αλληλεπιδρά με ένα εγγενές API του K8s. Η εξουσιοδότηση εκτελείται από έναν πελάτη SubjectAccessReview χρησιμοποιώντας τόσο το token όσο και το σύνολο των δικαιωμάτων που έλαβε ο Server. Πρόκειται για έναν άλλο πελάτη K8s που αλληλεπιδρά με λειτουργίες που παρέχονται από το API authorization.k8s.io [21].

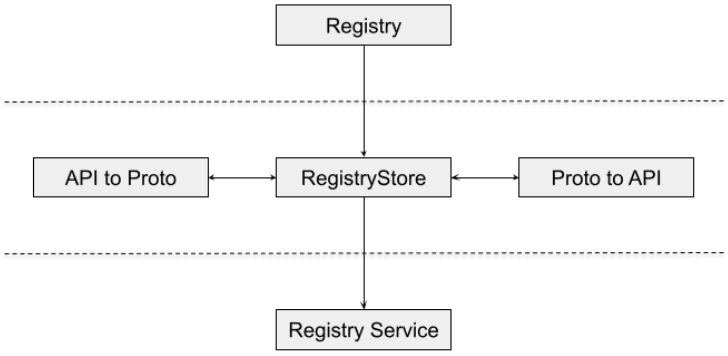
Resource Manager Λαμβάνει ένα αίτημα από τον Server και το μετατρέπει σε μια εσωτερική αναπαράσταση (μοντέλο) που μπορεί να αντιμετωπιστεί από στοιχεία χαμηλότερου επιπέδου. Διαχειρίζεται τους πόρους σε υψηλό επίπεδο και είναι υπεύθυνο για την ενεργοποίηση των κατάλληλων Stores χαμηλού επιπέδου προκειμένου να εκτελεστούν οι σωστές αλλαγές στην υποκείμενη βάση δεδομένων.

Store Λαμβάνει ένα μοντέλο και εκτελεί μια ενέργεια σε αυτό, η οποία διατηρείται στη βάση δεδομένων. Έτσι, αλληλεπιδρά με το πραγματικό σύστημα αποθήκευσης εκτελώντας ερωτήματα ή συναλλαγές. Σε αυτή την περίπτωση το σύστημα αποθήκευσης αναφέρεται σε μια βάση δεδομένων MySQL. Παρόλα αυτά, μπορεί να αλληλεπι-

δράσει και με άλλους τύπους SQL βάσεων αφού επιλέγουμε να χρησιμοποιήσουμε έναν ORM (Object-Relational Mapping) μηχανισμό.

3.3.4 Πελάτης Μητρώου

Η σχεδίαση του προγράμματος-πελάτη μητρώου είναι αρκετά απλοϊκός. Ο στόχος είναι να είναι ένα στοιχείο που αξιοποιεί τον μηχανισμό πρόσθετων που προσφέρει το Feast. Στον πυρήνα του εκτελεί προγραμματιστικά αιτήματα HTTP προς την υπηρεσία μητρώου Feast (FRS).



Σχήμα 3.9: *Αρχιτεκτονική Πελάτη Μητρώου*

RegistryStore Είναι υπεύθυνο για τη δημιουργία και τη διαχείριση του πελάτη που επικοινωνεί με το FRS. Συμπληρώνει τα αιτήματα με τα κατάλληλα πεδία και το απαιτούμενο token. Εάν το token δεν είναι πλέον έγκυρο, το ανανεώνει.

Proto to API Όπως αναφέραμε προηγουμένως, το αντικείμενο τύπου Registry μεταβιβάζει μια λίστα λειτουργιών και το περιεχόμενό τους στο αντικείμενο τύπου RegistryStore. Το περιεχόμενο όμως είναι σε μορφή protobuf. Πριν από την αποστολή ενός αιτήματος HTTP πρέπει να μετατρέψουμε το περιεχομενο του σε ένα κατάλληλο αντικείμενο API.

API to Proto Η υπηρεσία μητρώου απαντά με ένα ή περισσότερα αντικείμενα API. Ωστόσο, το αντικείμενο τύπου RegistryStore πρέπει να περάσει μια πλήρη αναπαράσταση μητρώου στο αντικείμενο τύπου Registry. Έτσι, πρέπει να μετατρέψουμε τα λαμβανόμενα αντικείμενα κατάλληλα σε αναπαράσταση protobuf.

4 Υλοποίηση

Αυτό το κεφάλαιο επικεντρώνεται στην παρουσίαση όλων των τεχνικών λεπτομερειών που απαιτούνται για την υλοποίηση της νέας αρχιτεκτονικής. Αρχικά, περιγράφει όλες τις μικρές αλλαγές στην υπάρχουσα βάση κώδικα του Feast. Στη συνέχεια, παρουσιάζει ολόκληρη την προσπάθεια σχετικά με τη νέα υπηρεσία μητρώου. Τέλος, παρουσιάζει τον νέο πελάτη μητρώου. Μαζί με όλες τις ειδικές λεπτομέρειες της υλοποίησης, επεξηγεί διεξοδικά τα βήματα ανάπτυξης ή εγκατάστασης όπου χρειάζεται.

4.1 Επισκόπηση

Έχοντας ήδη περιγράψει όλα τα νέα στοιχεία που πρέπει να δημιουργήσουμε είναι σημαντικό να τονίσουμε ότι ο στόχος της υλοποίησης είναι να εισαχθεί νέα λειτουργικότητα χωρίς να επηρεαστεί η υπάρχουσα λειτουργικότητα. Έτσι, πραγματοποιούμε ελάχιστες αλλαγές στην υπάρχουσα βάση κώδικα. Επιπλέον, τα εντελώς νέα στοιχεία προσπαθούν να παρέχουν λύσεις που δημιουργούν αφαιρέσεις και μπορούμε εύκολα να επεκτείνουμε και να χρησιμοποιήσουμε εκτός περιβάλλοντος Kubeflow.

Δεδομένου ότι ο πελάτης του Feast (SDK) είναι γραμμένος σε Python και αυτός είναι ο πιο δημοφιλής και εύκολος τρόπος για να ξεκινήσει κάποιος με το Feast, θα χρησιμοποιήσουμε Python για τις βασικές αλλαγές του Feast και τον νέο πελάτης του μητρώου. Ωστόσο, θα υλοποιήσουμε την υπηρεσία μητρώου χρησιμοποιώντας την Go, καθώς πολλές εγγενείς εφαρμογές νέφους έχουν ήδη αναπτυχθεί σε Go και το τελικό αποτέλεσμα είναι εξαιρετικά αποδοτικό. Επιπλέον, επαναχρησιμοποιούμε μέρος του κώδικα του backend του KFP και μειώνουμε τον συνολικό χρόνο που θα χρειαζόταν για την

υλοποίηση των ίδιων μηχανισμών σε άλλη γλώσσα προγραμματισμού.

4.2 Πυρήνας Feast

Η επέκταση της κλάσης Registry είναι απαραίτητη ώστε να χειρίζεται σωστά τη λίστα εκκρεμών ενεργειών. Όπως εξηγήσαμε προηγουμένως, αυτή είναι μια λίστα FIFO. Για την υλοποίησή της θα χρησιμοποιήσουμε το module queue [19].

```
class Registry:
  cached_registry_proto: Optional[RegistryProto] = None
  cached_registry_proto_created: Optional[datetime] = None
  cached_registry_proto_ttl: timedelta
  pending_ops: queue.Queue
```

Κατά την εκτέλεση πράξεων ”write” γεμίζουμε αυτή την ουρά με dicts που περιέχουν τον τύπο της λειτουργίας και το απαραίτητο περιεχόμενο. Ακολουθούν μερικά παραδείγματα:

```
# CreateEntity and UpdateEntity example
def apply_entity(
  self, entity: Entity, project: str, commit: bool = True
):
  update = False
  entity.is_valid()
  ...

  for idx, existing_entity_proto in enumerate(
    self.cached_registry_proto.entities
  ):
    if (
      existing_entity_proto.spec.name == entity_proto.spec.name
      and existing_entity_proto.spec.project == project
    ):
      del self.cached_registry_proto.entities[idx]
      self.pending_ops.put({
        ”op”: ”UpdateEntity”, ”proto”: entity_proto
      })
```

```
        update = True
        break

    self.cached_registry_proto.entities.append(entity_proto)
    if not update:
      self.pending_ops.put({
        "op": "CreateEntity", "proto": entity_proto
      })
    if commit:
      self.commit()
```

```
# DeleteEntity example
def delete_entity(
  self, name: str, project: str, commit: bool = True
):
  self._prepare_registry_for_changes()
  assert self.cached_registry_proto

  for idx, existing_entity_proto in enumerate(
    self.cached_registry_proto.entities
  ):
    if (
      existing_entity_proto.spec.name == name
      and existing_entity_proto.spec.project == project
    ):
      del self.cached_registry_proto.entities[idx]
      self.pending_ops.put({"op": "DeleteEntity", "name": name})
      if commit:
        self.commit()
      return

  raise EntityNotFoundException(name, project)
```

Κάποια στιγμή πρέπει να σώσουμε τις αλλαγές στο μητρώο. Για το λόγο αυτό, μεταβιβάζουμε τη λίστα εκκρεμών ενεργειών στο αντικείμενο RegistryStore.

```
def commit(self):
  if self.cached_registry_proto:
    self._registry_store.update_registry_proto(
```

```
        self.cached_registry_proto, self.pending_ops
    )
```

Ωστόσο, η αφηρημένη κλάση RegistryStore περιέχει μια αφηρημένη μέθοδο που ενημερώνει το registry proto με ένα μόνο όρισμα. Η νέα μέθοδος έχει ένα επιπλέον όρισμα με προεπιλεγμένη τιμή (μια κενή ουρά) σε περίπτωση που η λίστα των εκκρεμών λειτουργιών δεν είναι χρήσιμη για το υλοποιημένο RegistryStore.

```
# Old definition
def update_registry_proto(self, registry_proto: RegistryProto):

# New definition
def update_registry_proto
  self,
  registry_proto: RegistryProto,
  pending_ops: queue.Queue = queue.Queue()
):
```

4.3 Υπηρεσία Μητρώου

Για τη δημιουργία της υπηρεσίας μητρώου χρησιμοποιούμε σχεδιαστικά πρότυπα παρόμοια με το backend του KFP apiserver. Η διαδικασία που περιγράφουμε στο υπόλοιπο αυτής της ενότητας αναφέρεται σε αντικείμενα τύπου entity και λειτουργεί ως ένα εύκολο παράδειγμα να ακολουθήσει κάποιος. Ωστόσο, η ίδια διαδικασία ισχύει για όλα τα άλλα αντικείμενα πρώτης κατηγορίας του μητρώου.

Ορισμοί Protobuf

Το πρώτο βήμα της διαδικασίας είναι ο ορισμός των τελικών σημείων δημιουργώντας τους απαιτούμενους protobuf ορισμούς. Το ακόλουθο παράδειγμα παρουσιάζει ένα EntityService με πέντε διαφορετικά τελικά σημεία: Create, Get, Update, Delete και List.

```
service EntityService {
  rpc CreateEntity (CreateEntityRequest) returns (Entity) {
    option (google.api.http) = {
      post: "/CreateEntity",
      body: "entity"
    };
  }

  rpc GetEntity (GetEntityRequest) returns (Entity) {
    option (google.api.http) = {
      get: "/GetEntity"
    };
  }

  rpc UpdateEntity (UpdateEntityRequest) returns (Entity) {
    option (google.api.http) = {
      post: "/UpdateEntity",
      body: "entity"
    };
  }

  rpc DeleteEntity (DeleteEntityRequest) returns (google.protobuf.Empty) {
    option (google.api.http) = {
      delete: "/DeleteEntity"
    };
  }

  rpc ListEntities (ListEntitiesRequest) returns (ListEntitiesResponse) {
    option (google.api.http) = {
      get: "/ListEntities"
    };
  }
}
```

Δεδομένου ότι το πλαίσιο gRPC βασίζεται σε πελάτες που μπορούν να καλέσουν απευθείας μια μέθοδο σε μια εφαρμογή διακομιστή σε διαφορετικό μηχάνημα, οι παραπάνω ορισμοί λειτουργούν ως η πηγή της αλήθειας. Χρησιμοποιούμε έναν μεταγλωττιστή για τη μετάφραση αυτών των ορισμών σε πραγματικό κώδικα που χρησιμοποιούμε ως stub πελάτη που παρέχει τις ίδιες μεθόδους με τον διακομιστή. Στην πλευρά του διακομιστή, είναι ευθύνη του προγραμματιστή να υλοποιήσει αυτές τις μεθόδους και να δημιουργήσει έναν διακομιστή που χειρίζεται τα αιτήματα του πελάτη.

Server

Έτσι, το επόμενο βήμα είναι η υλοποίηση αυτών των μεθόδων. Για αρχή, συγκρίνουμε πώς φαίνονται αυτές οι μέθοδοι στην πλευρά του πελάτη και του διακομιστή.

Για την πλευρά του πελάτη, εδώ είναι ένα παράδειγμα του παραγόμενου κώδικα που δημιουργεί ένας μεταγλωττιστής:

```
func (c *entityServiceClient) CreateEntity(
    ctx context.Context, in *CreateEntityRequest, opts ...grpc.CallOption
  ) (*Entity, error) {
  out := new(Entity)
  err := c.cc.Invoke(
    ctx, "/api.EntityService/CreateEntity", in, out, opts...
  )
  if err != nil {
    return nil, err
  }
  return out, nil
}
```

Για την πλευρά του διακομιστή, έτσι υλοποιούμε την αντίστοιχη μέθοδο:

```
func (s *EntityServer) CreateEntity(
    ctx context.Context, request *api.CreateEntityRequest
  ) (*api.Entity, error) {
  resourceAttributes := &authorizationv1.ResourceAttributes{
    Namespace: request.Namespace,
    Verb:      common.RbacResourceVerbCreate,
  }
  err := s.haveAccess(ctx, resourceAttributes)
  if err != nil {
    return nil, util.Wrap(err, "Failed to authorize the request.")
  }
  entity, err := s.resourceManager.CreateEntity(
    request.Entity, request.Namespace
  )
  if err != nil {
    return nil, util.Wrap(err, "Create entity failed.")
  }
  return ToApiEntity(entity), nil
}
```

Πιστοποίηση και Εξουσιοδότηση

Καθώς ένα αίτημα φτάνει στον server, αυτός πραγματοποιεί έλεγχος πρόσβασης πριν από την πραγματική εκτέλεση της ζητούμενης λειτουργίας. Αρχικά, εξάγει το token από το αίτημα και ξεκινάει ένα TokenReview. Το TokenReview είναι στην πραγματικότητα ένα αίτημα προς το TokenReview API του Κυβερνήτη, το οποίο επικυρώνει το token και εξασφαλίζει ότι προορίζεται για το καθορισμένο audience, στην προκειμένη περίπτωση *features.kubeflow.org*.

```
func (tra *TokenReviewAuthenticator) doTokenReview(
    ctx context.Context, userIdentity string
  ) (*authv1.UserInfo, error) {
  review, err := tra.client.Create(ctx,
    &authv1.TokenReview{
      Spec: authv1.TokenReviewSpec{
        Token:     userIdentity,
        Audiences: tra.audiences,
      },
    },
    v1.CreateOptions{},
  )
  ...
  return &review.Status.User, nil
}
```

Μετά από αυτό, ο server εξουσιοδοτεί το αίτημα. Αυτή τη φορά χρησιμοποιεί την ταυτότητα του χρήστη μαζί με ένα σύνολο χαρακτηριστικών πόρων για την εκτέλεση ενός SubjectAccessReview. Ένα SubjectAccessReview είναι και πάλι ένα αίτημα προς το αντίστοιχο API το οποίο εξασφαλίζει ότι ένας χρήστης μπορεί να εκτελέσει μια ενέργεια (verb) σε έναν συγκεκριμένο πόρο ενός χώρου ονομάτων.

```
func (r *ResourceManager) IsRequestAuthorized(
    ctx context.Context, userIdentity string, userGroups []string,
    resourceAttributes *authorizationv1.ResourceAttributes
  ) error {
  result, err := r.subjectAccessReviewClient.Create(ctx,
    &authorizationv1.SubjectAccessReview{
      Spec: authorizationv1.SubjectAccessReviewSpec{
        ResourceAttributes: resourceAttributes,
```

```
        User:                 userIdentity,
        Groups:               userGroups,
      },
    },
    v1.CreateOptions{},
  )
  ...
  return nil
}
```

Για παράδειγμα, ένα αίτημα */CreateEntity* που χρησιμοποιεί τα ακόλουθα χαρακτηριστικών πόρων:

- Namespace: kubeflow-user
- Verb: create
- Group: features.kubeflow.org
- Version: v1beta1
- Resource: entities

εξουσιοδοτείται από τον server μόνο εάν ο χρήστης που υποβάλλει το αίτημα έχει δικαίωμα εκτέλεσης *create* σε *entities* του *features.kubeflow.org* και έκδοσης *v1beta1* στο *kubeflow-namespace*.

Resource Manager

Το επόμενο βήμα της διαδικασίας είναι να δώσει ο server εντολή στον resource manager να εκτελέσει την αιτούμενη λειτουργία.

```
func (r *ResourceManager) CreateEntity(
    apiEntity *api.Entity, namespace string
  ) (*model.Entity, error) {
  project, err := r.projectStore.GetProject(apiEntity.Project, namespace)
  if err != nil {
    return nil, util.Wrap(err, "Failed to find project")
  }
```

```
  entity, err := r.ToModelEntity(apiEntity, ”””, project.Id)
  if err != nil {
    return nil, util.Wrap(err, ”Failed to convert entity model”)
  }

  return r.entityStore.CreateEntity(entity)
}
```

Δεδομένου ότι ο resource manager μπορεί να διαχειριστεί όλα τα αντικείμενα του μητρώου, ξεκινά με την εύρεση του project στο οποίο ανήκει αυτή το νέο entity και χρησιμοποιεί το projectId του μαζί με το entity του αιτήματος για να δημιουργήσει ένα μοντέλο (μια εσωτερική αναπαράσταση ενός αντικειμένου). Στη συνέχεια μεταβιβάζει αυτό το μοντέλο στο κατάλληλο υποκείμενο store όπου το projectId θα χρησιμοποιηθεί από το Store ως εξωτερικό κλειδί.

Store

Το τελικό βήμα της διαδικασίας περιλαμβάνει το Store το οποίο λαμβάνει ένα μοντέλο τύπου entity και δημιουργεί το αντίστοιχο ερώτημα SQL. Υποθέτοντας ότι το ερώτημα SQL είναι έγκυρο, το εκτελεί και ελέγχει για πιθανά σφάλματα κατά την εκτέλεση.

```
func (s *EntityStore) CreateEntity(
    e *model.Entity
  ) (*model.Entity, error) {
  newEntity := *e

  id, err := s.uuid.NewRandom()
  ...
  newEntity.Id = id.String()

  sql, args, err := sq.
    Insert(”entities”).
    SetMap(
      sq.Eq{
        ”id”:                    newEntity.Id,
        ”project_id”:            newEntity.ProjectId,
        ”name”:                  newEntity.Name,
        ”value_type”:            newEntity.ValueType,
        ”description”:           newEntity.Description,
```

```
        "join_key":                newEntity.JoinKey,
        "tags":                    newEntity.Tags,
        "owner":                   newEntity.Owner,
        "created_timestamp":       newEntity.CreatedTimestamp,
        "last_updated_timestamp": newEntity.LastUpdatedTimestamp,
      }).
    ToSql()
  ...

  _, err = s.db.Exec(sql, args...)
  if err != nil {
    ...
  }

  return &newEntity, nil
}
```

Η διαδικασία ολοκληρώνεται με την αποστολή μιας απάντησης πίσω στον πελάτη που έκανε το αίτημα. Πριν από αυτό, ο Server μετατρέπει το μοντέλο ξανά σε ένα κατάλληλο αντικείμενο API.

Αυτή η διαδικασία από άκρο σε άκρο είναι σχεδόν πανομοιότυπη για τα υπόλοιπα αντικείμενα του μητρώου. Ωστόσο, υπάρχουν αντικείμενα όπως τα Feature Views που πρέπει να χειριστούν π.χ. τα εξαρτώμενα Features τους και τις αντίστοιχες καταχωρήσεις τους στη βάση δεδομένων. Από πλευράς υλοποίησης γίνεται λίγο πιο πολύπλοκο, καθώς πρέπει να χρησιμοποιήσουμε τυπικές συναλλαγές βάσης δεδομένων για να εξασφαλίσουμε την συνέπεια. Ακολουθεί ένα παράδειγμα:

```
func (s *FeatureViewStore) CreateFeatureView(
    fv *model.FeatureView
  ) (*model.FeatureView, error) {
  newFeatureView := *fv
  ...

  tx, err := s.db.Begin()
  if err != nil {
    ...
  }

  _, err = tx.Exec(sql, args...)
  if err != nil {
```

```
    tx.Rollback()
    ...
  }

  for _, feature := range newFeatureView.Features {
    _, err = s.featureStore.CreateFeature(
      tx, feature, newFeatureView.Id
    )
    if err != nil {
      tx.Rollback()
      ...
    }
  }

  err = tx.Commit()
  if err != nil {
    tx.Rollback()
    ...
  }

  return &newFeatureView, nil
}
```

Σε περίπτωση που προκύψει κάποιο σφάλμα κατά τη διάρκεια της συναλλαγής, εκτελούμε επαναφορά (rollback) για να εξασφαλίσουμε μια συνεπής κατάσταση της βάσης δεδομένων. Εάν όλα τα ερωτήματα εκτελεστούν με επιτυχία, διατηρούμε το αποτέλεσμα εκτελώντας το commit της συναλλαγής.

4.4 Πελάτης Μητρώου

Η δημιουργία του πελάτη μητρώου αφορά την υλοποίηση των τριών αφηρημένων μεθόδων της κλάσης RegistryStore. Ας δούμε πώς λειτουργούν όλα βήμα προς βήμα.

Η αρχικοποίηση του RegistryStore περιλαμβάνει τη δημιουργία ενός ApiClient και της ρύθμισής του. Το πακέτο frs_api Python που χρησιμοποιούμε, έχει ήδη δημιουργηθεί αυτόματα από τους ορισμούς protobuf της υπηρεσίας μητρώου (βλ. ενότητα 4.3). Ως αποτέλεσμα, αφαιρεί όλη τη δύσκολη δουλειά της αποστολής ενός αιτήματος και του χειρισμού της απάντησης. Το αντικείμενο Configuration που μεταβιβάζουμε στον ApiClient παρέχει μια συνάρτηση που είναι υπεύθυνη για την ανανέωση του token που

πρέπει να συμπεριλάβουμε στην επικεφαλίδα εξουσιοδότησης των αιτημάτων. Μπορούμε να βρούμε το token είτε σε ένα αρχείο ή σε μια μεταβλητή περιβάλλοντος.

```
class KubeflowRegistryStore(RegistryStore):
  def __init__(self, registry_config: RegistryConfig, repo_path: Path):
    ...
    config = frs_api.configuration.Configuration()
    config.host = registry_config.path

    config.api_key['authorization'] = 'token'
    config.api_key_prefix['authorization'] = 'Bearer'
    config.refresh_api_key_hook =
      ServiceAccountTokenVolumeCredentials().refresh_api_key_hook

    api_client = frs_api.api_client.ApiClient(configuration=config)

    self._entity_api = frs_api.api.entity_service_api.
      EntityServiceApi(api_client)
    ...
```

Σκοπός της μεθόδου get_registry_proto() είναι η ανάκτηση ολόκληρης της αναπαράστασης του μητρώου. Αρχικά, λαμβάνει τα μεταδεδομένα που αφορούν το project, τα οποία και επαληθεύουν την ύπαρξη του. Στη συνέχεια εκτελεί ένα αίτημα List για όλα τα αντικείμενα πρώτης κατηγορίας του μητρώου για να συγκεντρωθούν οι αποθηκευμένοι ορισμοί των αντικειμένων του project.

```
  @log_exceptions_and_usage(registry="kubeflow")
  def get_registry_proto(self):
    try:
      if not self.readMode:
        proj = self._project_api.project_service_get_project(
          project=self.project,
          namespace=self.namespace
        )
    except frs_api.exceptions.ApiException as e:
      raise FileNotFoundError(
        f'Project named "{self.project}" not found.'
      )
    self._get_entities(registry_proto)
    ...
    return registry_proto
```

```
def _get_entities(self, registry_proto: RegistryProto):
  res = self._entity_api.entity_service_list_entities(
    project=self.project,
    namespace=self.namespace
  )

  if not res.entities:
    return

  for e in res.entities:
    entity_proto = entity_to_proto(e)
    registry_proto.entities.append(entity_proto)
```

Σκοπός της μεθόδου update_registry_proto() είναι να αλλάξει την κατάσταση της αποθηκευμένης αναπαράστασης του μητρώου και αυτό γίνεται με λεπτομερή τρόπο εκτελώντας μόνο τις απαραίτητες αλλαγές. Με βάση τη λίστα των λειτουργιών που λαμβάνεται από το αντικείμενο τύπου Registry, εκτελούνται οι αντίστοιχες κλήσεις API μαζί με το απαιτούμενο περιεχόμενο. Μετά την ολοκλήρωση μιας επιτυχούς ενημέρωσης, η λίστα αδειάζει.

```
@log_exceptions_and_usage(registry="kubeflow")
def update_registry_proto(
    self,
    registry_proto: RegistryProto,
    pending_ops: queue.Queue = queue.Queue()
  ):
  while not pending_ops.empty():
    op = pending_ops.get()
    ...
    elif op['op'] == 'CreateEntity':
      entity = entity_to_api(op['proto'])
      self._entity_api.entity_service_create_entity(
        body=entity,
        namespace=self.namespace
      )
    elif op['op'] == 'UpdateEntity':
      entity = entity_to_api(op['proto'])
      self._entity_api.entity_service_update_entity(
        body=entity,
        namespace=self.namespace
      )
```

```
      elif op['op'] == 'DeleteEntity':
        self._entity_api.entity_service_delete_entity(
          name=op['name'],
          project=self.project,
          namespace=self.namespace
        )
      ...
```

Ο στόχος της μεθόδου teardown() είναι η διαγραφή όλων όσων σχετίζονται με το συγκεκριμένο project. Μια απλή κλήση API είναι αρκετή, καθώς ολόκληρη η διαδικασία εκτελείται στην πλευρά της υπηρεσίας μητρώου.

```
  @log_exceptions_and_usage(registry="kubeflow")
  def teardown(self):
    self._project_api.project_service_delete_project(
      project=self.project,
      namespace=self.namespace
    )
```

Μετατροπή Αντικειμένων

Όπως έχουμε εξηγήσει στο κεφάλαιο Σχεδίαση, το αντικείμενο τύπου Registry περνάει μια αναπαράσταση μητρώου σε protobuf μορφή στο αντικείμενο RegistryStore. Έτσι, χρειαζόμαστε έναν τρόπο μετασχηματισμού των αντικειμένων protobuf σε κατάλληλα αντικείμενα API και αντίστροφα. Υλοποιούμε συναρτήσεις όπως η entity_to_api() ή η entity_to_proto() για αυτό το λόγο:

```
def entity_to_api(proto: EntityProto) -> ApiEntity:
  return ApiEntity(
    name=proto.spec.name,
    project=proto.spec.project,
    value_type=proto.spec.value_type,
    description=proto.spec.description,
    join_key=proto.spec.join_key,
    tags=dict(proto.spec.tags),
    owner=proto.spec.owner,
    created_timestamp=proto.meta.created_timestamp.ToDatetime().
      replace(tzinfo=timezone.utc),
    last_updated_timestamp=proto.meta.last_updated_timestamp.
```

```
        ToDatetime().replace(tzinfo=timezone.utc)
    )

def entity_to_proto(api: ApiEntity) -> EntityProto:
    meta = EntityMetaProto()
    meta.created_timestamp.FromDatetime(getattr(
      api,
      "created_timestamp",
      datetime(1970, 1, 1, tzinfo=timezone.utc))
    )
    meta.last_updated_timestamp.FromDatetime(getattr(
      api,
      "last_updated_timestamp",
      datetime(1970, 1, 1, tzinfo=timezone.utc))
    )

    spec = EntitySpecProto(
        name=api.name,
        project=api.project,
        value_type=ValueType[api.value_type].value,
        description=api.description,
        join_key=api.join_key,
        tags=api.tags,
        owner=api.owner,
      )

    return EntityProto(spec=spec, meta=meta)
```

4.5 Ρύθμιση Συστάδας

4.5.1 Υπηρεσία Μητρώου

Μετά την ανάπτυξη του FRS, το επόμενο βήμα είναι η πραγματική εγκατάσταση της υπηρεσίας στο Kubeflow. Το Kubeflow αναπτύσσει όλες τις υπηρεσίες του κάτω από το namespace kubeflow. Θα ακολουθήσουμε την ίδια προσέγγιση και για το FRS.

Η εγκατάσταση της βάσης δεδομένων MySQL που χρησιμοποιεί το FRS είναι τόσο απλή όσο η δημιουργία ενός StatefulSet και ενός Service στον Κυβερνήτη.

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql-feast-sts
spec:
  replicas: 1
  serviceName: mysql-feast-svc
  selector:
    matchLabels:
      app: mysql-feast
  template:
    metadata:
      name: mysql-feast
      labels:
        app: mysql-feast
    spec:
      containers:
      - name: mysql
        image: bitnami/mysql:8.0
        volumeMounts:
          - name: data
            mountPath: /bitnami/mysql/data
        env:
          - name: MYSQL_USER
            value: "feastadmin"
          - name: MYSQL_PASSWORD
            value: "feastadmin"
          - name: MYSQL_DATABASE
            value: "feast"
        ports:
          - name: mysql
            containerPort: 3306
            protocol: TCP
        ...
---
apiVersion: v1
kind: Service
metadata:
  name: mysql-feast-svc
spec:
  type: ClusterIP
  ports:
  - name: mysql
    port: 3306
```

```
    targetPort: mysql
  selector:
    app: mysql-feast
```

Ένα StatefulSet διασφαλίζει ότι δεν θα χαθούν δεδομένα εάν ένα Pod σταματήσει να λειτουργεί απροσδόκητα και ένα νέο πάρει τη θέση του.

Για να εγκαταστήσουμε τον διακομιστή API, πρέπει πρώτα να δημιουργήσουμε ένα ConfigMap [8]. Αυτό περιέχει ένα αρχείο json με όλες τις ρυθμίσεις, π.χ. τις ρυθμίσεις της βάσης δεδομένων, που χρειάζεται ο διακομιστής για να λειτουργήσει. Στη συνέχεια, δημιουργούμε ένα τυπικό Deployment και ένα Service.

```
apiVersion: v1
data:
  # apiserver assumes the config is named config.json
  config.json: |
    {
      "DBConfig": {
        "DriverName": "mysql",
        "DBName": "feast",
        "GroupConcatMaxLen": "4194304",
        "ConMaxLifeTime": "120s",
        "Host": "mysql-feast-svc",
        "User": "feastadmin",
        "Password": "feastadmin"
      },
      "InitConnectionTimeout": "10s",
      "MULTIUSER": true
    }
kind: ConfigMap
metadata:
  name: frs-config
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: feast-registry-d
  labels:
    app: feast-registry
spec:
  selector:
    matchLabels:
```

```yaml
      app: feast-registry
  template:
    metadata:
      labels:
        app: feast-registry
    spec:
      containers:
      - name: feast-registry-api-server
        env:
        - name: POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
        image: apiserver
        imagePullPolicy: Never
        command:
          - /tmp/apiserver
          - --config=/etc/frs-config
        ports:
        - containerPort: 8888
        - containerPort: 8887
        volumeMounts:
        - name: config-volume
          mountPath: /etc/frs-config
      volumes:
        - name: config-volume
          configMap:
            name: frs-config
---
apiVersion: v1
kind: Service
metadata:
  name: feast-registry-svc
spec:
  type: ClusterIP
  ports:
  - name: http
    port: 8888
    protocol: TCP
    targetPort: 8888
  - name: grpc
    port: 8887
    protocol: TCP
    targetPort: 8887
  selector:
```

```
    app: feast-registry
```

Ο διακομιστής εκθέτει δύο θύρες (8888, 8887), μία για τον διακομιστή HTTP και μία άλλη για τον διακομιστή gRPC.

4.5.2 Υποστήριξη Πιστοποίησης

Για την υποστήριξη της πιστοποίησης χρησιμοποιούμε μια προσέγγιση που βασίζεται σε token. Για το λόγο αυτό, απαιτείται ένα token που να μπορούμε να βρούμε σε ένα αρχείο ή μια μεταβλητή περιβάλλοντος μέσα σε ένα Pod. Αυτό το token είναι ένα ServiceAccountToken [12] το οποίο συνδέεται με έναν συγκεκριμένο ServiceAccount.

Ένα ServiceAccount στον Κυβερνήτη είναι ένας τρόπος παροχής ταυτότητας για τα Pods. Πολλαπλά service accounts συνδέονται με ένα namespace και έχουν διαφορετικά σύνολα δικαιωμάτων σε αυτό. Μπορούμε να προσαρτήσουμε το ServiceAccountToken σε Pods σε καθορισμένη τοποθεσία χρησιμοποιώντας ένα ProjectedVolume [11] και να το χρησιμοποιήσουμε απλά διαβάζοντας το αρχείο που το περιέχει. Η ρύθμιση ενός νέου PodDefault διασφαλίζει την προσάρτηση του volume σε κάθε νέο Pod.

```
apiVersion: kubeflow.org/v1alpha1
kind: PodDefault
metadata:
  name: access-features
spec:
  desc: Allow access to Kubeflow Features
  selector:
    matchLabels:
      access-ml-pipeline: "true"
  volumeMounts:
  - mountPath: /var/run/secrets/kubeflow/features
    name: volume-features-token
    readOnly: true
  volumes:
  - name: volume-features-token
    projected:
      sources:
      - serviceAccountToken:
        path: token
        expirationSeconds: 7200
```

```
        audience: features.kubeflow.org
  env:
  - name: KF_FEATURES_SA_TOKEN_PATH
    value: /var/run/secrets/kubeflow/features/token
```

Το αρχείο με το token βρίσκεται στο /var/run/secrets/kubeflow/features/token και ανανεώνεται αυτόματα από το σύστημα κάθε 7200 δευτερόλεπτα. Αυτό το token ισχύει μόνο για το features.kubeflow.org audience.

4.5.3 Υποστήριξη Εξουσιοδότησης

Για να λειτουργήσει η εξουσιοδότηση χρησιμοποιούμε τον εγγενή μηχανισμό RBAC του Κυβερνήτη. Αυτός ο μηχανισμός βασίζεται σε τυπικά αντικείμενα API του K8s (ClusterRoles/Roles, RoleBindings, ServiceAccounts) και τον μηχανισμό SubjectAccessReview (SAR). Η γενική ιδέα είναι ο καθορισμός πόρων, ενεργειών που μπορούν να εκτελεστούν σε αυτούς τους πόρους, καθώς και η σύνδεση αυτών των ενεργειών με service accounts ή χρήστες. Για αυτό, χρησιμοποιούμε τον SAR για να ελέγξουμε αν ένας χρήστης επιτρέπεται να εκτελέσει μια ενέργεια σε έναν πόρο ενός χώρου ονομάτων.

Τα service accounts συνδέονται με ClusterRoles ή Roles [10] μέσω RoleBindings που καθορίζουν δικαιώματα ενός ServiceAccount σε ένα συγκεκριμένο χώρο ονομάτων. Ένας ClusterRole ή Role περιέχει ένα σύνολο κανόνων που συνδυάζουν apiGroups, resources, resourceNames και verbs. Ως αποτέλεσμα, ένας κανόνας καθορίζει ποιες ενέργειες (verbs) μπορεί να εκτελέσει ένα ServiceAccount σε συγκεκριμένους πόρους.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: kubeflow-feature-edit
rules:
- apiGroups:
  - features.kubeflow.org
  resources:
  - data_sources
  - entities
  - feature_services
  - feature_views
  - infra_objects
```

```
  - on_demand_feature_views
  - projects
  - request_feature_views
  - saved_datasets
  verbs:
  - create
  - update
  - delete
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: kubeflow-feature-view
rules:
- apiGroups:
  - features.kubeflow.org
  resources:
  - data_sources
  - entities
  - feature_services
  - feature_views
  - infra_objects
  - on_demand_feature_views
  - projects
  - request_feature_views
  - saved_datasets
  verbs:
  - get
  - list
```

Για παράδειγμα, τα service accounts που είναι συνδεδεμένα με το ρόλο kubeflow-feature-edit μπορούν να δημιουργούν, να ενημερώνουν και να διαγράφουν όλα τα αντικείμενα του μητρώου, ενώ αυτοί που είναι συνδεδεμένοι με το ρόλο kubeflow-featureview μπορούν μόνο να λαμβάνουν ή να παραθέτουν αντικείμενα του μητρώου.

5 Επίλογος

Αυτό το τελευταίο κεφάλαιο, παρουσιάζει μια σύντομη περίληψη της δουλειάς μας και υπογραμμίζει τις κύριες συνεισφορές μας. Αξιολογεί ορισμένα κύρια χαρακτηριστικά της διαδικασίας σχεδιασμού και υλοποίησης και, τέλος, αναφέρει πιθανές επεκτάσεις και περαιτέρω βελτιώσεις που θα μπορούσαν να αναπτυχθούν στο μέλλον.

5.1 Συμπερασματικά Σχόλια

Ο αρχικός στόχος αυτής της διπλωματικής εργασίας ήταν η ενσωμάτωση ενός συστήματος που διαχειρίζεται χαρακτηριστικά, ενός αποθετηρίου χαρακτηριστικών, σε μια πλατφόρμα ΜΜ, στο Kubeflow. Καταφέραμε να πετύχουμε αυτόν τον στόχο μετατρέποντας ένα αποθετήριο χαρακτηριστικών ανοιχτού κώδικα, το Feast, με λίγη υποστήριξη για περιβάλλοντα νέφους σε ένα νεφο-εγγενές σύστημα πελάτη-εξυπηρετητή. Για να το πετύχουμε αυτό:

1. Δημιουργήσαμε έναν διακομιστή REST API που διαχειρίζεται τους ορισμούς των χαρακτηριστικών και τα μεταδεδομένα τους.

2. Αναπτύξαμε έναν πελάτη που αλληλεπιδρά με τον διακομιστή API σε περιβάλλον Κυβερνήτη.

3. Βελτιώσαμε το Feast προσθέτοντας υποστήριξη πιστοποίησης και εξουσιοδότησης.

Τώρα, οι επιστήμονες δεδομένων είναι σε θέση να μοιράζονται με ασφάλεια χαρακτηριστικά και να συνεργάζονται. Μπορούν επίσης να χρησιμοποιήσουν όλα τα εργαλεία που προσφέρει ένα αποθετήριο χαρακτηριστικών και να βελτιώνουν ριζικά τις ροές εργασίας ΜΜ μειώνοντας τον χρόνο που δαπανούν για την ανάπτυξη νέων χαρακτηριστικών και διασφαλίζοντας ότι τα μοντέλα τους λαμβάνουν συνεπή χαρακτηριστικά υψηλής ποιότητας.

Κατά τη διάρκεια των φάσεων σχεδιασμού και υλοποίησης έπρεπε να αλληλεπιδρούμε δυναμικά, να βελτιώνουμε και να τεκμηριώνουμε συνεχώς τη δουλειά μας. Ξοδέψαμε πολύ χρόνο παραδίδοντας εσωτερικά παρουσιάσεις, τα οποία λειτούργησαν ως απόδειξη της έννοιας (proof of concept) και είχαν ως αποτέλεσμα πολύτιμες παρατηρήσεις. Ωστόσο, το αποκορύφωμα της όλης διαδικασίας ήταν η παράδοση μιας ζωντανής παρουσίασης στην κοινότητα του Feast κατά τη διάρκεια της κλήσης που πραγματοποιεί η κοινότητα κάθε δεύτερη εβδομάδα. Αυτό μας έδειξε πραγματικά ότι κινούμαστε προς τη σωστή κατεύθυνση, απέδειξε το ενδιαφέρον της κοινότητας για τη δουλειά μας και μας βοήθησε να συγκεντρώσουμε χρήσιμα σχόλια.

5.2 Μελλοντικό Έργο

Η ριζική αλλαγή του τρόπου λειτουργίας του Feast αποτελεί το πρώτο βήμα για τη δημιουργία ενός πραγματικά ισχυρού αποθετηρίου χαρακτηριστικών ανοικτού κώδικα που θα μπορεί να υποστηρίζει διαφορετικά είδη περιπτώσεων χρήσης και περιβαλλόντων. Αυτό σημαίνει ότι υπάρχουν μεγάλα περιθώρια βελτίωσης και πολλές δυνατότητες επέκτασης του Feast. Οι παρακάτω ιδέες είναι μόνο η κορυφή του παγόβουνου όσον αφορά την πλήρη αξιοποίηση της νέας αρχιτεκτονικής:

- Επέκταση του προγράμματος-πελάτη του Feast SDK ώστε να χρησιμοποιεί όλες τις δυνατότητες του νέου διακομιστή API. Δυνατότητα λήψης συγκεκριμένων ορισμών χαρακτηριστικών χωρίς να χρειάζεται να ληφθεί πρώτα ολόκληρο το project.

- Χρήση μηχανισμών του Κυβερνήτη για την ενίσχυση των δυνατοτήτων του Feast. Για παράδειγμα, χρήση του RBAC του Κυβερνήτη για επιβολή ελέγχου πρόσβασης στις υποκείμενες πηγές δεδομένων (τόσο στο Offline όσο και στο Online

Store) ή βεβαίωση ότι διαγράφονται οι πόροι ενός namespace σε περίπτωση καταστροφής του.

- Χρήση του Rok ή άλλων συστημάτων διαχείρισης συνόλων δεδομένων για την κοινή χρήση ολόκληρων συνόλων δεδομένων με αποδοτικό τρόπο μέσα σε ένα περιβάλλον Kubeflow.

- Δυνατότητα εκτέλεσης αυτοματοποιημένων αγωγών χαρακτηριστικών με τη χρήση του KFP επεκτείνοντας τον διακομιστή API σε έναν πραγματικό χρονοδρομολογητή αγωγών.

- Κατασκευή ενός συστήματος παρακολούθησης της ποιότητας δεδομένων γύρω από το Feast ή ενσωμάτωση ενός υπάρχοντος. Αυτή είναι στην πραγματικότητα μια κατεύθυνση προς την οποία κατευθύνεται επί του παρόντος η κοινότητα του Feast.

Ελπίζουμε πραγματικά ότι αυτές οι ιδέες είναι αρκετές για να κάνουν άλλους να σκεφτούν περαιτέρω βελτιώσεις και επεκτάσεις. Για εμάς, ο στόχος αυτή τη στιγμή είναι να προωθήσουμε και να συνεισφέρουμε πραγματικά τη δουλειά μας στο upstream πρότζεκτ του Feast και να του επιτρέψουμε να ανθίσει παράλληλα με την κοινότητά του.

1 Introduction

The first chapter aims on setting the context of this thesis. At the beginning, it presents the motives that led to exploring feature stores. Then, it showcases the problems that this entire effort targets to solve as well as existing solutions. Finally, it exposes a brief summary of the proposed solution.

1.1 Motivation

Nowadays, more and more people are involved in the development process or at least use machine learning (ML) applications on a daily basis as it proves to be valuable for solving complex real-world problems. However, ML workloads and processes seem to vary at different stages of an application (development, use, maintenance). Thus, providing the right systems and tools for data scientists and engineers so as to make the workloads more efficient and the processes simpler and sustainable is crucial.

Machine Learning Lifecycle and Workflows

Diving deeper into scientists' and engineers' daily processes we have observed that they are always working on designing and implementing workflows that support the entire ML lifecycle. A high level overview of a typical ML workflow that tries to match the lifecycle of an ML application is the following:

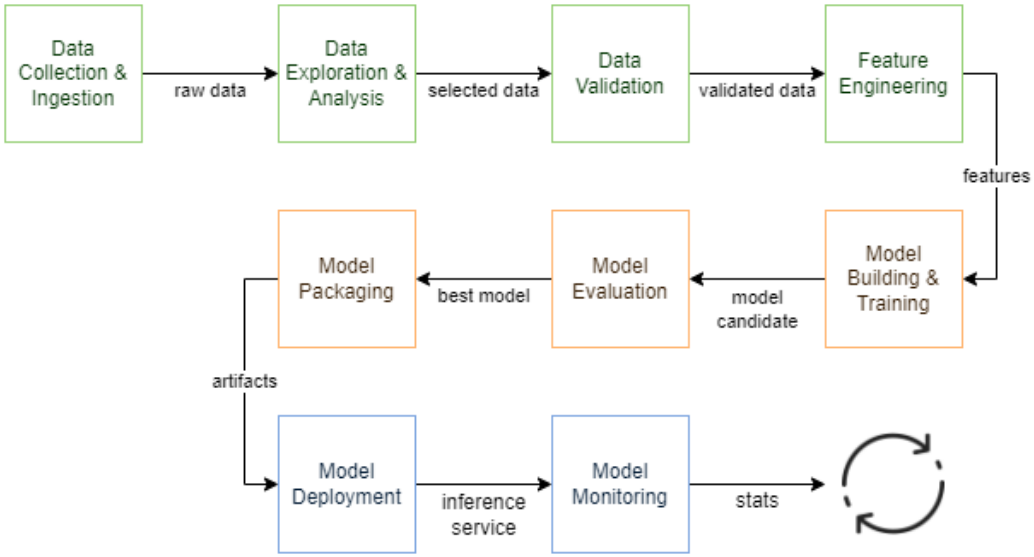


Figure 1.1: *Typical ML Workflow*

The above workflow contains multiple steps and could be separated into three main sub-workflows:

- Data workflow
- Model workflow
- Serving workflow

All three of them require different kinds of domain expertise and special skills, making the development and deployment of the entire workflow difficult and time consuming. Assuming that one or multiple groups of people are working on each of these three parts, iterating on them independently is necessary and acts as a great motive in the effort of optimizing ML workflows.

MLOps and Best Practices

The last couple of years a new approach called MLOps has emerged trying to address challenges in ML workflows and systems by applying DevOps [27] principles on them. This approach is indeed an ML engineering culture and practice that aims at unifying ML system development (Dev) and ML system operation (Ops) [16].Automation and

monitoring at all steps of a typical ML workflow are considered among best practices for avoiding common pitfalls usually described as the technical debt of ML [14]. Following best practices to simplify complex processes and further improve ML workflows is another motive of this thesis.

1.2 Problem Statement

Data engineering (or feature engineering) workflows are usually an important piece of an ML workflow as features and data in general is of extreme value for a model's success. Creating new features requires a lot of effort and time, therefore reusing and sharing them is crucial. Moreover, it is essential to use produced features in a consistent way both for training a model (see Model workflow) and during inference (see Serving workflow) to avoid training-serving skew.

Challenges of Machine Learning Systems

Taking a closer look at why feature creation and subsequently serving is hard, the first thing we noticed is the variety of data sources these features originate from. Batch or streaming data sources as well as a mix of both is very common in cutting-edge ML applications. It is obvious that these sources have different characteristics regarding data quality or data freshness. For example, a batch source like a data warehouse contains historical time-series data creating a complete history of a feature, whereas a stream source keeps only real-time data. As a result, the operations that can be performed on this data are totally different as well.

Continuing, these features may need to be calculated at different time intervals or even at the time of a prediction, thus having a data pipeline run at different points in time requires immense engineering effort and it can greatly increase prediction time. As a consequence, the decoupling of feature creation and feature consumption is mandatory in production ML systems. New questions that occur are:

- How often should features be created?
- What is the cost-efficiency tradeoff?

Next up, is the very common training-serving skew problem [24]. This is a result of failing to provide the same features both for training and inference by building two distinct implementations for feature creation. One implementation that mostly happens in specified time intervals and another one that executes in real-time during each prediction request.

Since feature engineering is time-consuming, making sure that created features have a standard structure and can be easily accessed by multiple data scientists and engineers is another challenge. Lack of ways to share features and collaborate results in excess duplication inside teams and organizations.

Last but not least, ensuring the quality of data that is being served to models is another challenging task. Questions such as "Is my model receiving the right data and still operating correctly?" or "Has there been a drift in data over time?" are very common in production ML systems.

Summarizing, the 4 main challenges this thesis' effort aims to solve or at least decrease their negative effects are:

- Creation of feature pipelines
- Consistent data access
- Duplicate features
- Ensuring data quality

1.3 Existing Solutions

Until now a plethora of projects and organizations have tried to develop their own solutions to tackle the aforementioned challenges. The ideas and products they have created are a great starting point to understand the value they bring in the ML world.

Michelangelo ML Platform

Michelangelo [23] was essentially the first complete ML platform that Uber created around 2017 as a need for tackling a number of challenges related with building and

deploying numerous machine learning models at scale. During the first years of Uber data scientists were using various tools to create models and engineers were building bespoke one-off systems to use these models in production. Michelangelo standardized workflows and tools across teams by providing an end-to-end system which helped to easily build and operate ML systems at scale.

Building better systems for managing and sharing feature pipelines was an important part of the entire system. This is how the term Feature Store was born. Uber engineers created a complete system for building data pipelines that generate features and label data sets both for training and predictions. They also added a data management layer that allowed teams to share, discover and use more than 10.000 features which automatically got calculated and updated daily. Finally, they created feature monitoring tools that observe the importance of a feature to a model along with partial dependence plots and distribution histograms.

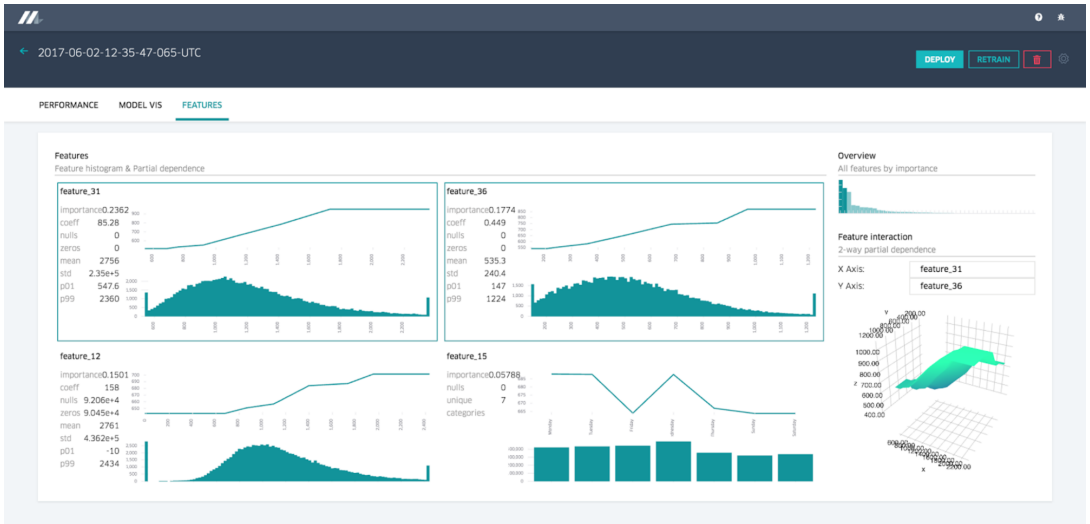


Figure 1.2: *Feature Report*

Tecton Feature Platform

Tecton was founded by a group of people that created Uber Michelangelo. Rather than focusing on the entire ML process, it provides an enterprise-ready feature platform that is built to orchestrate the complete lifecycle of features, from transformation to online serving [28]. Its open-source solution is Feast which will also be the focal point of this thesis.

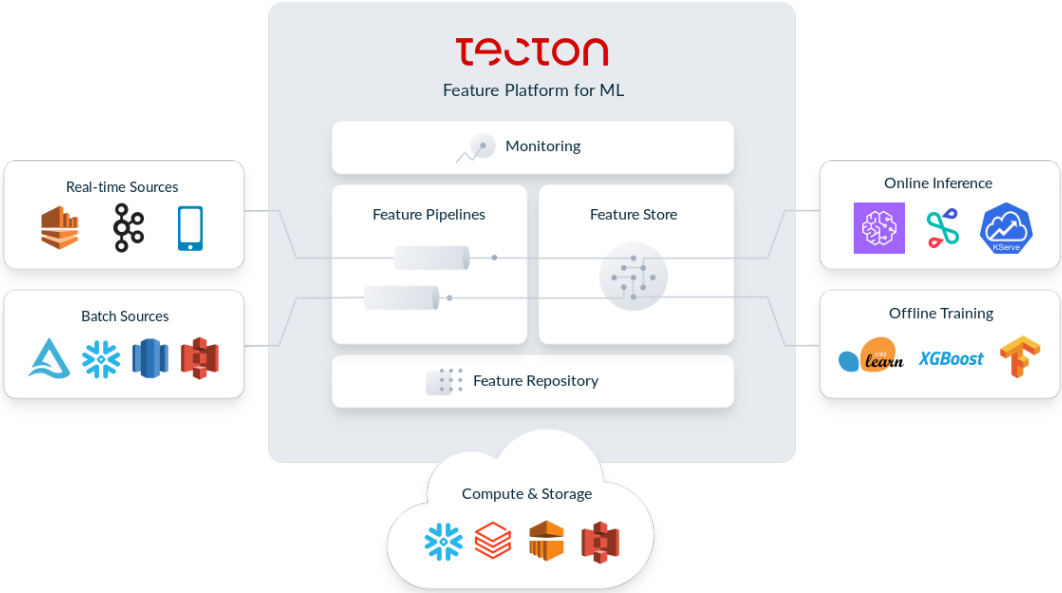


Figure 1.3: *Tecton Architecture*

Tecton allows users to define and manage features using code in a git repository. It helps them automatically compute and orchestrate feature transformations as well as consistently store online and offline feature data. Finally, it makes sure to effectively serve feature data both for model training and inference.

Vertex AI Feature Store

Vertex AI developed by Google is another ML platform that aims on building, deploying and scaling ML models faster, with pre-trained and custom tooling. Part of this cutting-edge platform is a feature store. Vertex AI's Feature Store provides a centralized repository for organizing, storing, and serving ML features [15].

Among other benefits, it provides a managed online feature serving layer, data drift detection mechanisms and automatic data retention, keeping feature values fresh. Moreover, it enforces quotas and limits to manage resources efficiently both on the serving and data layer.

Custom Solutions

At the moment, more and more organizations and big tech companies have created their own custom solutions to solve part or all of the challenges mentioned earlier. Here is a

list with the most common solutions:

Amazon SageMaker Feature Store	AirBnB Zipline
Databricks Feature Store	FBLearner Flow
Hopswork Feature Store	Spotify Jukebox Feature Store
Linkedin Feathr	Apple Overton
Rasgo	Salesforce Feature Store
Scribble Data	Netflix Metaflow

1.4 Proposed Solution

Integration of Feast into Kubeflow

The purpose of this thesis is to integrate a Feature Store inside an existing ML platform, in this case Kubeflow. Kubeflow is already a well developed and maintained open-source platform, but it's still missing a way to manage data and features. Integrating it with a Feature Store will attempt to solve all of the challenges we mentioned earlier and try to actually detach the data engineering workflow from model training and model serving workflows (see Figure 1.4). In addition, it will help users make ML workflows simpler, more portable and more scalable.

Using Feast as the chosen feature store has many benefits as it is open-source and doesn't rely on vendor specific technologies or infrastructure. It is just the right candidate as it also has many contributors and an active community.

To successfully integrate Feast into Kubeflow, we will extend its Registry component. The goal is to apply minimal changes to the existing code base and leverage the plugin mechanism Feast offers as much as possible. For this reason we will create a pluggable client that handles all interactions with the new Registry backend. The backend will be a REST API server that is responsible for storing feature definitions and related metadata in an SQL database as well as enforcing access control mechanisms. Worth mentioning is the fact that this new backend is part of the current Feast roadmap and something

highly demanded by active users. We will eventually contribute it to the open-source project in order to make Feast even more flexible and extensible.

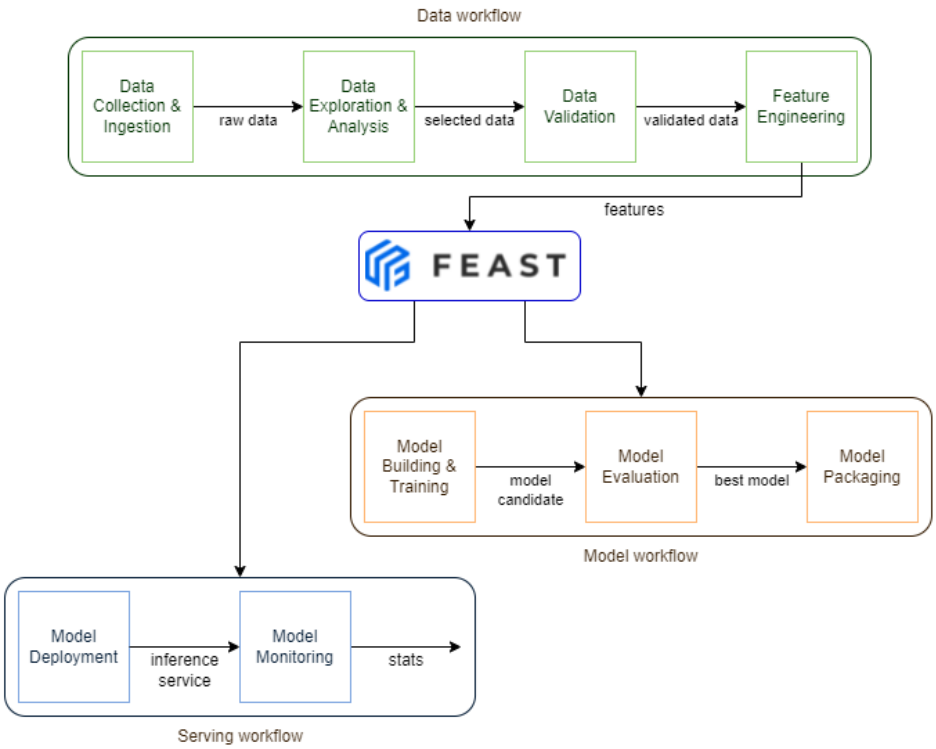


Figure 1.4: *Split ML Workflow*

1.5 Outline

The rest of this thesis has the following structure:

- **Chapter 2**: High level overview of useful technologies, terms and concepts
- **Chapter 3**: Complete presentation of the current Feast architecture and new components' design
- **Chapter 4**: Implementation specific details regarding new components
- **Chapter 5**: Concluding remarks and possible future work

2 Background

This chapter presents a high level overview of all prerequisite knowledge. Having a basic understanding of the following concepts and technologies is essential for understanding the design and implementation parts of this thesis. For a start, it explains containers and Kubernetes as everything runs on top of them. Next up is Kubeflow, a complete ML platform which leverages Kubernetes to support the entire ML lifecycle. Lastly, this chapter describes a feature store's core components and functionalities.

2.1 Containers

Nowadays, containers are the most popular form of operating system virtualization. They are executable units of software in which application code is packaged, along with its libraries and dependencies, so that it can be run across multiple environments such as the cloud or a personal computer [18]. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. These processes are also lightweight, efficient and portable allowing them to take full advantage of the underlying machine resources [1].

To better understand containers, let's see how they differ from traditional virtual machines (VMs). VMs are an abstraction of the physical hardware (CPU, memory, storage etc.) whereas containers are an abstraction at the app layer. In the first case, we leverage a hypervisor to virtualize physical hardware, thus each VM contains a guest OS, a virtual copy of the hardware that the OS requires to run, along with an application and its dependencies. In the second case, containers virtualize the operating system so each

container contains only the application and its dependencies. The absence of a guest OS results in saving resources and improving performance.

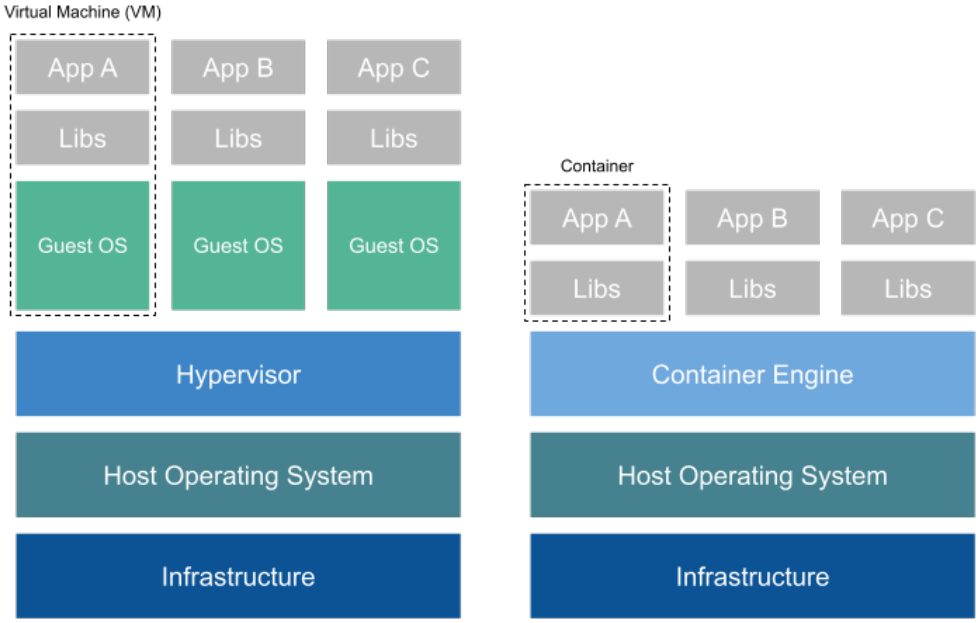


Figure 2.1: *Virtual Machines vs Containers*

2.2 Kubernetes

Having already explained what containers are, it is clear that containerizing applications brings a lot of benefits to the table. However, standardizing app deployment and using resources efficiently across multiple environments are just the tip of the iceberg. In large scale environments where multiple applications run on hundreds of nodes there is a vital need to orchestrate and manage these applications. This is where Kubernetes (K8s) comes to play.

2.2.1 Overview

Kubernetes is an orchestrator of containerized applications at its core. It is a system which deploys and manages applications dynamically without the need of user intervention. To do so, it provides a declarative way of specifying a desired state and is re-

sponsible for matching it with the actual running state. The best part of it is that it is a fast-moving open-source project that can run on any cloud or on-premise data center by abstracting the underlying infrastructure [9].

2.2.2 Architecture

After deploying Kubernetes you get a cluster. This cluster consists of a set of nodes and a control plane. The control plane could be thought of as the brain of the cluster whereas the nodes as the muscle. More specifically, the control plane exposes an API, has a scheduler for assigning work to nodes and records its state in a persistent store. Nodes are responsible for running the applications, which means listening to the control plane for new tasks, executing them and reporting back to it [26].

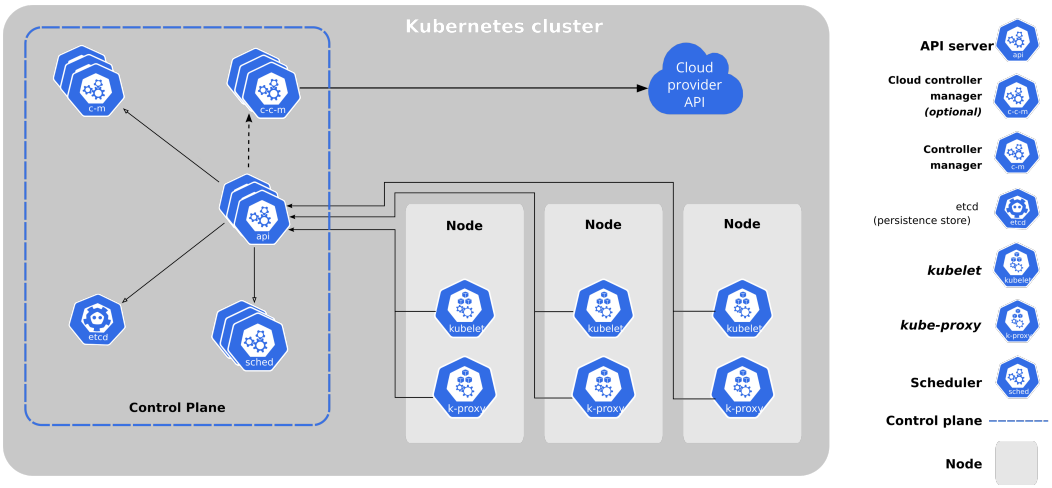


Figure 2.2: *Kubernetes Cluster Components*

Control Plane

The control plane has five main components:

API Server It exposes the Kubernetes API that all components use to communicate between each other. Users send manifests (YAML configuration files) containing the desired state of an application to the API server which initially authenticates and authorizes the requests, then validates and persists these files to the cluster store and finally deploys them to the cluster.

Cluster Store It is the only stateful part of the control plane which is responsible for storing the configuration and state of the cluster. It is based on etcd, a distributed key-value store, and acts as the single source of truth. The cluster store is vital, as the absence or failure of it means essentially no cluster. Consistency is the number one priority and in case something goes wrong the system will halt and wait so as to maintain it.

Scheduler It watches the API server for newly created Pods or tasks and assigns them to healthy worker nodes. For a node to run a task multiple checks regarding free resources, available network ports etc. are performed and nodes are ranked accordingly. The highest ranked node will execute the requested task. In case a suitable node cannot be found by the system, a task will be marked as pending and wait until it can be scheduled.

Controller Manager It is responsible for executing control loops that monitor the cluster and respond to events. It is actually a set of controllers running under a single process. These controllers execute watch loops constantly watching the API server and ensuring that the current state of the cluster matches the desired one.

Cloud Controller Manager In K8s clusters running in cloud platforms such as AWS, Azure, GCP etc., it handles controllers specific to the cloud provider. For example, it creates, updates or deletes the cloud provider's load balancers for applications using internet facing load balancers.

Nodes

Nodes consist of three major components:

Kubelet It is an agent that runs on each node and is responsible for registering it with the cluster. It actively watches the API server for new tasks, executes them and reports back to the API server.

Container Runtime Each node has a container runtime which is used by kubelet to start/stop containers, pull images etc. K8s provides a plugin mechanism called Con-

tainer Runtime Interface (CRI) that exposes an interface for all kinds of container runtimes to plug into. At the time of writing containerd is the most popular container runtime used in K8s.

Kube-Proxy It is a network proxy that handles local cluster networking by maintaining network rules, getting unique IP addresses, redirecting traffic etc.

2.2.3 Core Concepts

Let's continue by taking a look at the most common K8s API objects that will help to better understand how applications are deployed and run in a K8s environment.

Pods

A Pod is the atomic unit of scheduling in Kubernetes, the same way a VM is in virtualization or a container is in Docker. A single Pod acts as a sandbox consisting of one or more containers which share the same environment (IPC namespace, shared memory, volumes, network etc.). Deploying a Pod is an atomic operation requiring all of its containers to become ready before it is considered ready. All of the containers are deployed in the same node for obvious reasons. In general, Pods are treated by users as expendables, they are destined to be created, live, die and then seamlessly be replaced by new ones.

Deployments

Standalone Pods are not enough in a large scale environment as they don't offer self-healing or scalability capabilities. A Deployment is responsible for doing so by replacing failing Pods with new ones or by increasing/decreasing Pods based on a specified metric. In addition, it provides rolling updates and rollbacks allowing the sytem to go back and forth between application versions with zero-downtime. It does so by gradually increasing/decreasing Pods of newer/older versions while also keeping history of all the used configurations.

Stateful Sets

Deployments add great value in stateless applications. However, when Pods fail, they are replaced by totally new ones (new name, hostname, volume bindings) resulting in loss of the Pod's state. Stateful applications require that the Pod's state is kept by the system even if a Pod fails. To do so, a StatefulSet ensures predictable and persistent Pod names and DNS hostnames as well as a unique set of volumes which stays with the Pod for its entire lifecycle.

Services

A Service is an abstraction that allows users to expose applications running on a set of Pods as a network service. To do so, it sits in front of the Pods and provides a reliable name, IP address and port. Clients can connect to the underlying Pods using the Service, which load-balances the requests to the target Pods.

Namespaces

Namespaces is a mechanism that provides isolation of resources in a single K8s cluster. It is a way to partition the cluster in multiple virtual clusters and is useful when applying resource quotas or access control policies. It is important to mention that not every single object can be namespaced. For example, API objects such as Nodes or PersistentVolumes cannot be namespaced.

2.3 Kubeflow

After going through K8s fundamentals, it is obvious that it offers various capabilities and extreme flexibility. Thus, it has emerged as the de facto platform for deploying and managing workloads in a cloud native way. Meanwhile, machine learning workloads have evolved to be one of the most common workloads executed in cloud infrastructure. The need to create an ecosystem that supports these kinds of workloads in a cloud environment led to Kubeflow.

2.3.1 Overview

Kubeflow is a platform designed for development and deployment of complete machine learning (ML) systems. It is built for data scientists who want to build and experiment with ML pipelines as well as ML engineers and Ops teams wanting to deploy ML systems in multiple environments (development, testing, production) [7]. Furthermore, it is open-source, cloud-native, as it runs on top of K8s, and supports the entire ML lifecycle. To do so, it provides a set of tools crucial for ML while also integrating them in a sharing and collaboration environment which manages permissions and handles access control.

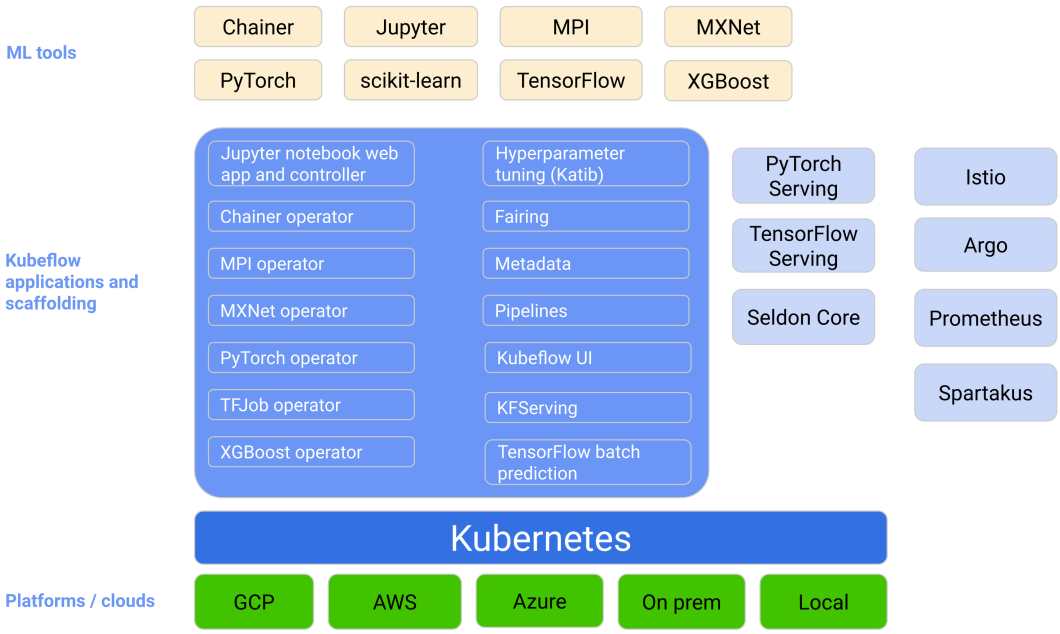


Figure 2.3: *Kubeflow Overview*

2.3.2 Core Concepts

As mentioned before, Kubeflow is comprised of cloud-native tools that support the entire ML lifecycle. Let's have a closer look on the most common ones to better understand the value of the platform.

Notebook Servers

Notebook Servers or Kubeflow Notebooks (as they are also called) provide various web-based development environments such as JupyterLab, VS Code or R Studio which run

inside Pods. These servers support the entire process of experimentation, code development and execution. For this reason, they are extremely customizable allowing users to specify the image they want to use or the resources belonging to the Pod (CPU, memory, volumes etc.).

Pipelines

Kubeflow Pipelines (KFP) is a platform designed for building and deploying portable, scalable ML workflows based on Docker containers [5]. It is capable of orchestrating complex machine learning pipelines, performing and tracking experiments as well as scheduling recurring runs of these pipelines. It comes with a UI and a rich SDK client that interact with the platform making it easy for users to build end-to-end workflows using various tools.

Serving

After experimenting with Kubeflow Notebooks and Pipelines, users need a way to effectively serve and track their trained models. Kubeflow provides a way to serve models using TFServing, PyTorch, Triton etc. In general, an inference service is created from a saved model and becomes available for prediction requests. Adding to this, metadata and metrics are kept by the service so as to monitor model performance and quality of predictions. In this effort, KServe is the primary solution further encapsulating the complexity of autoscaling, networking and health checking as well as other cutting edge features [6].

Multi-Tenancy

In a Kubeflow environment where a lot of users collaborate and work on the same set of resources there is an inherent need for isolation and grouping of these users. The Kubeflow multi-user isolation permits users to only view and edit their own resources and configuration. At a high level administrators create users and configure permissions on different namespaces. Authentication is then applied using Istio and OIDC and authorization is provided by K8s RBAC. To successfully authenticate and authorize users, K8s provides them with a short-lived access token that they use to perform their actions.

2.4 Feature Store

Making Kubeflow a complete ML platform requires a component that is responsible for data. Machine Learning is essentially data and code. Till now Kubeflow has taken steps to provide tools that handle code and subsequent models, but the same hasn't happened for data. A feature store aims at bridging this gap and providing solutions to commonly faced data problems in applied ML.

2.4.1 Overview

A feature store [25] is an ML-specific data system that aims at productionizing and operationalizing data and features. For this reason, it offers various interesting functionalities:

1. It is responsible for managing and running data pipelines that transform raw data into valuable features. This includes automated feature computation, backfills and logging.
2. It stores and manages the created features and datasets and keeps track of feature versions, lineage and related metadata.
3. It provides a serving layer so as to serve feature data consistently both for training and inference purposes. It does so by using two database systems, one scale-out SQL database intended for batch training and another low latency store, which contains only the latest feature data, intended for lightning fast inference.
4. It offers a central registry which keeps feature data definitions and allows them to be shared and discovered among multiple users.
5. It monitors features and their quality in production systems making sure no drift occured. Data quality is of extreme value in ML systems as it is inseparably connected to a model's performance.

The end result is features that can now be defined in a standard way, registered in a central repository, stored and accessed for model training and inference by multiple users.

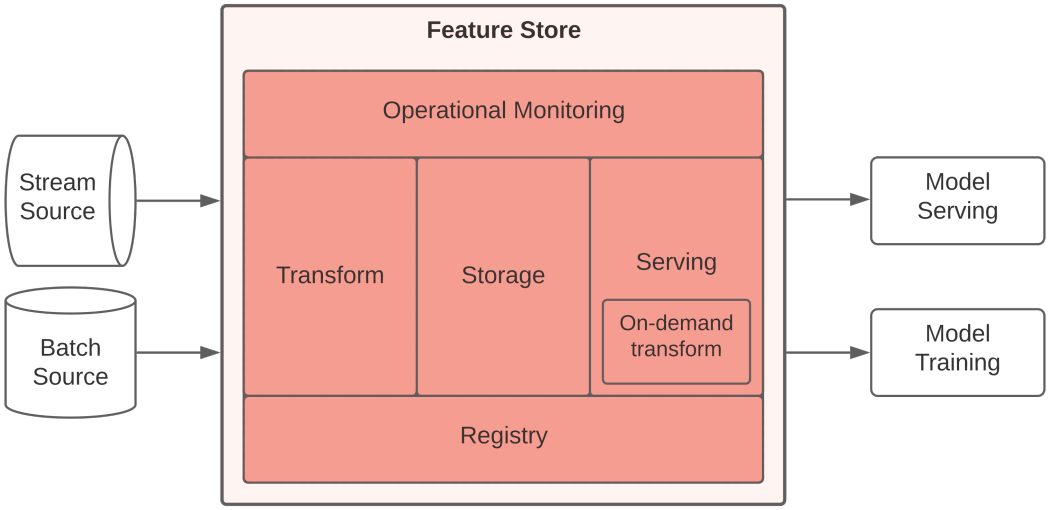


Figure 2.4: *Feature Store Overview*

2.4.2 Core Components

Moving forward it is important to understand in detail the purpose of a Feature Store's individual components and which functionalities they implement. Let's go through them one by one!

Transformation

The majority of ML projects perform some kind of feature engineering workflow. That means collecting raw data, validating and transforming them into valuable features. Raw data can be collected from various types of sources such as batch sources (data warehouse, database etc.), stream sources (Kafka, Kinesis etc.) or even request-time data (data that is collected at the time of prediction). Meanwhile, models need to access fresh feature values so as to improve predictions.

Transformations managed and executed by a feature store ensure that new data is processed and turned into fresh new feature values. This can happen on existing data resting on a data warehouse as part of a backfill job (batch transformation) or on streaming sources that need to aggregate values over a period of time (streaming transformation). They can also be applied to request-time data (on-demand transformation) e.g. transforming a user's GPS coordinates at the time of the request to an actual location. Having a consistent way of defining and reusing transformations is of great value as it ensures

that models get consistent feature data.

Storage

As its name suggests, a feature store includes or at least integrates with a storage system. Persisting features is crucial for feature retrieval through the serving layer later on. As mentioned before, a typical feature store contains an offline and an online storage layer.

The offline store is used to store large amounts of historical time-series data (data combined with a timestamp) and it is usually a well-known data warehouse or data lake such as S3 or BigQuery. This vast amount of data is later used for model training.

On the flip side, an online store contains the latest feature values for each entity which will be used for low-latency inference. It is usually a key-value store like DynamoDB or Redis.

Serving

A feature store is responsible for serving features to applications and models in a consistent way during training and inference. In order for a model to reach its full potential and avoid the very common training-serving skew problem [24], features that are used to train a model need to match the features provided for online serving.

The serving layer abstracts away the complex queries needed to extract feature values with point-in-time correctness and provides a uniform way to access these features from anywhere using an SDK. Based on the performance requirements different APIs are used to achieve the required results. Implementations of the same API in different languages or frameworks improve performance as well.

Registry

A centralized registry is an essential component of a feature store. It stores feature definitions and related metadata in a standardized way and acts as the single source of truth. Data scientists and teams use it to share and discover new features making it a collaborative tool. All the configuration is stored in the registry, thus all operations consult

the registry before performing any action. For example, serving APIs use the registry to understand which feature values to return or where to find them in a data warehouse.

Regarding stored metadata, the registry keeps various information related to feature definitions such as owner, description, domain specific information and lineage. This opens up an entire space of data monitoring and auditing as well as lineage tracking and ownership. In general, feature stores are designed to integrate with external systems or components that leverage the metadata and provide useful insight.

Monitoring

A feature store seems to be in the perfect position to monitor data. Since it is the intermediate layer between data and code, it can calculate metrics and monitor correctness and quality, alerting data scientists if unwanted behavior occurs. Data problems are in most cases the reason ML systems fail to perform to expectations, so monitoring data drift and training-serving skew as well as validating feature data can minimize these problems.

Furthermore, a feature store can keep track of operational metrics such as latency, throughout or processing time. All of these metrics can then be used by external monitoring or observability tools and provide extra visibility on the features and models using them. Again, it is very common for feature stores to easily integrate with external systems or components like the aforementioned.

3 Design

This chapter focuses on providing a full picture of the new design based on the proposed solution (see section 1.4). At first, it describes a high-level overview of the Feast components, core concepts and basic functionalities. Then, it presents technical details about the Feast registry and exposes existing problematic designs. Finally, it showcases the new proposed architecture in detail, focusing on changes and new components.

3.1 Feast Feature Store

Feast is an open-source feature store. It is an operational data system for managing and serving machine learning features to models in production. Feast is able to serve feature data to models from a low-latency online store (for real-time prediction) or from an offline store (for scale-out batch scoring or model training) [3]. Besides that, it is a system that makes feature data consistent and easily available across multiple environments (development, production) and teams while also providing a centralized registry that allows feature reuse.

Feast seems to be very similar to a common feature store, but at the moment of writing it is still missing important parts of it (transformations, validation, monitoring, discovery). Since it is an open-source project, its goal is not to create solutions from scratch, but develop abstractions and plugin mechanisms that will allow it to integrate with other systems seamlessly. This approach makes it unique and customizable among other solutions as it's not vendor or framework specific nor requires dedicated infrastructure.

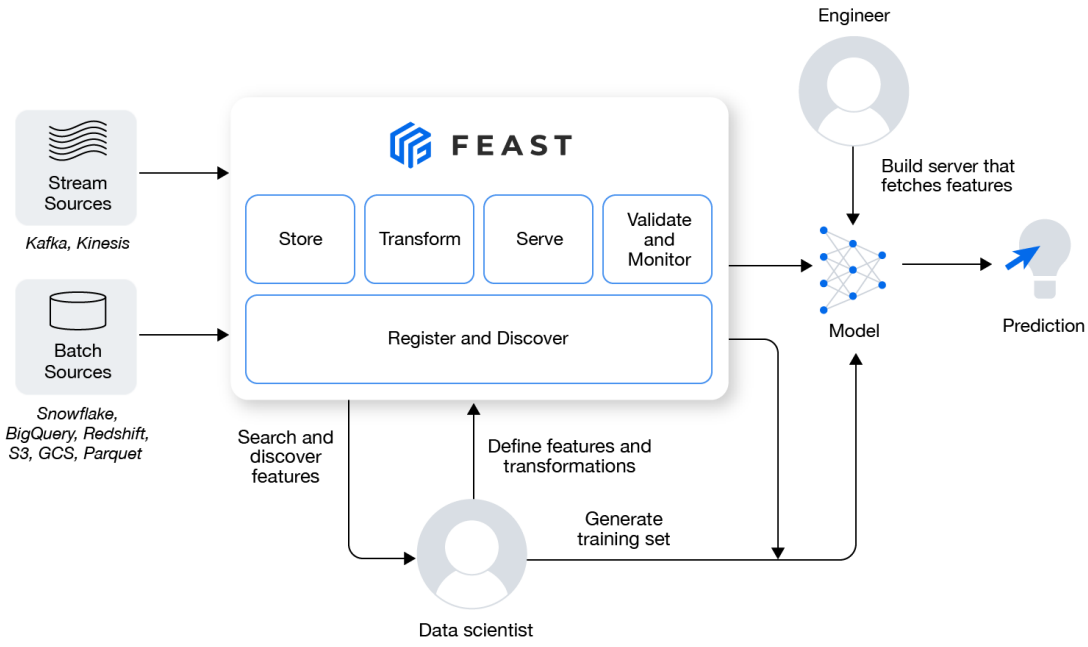


Figure 3.1: *Feast Overview*

3.1.1 Architecture

The Feast architecture is pretty simplistic and contains 3 main components:

1. Offline Store
2. Online Store
3. Registry

Other than that, there are components or layers that enhance the use of the feature store such as a feature repository, a feature server or a provider. Moving forward, all parts of the architecture will be presented in detail.

Offline Store

An Offline Store refers to the actual data source which contains historic time-series feature values. This data source is usually a data warehouse/data lake (Google's BigQuery, Amazon's Redshift etc.) or simply a database (e.g. PostgreSQL). It supports stream data sources (e.g. Kafka or Kinesis streams) as well. It is the component whose purpose is

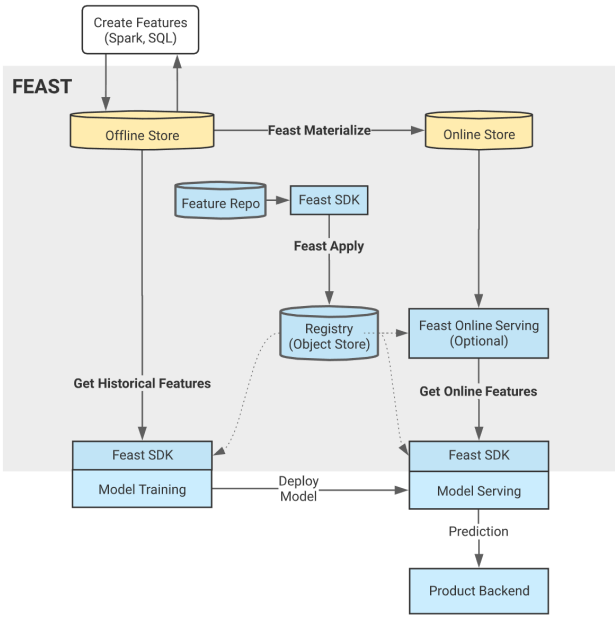


Figure 3.2: *Feast Architecture*

to offer feature data for offline training and to provide the latest feature values to the Online Store during the materialization process.

In general, Feast is not a storage system, but instead uses the Offline Store as the interface for querying existing feature values. Minimal configuration is needed to define the preferred Offline Store. Here is an example:

```
offline_store:
  type: bigquery
  dataset: feast_bq_dataset
  project_id: feast-oss
  location: EU
```

Online Store

An Online Store is usually a fast and scalable database or key-value store (Google's Datastore, Amazon's DynamoDB, Redis, etc.) managed by Feast. It is the component responsible for offering only the latest feature data and is used for online inference.

The latest feature values are loaded into the Online Store from data sources in Feature Views during the materialization process. The schema of the Online Store mirrors the

schema of the data sources used to populate it. We will describe a more detailed description of how this process works later on (see section 3.1.3). Again, defining an Online Store requires minimal effort:

```
online_store:
  type: redis
  redis_type: redis_cluster
  connection_string: "redis1:6379,redis2:6379,password=my_password"
```

Registry

A Registry is a fully managed Feast component, meaning that Feast is responsible for creating, updating and tearing it down. It refers to an object storage (e.g. GCS bucket, S3 bucket) that keeps the registry proto file. It is the component that contains all of the feature definitions and their related metadata. It is like a feature data catalog and acts as the single source of truth about features. Each Feast deployment has a single Registry and whenever feature data is retrieved either from the Offline and the Online Store or another operation is performed, the Registry acts as a consultant providing useful metadata.

The registry proto file that is essentially stored, is a Protobuf [17] representation of Feast metadata. In a user friendly format the registry looks like the following:

```
{
  "entities": [
    {
      "spec": {
        "name": "driver",
        "valueType": "STRING",
        "joinKey": "driver_id"
      },
      "meta": {}
    }
  ],
  "featureViews": [
    {
      "spec": {
        "name": "driver_activity",
        "entities": [ "driver" ],
        "features": [
          {
```

```
            "name": "trips_today",
            "valueType": "INT64"
          },
          {
            "name": "rating",
            "valueType": "FLOAT32"
          }
        ],
        "ttl": "3600s",
        "batchSource": {
          "type": "BATCH_BIGQUERY ",
          "eventTimestampColumn": "timestamp",
          "bigQueryOptions": {
            "table": "feast-oss.demo_data.driver_activity"
          },
          "dataSourceClassType":
            "feast.infra.offline_stores.bigquery_source.BigQuerySource"
        },
        "online": true
      },
      "meta": {
        "materializationIntervals": [
          {
            "startTime": "2021-11-17T15:00:00Z",
            "endTime": "2021-11-17T17:01:00Z"
          }
        ]
      }
  }
}
```


The above example showcases a minified version of the registry file contents. Generally though, the complete version of it contains the following objects:

- Data Sources
- Entities
- Feature Services
- Feature Views
- On Demand Feature Views
- Request Feature Views
- Saved Datasets

We will explain more on these objects in the following section (see section 3.1.2).

Provider

A Provider is a transparent component of Feast. It is optional and users can explicitly define it in the configuration. Its purpose is to act like some kind of a middleman through which all Feast operations execute. Operations such as feature retrieval or infrastructure management etc. pass through the Provider first which executes any custom logic. Furthermore, it targets specific environments (GCP, AWS, Local) and is responsible for seamlessly orchestrating Feast components in them. For example, an AWS Provider makes sure that the Registry is going to be created in an S3 bucket and nowhere else.

Feature Repository

It is a git repo containing a configuration file feature_store.yaml and all the feature definitions (.py files). Having a central repository where users can manage the configuration and the feature definitions, written declaratively and stored as Python code, creates a robust source of truth. The Feature Repository indicates the desired state of the feature store and is used to configure, deploy and manage it. Eventually, all of the configuration and definitions are stored by users in the Registry which is programmatically accessible by other Feast components. Here is an example structure of the repo:

```
$ tree -a
.
├── data
│   └── driver_stats.parquet
├── driver_features.py
├── feature_store.yaml
└── .feastignore
```

Feature Server

Feast allows the creation of Feature Servers that provide HTTP endpoints which serve online features in JSON format. This functionality enables users to read features from the Online Store using any programming language that can execute HTTP requests.

Underneath, a Feature Server periodically reads the Registry's proto file and keeps it in cache. This way it is able to respond quickly to incoming requests by querying the Online Store immediately.

3.1.2 Core Concepts

Next up is an overview of core concepts and Feast specific terminology. This is an important section providing details on most of the Registry's first class objects and what their purpose is.

Project

Projects is a way of namespacing or grouping different kinds of Registry objects. A Project is logically related to one or more Feature Views, Data Sources, Entities etc. It also provides a complete isolation layer at the infrastructure level by namespacing resources, such as tables, using the Project's name as prefix. In other words it creates a completely separate universe of entities and features.

Data Source

A Data Source is an object that refers to raw underlying data such as an SQL table, a stream of data or a single file. Since Feast requires time-series data to perform its operations, all supported data sources must contain a timestamp field together with every row, entry or stream event. Out of the box various batch (Snowflake, BigQuery, Redshift etc.) and stream (Kafka, Kinesis, etc.) sources are supported by Feast, but it also allows users to add support for custom sources by using its plugin mechanism.

```
# Data Source definition
  driver_stats_source = BigQuerySource(
    table_ref="feast-oss.demo_data.driver_hourly_stats",
    event_timestamp_column="datetime",
    created_timestamp_column="created",
  )
```

Entity

In general, an entity is a collection of semantically related features. In Feast world an Entity is part of a Feature View and is defined by a name, a value type and a set of join keys. The join keys are important as Feast uses them during the lookup of feature values from the Online Store and the join process in point-in-time joins. They could be easily compared to primary keys in SQL databases and they uniquely describe a Feature View record.

```
# Entity definition
  driver = Entity(
    name="driver",
    value_type=ValueType.STRING,
    join_keys=["driver_id"]
  )
```

Feature View

A Feature View is a logical grouping of features, entities and data sources. To be more specific, it consists of zero or more Entities, one or more Features (Fields) and exactly one Data Source. It describes a view or a subset of the underlying data source. It's the object that actually allows modeling of existing feature data in a consistent way. The Features tied with the Feature View are essentially properties or characteristics of the related Entity.

Feast uses Feature Views during feature retrieval either from the Online Store or from the Offline Store. They also determine the storage schema in the Online Store which is crucial during the materialization process and online inference.

```
# Feature View definition
  driver_stats_fv = FeatureView(
    name="driver_activity",
    entities=["driver"],
    schema=[
      Field(name="trips_today", dtype=Int64),
      Field(name="rating", dtype=Float32),
    ],
```

```
    source=BigQuerySource(
      table="feast-oss.demo_data.driver_activity"
    )
  )
```

Another interesting kind of Feature View is the On Demand Feature View. It allows use of existing features and request time data (features only available at request time) to transform and create new features on the fly. This happens by specifying a transformation function which Feast executes both in historical and online feature retrieval operations.

```
# On Demand Feature View definition
  @on_demand_feature_view(
    sources=[
      driver_hourly_stats_view,
      input_request
    ],
    schema=[
      Field(name="conv_rate_plus_val1", dtype=Float64),
      Field(name="conv_rate_plus_val2", dtype=Float64)
    ]
  )
  def transformed_conv_rate(features_df: pd.DataFrame)
  -> pd.DataFrame:
    df = pd.DataFrame()
    df["conv_rate_plus_val1"] =
      (features_df["conv_rate"] + features_df["val_to_add"])
    df["conv_rate_plus_val2"] =
      (features_df["conv_rate"] + features_df["val_to_add_2"])
    return df
```

Feature Service

A Feature Service is an object that represents a logical group of Features from one or more Feature Views. It is usually a good way to match models with a group of features that they require to perform training or inference. This also results in better tracking and monitoring mechanisms for deployed models.

```
# Feature Service definition
  driver_stats_fs = FeatureService(
    name="driver_activity",
    features=[
      driver_stats_fv,
      driver_ratings_fv[["lifetime_rating"]]
    ]
  )
```

Dataset

A Dataset allows saving data frames that contain both entities and feature data which can be later used for model training. In addition, having a snapshot of data for a specific time range helps perform data analysis and quality monitoring. Behind the scenes, Feast stores Dataset's metadata in the Registry and actual feature data in the Offline Store.

```
# Dataset creation
historical_job = store.get_historical_features(
  features=["driver:avg_trip"],
  entity_df=["driver"],
)

dataset = store.create_saved_dataset(
  from_=historical_job,
  name='training_dataset',
  storage=SavedDatasetBigQueryStorage(
    table_ref='feast-oss.demo_data.driver_activity'
  )
)

dataset.to_df()
```

3.1.3 Usage

At this point, everything is set to move on with Feast operations and ways to use it. At first, we describe the initialization and teardown of Feast. Then, we explain typical

feature retrieval operations and finally present the materialization process in detail. To understand how all Feast components coordinate during these operations we use the Python SDK client as reference.

apply() - teardown() functions

As mentioned earlier we can find all feature definitions in the Feature Repository. To make these accessible from other Feast components we need to store them in the Registry. The process of parsing the Feature Repository, transforming it in a protobuf representation and storing it in the Registry is performed by *feast apply* CLI command. Underneath, after Feast validates and parses the repo, the *apply()* method of FeatureStore class runs.

```
def apply(
  self,
  objects: Union[
    DataSource,
    Entity,
    FeatureView,
    OnDemandFeatureView,
    RequestFeatureView,
    FeatureService,
    List[FeastObject],
  ],
  objects_to_delete: Optional[List[FeastObject]] = None,
  partial: bool = True,
): ...
```

It essentially gets two arguments: objects to create or update and objects to delete. At first, it stores the new or updated objects in the Registry, then it deletes the deprecated objects and finally it updates the infrastructure. The last action mainly refers to the creation and deletion of structures of the Online Store, where Feast stores the latest features values.

The execution of *teardown()* performs the reverse process. At first, it destroys all the existing infrastructure and then it deletes all the Registry objects.

get_historical_features() function

Getting historical features is probably the most common Feast operation. Its goal is to join multiple features from one or more Feature Views onto an entity data frame in a point-in-time correct way. An entity data frame which acts as the input of the operation is actually a set of records that contains a join key(s) and a timestamp for each record.

Entity Data Frame

row	event_timestamp	driver_id
0	2021-04-16 20:29:28+00:00	1001
1	2021-04-15 12:29:28+00:00	1003
2	2021-04-17 04:29:28+00:00	1002
3	2021-02-16 10:29:28+00:00	1000

The output is a reproducible state of features at a specific point in time.

Joined Training Data Frame

row	event_timestamp	driver_id	conv_rate	acc_rate
0	2021-04-16 20:29:28+00:00	1001	0.675539	0.657475
1	2021-04-15 12:29:28+00:00	1003	0.128302	0.913942
2	2021-04-17 04:29:28+00:00	1002	0.313097	0.770170
3	2021-02-16 10:29:28+00:00	1000	NULL	NULL

The process of populating the input data frame with the correct feature values is very simple. Assume the underlying data source is the following:

row	event_timestamp	driver_id	conv_rate	acc_rate
1804	2021-04-12 07:00:00+00:00	1001	0.373866	0.896520
1443	2021-04-15 07:00:00+00:00	1003	0.675539	0.657475
1082	2021-04-16 07:00:00+00:00	1001	0.128302	0.913942
721	2021-04-16 07:00:00+00:00	1002	0.802812	0.410884
456	2021-03-16 07:00:00+00:00	1000	0.402842	0.510674
360	2021-04-17 07:00:00+00:00	1002	0.313097	0.770170

For each row within the entity data frame Feast scans backward in time from event timestamp up to a maximum of the TTL time. For example, for row 0 (driver_id 1001)

it scans the underlying data source and gets rows 1804 and 1082. Then, it joins the row with the closest timestamp, thus row 1082. Row 3 (driver_id 1000) has an older timestamp than the earliest record of driver_id 1000 in the underlying data source, thus NULL values populate the result. Summarizing, the join process is very similar to an ordered left join, a loose one though, based on nearest timestamps.

To perform this operation, Feast uses the *get_historical_features()* method of the FeatureStore class.

```
def get_historical_features(
  self,
  entity_df: Union[pd.DataFrame, str],
  features: Union[List[str], FeatureService],
  full_feature_names: bool = False,
) -> RetrievalJob: ...
```

This method follows a lazy approach returning a RetrievalJob instead of the actual data. Getting the feature data requires executing *to_df()* or *to_arrow()* methods.

```
entity_df = pd.read_csv("entity_df.csv")

training_job = store.get_historical_features(
  entity_df=entity_df,
  features = [
    'driver_hourly_stats:conv_rate',
    'driver_hourly_stats:acc_rate'
  ],
)

training_df = training_job.to_df()
```

get_online_features() function

Getting online features is the operation Feast uses during online inference to retrieve the latest feature values from the Online Store. The *get_online_features()* method of the FeatureStore class is used by the SDK client for this reason.

```
def get_online_features(
  self,
  features: Union[List[str], FeatureService],
  entity_rows: List[Dict[str, Any]],
  full_feature_names: bool = False,
) -> OnlineResponse: ...
```

This time there is no need for timestamps, users provide a set of join keys and get back a new data frame with the requested feature values. The operation is not lazy as with *get_historical_features()*. It returns feature values in protobuf format and then the client uses the *to_df()* or *to_dict()* methods to transform them in a useful format.

```
online_response = fs.get_online_features(
  features=[
    "driver_hourly_stats:conv_rate",
    "driver_hourly_stats:acc_rate"
  ],
  entity_rows=[{"driver_id": 1001}, {"driver_id": 1002}],
)
online_response_dict = online_response.to_dict()
```

materialize() function

Last but not least, there is the materialization process. It is the process responsible for refreshing Online Store's feature values. A user provides a time interval and specifies a set of Feature Views they want to materialize.

```
def materialize(
  self,
  start_date: datetime,
  end_date: datetime,
  feature_views: Optional[List[str]] = None,
) -> None: ...
```

Feast queries the batch sources for all Feature Views over the provided time range and loads the latest feature values into the Online Store. The query slices the underlying data

source keeping only the requested time range, then it sorts the remaining records based on the timestamp and finally keeps the latest for each unique join key(s).

Step 1: Sliced Records

row	event_timestamp	driver_id	conv_rate	acc_rate
1804	2021-04-12 07:00:00+00:00	1001	0.373866	0.896520
1082	2021-04-16 07:00:00+00:00	1001	0.128302	0.913942
1443	2021-04-15 07:00:00+00:00	1003	0.675539	0.657475
360	2021-04-17 07:00:00+00:00	1002	0.313097	0.770170
721	2021-04-16 07:00:00+00:00	1002	0.802812	0.410884

Step 2: Sorted Records

row	event_timestamp	driver_id	conv_rate	acc_rate
1804	2021-04-12 07:00:00+00:00	1001	0.373866	0.896520
1443	2021-04-15 07:00:00+00:00	1003	0.675539	0.657475
1082	2021-04-16 07:00:00+00:00	1001	0.128302	0.913942
721	2021-04-16 07:00:00+00:00	1002	0.802812	0.410884
360	2021-04-17 07:00:00+00:00	1002	0.313097	0.770170

Step 3: Latest Feature Values

driver_id	conv_rate	acc_rate
1003	0.675539	0.657475
1001	0.128302	0.913942
1002	0.313097	0.770170

3.2 Registry Architecture

In order to identify what is missing and what we need to do in terms of design and architecture, we need to completely understand the Registry architecture. This section focuses mainly on two Python classes, the Registry and RegistryStore classes, of the Python SDK. These classes are responsible for managing Registry objects and interacting with the storage layer.

3.2.1 Functionality

Almost every single Feast operation uses the Registry. Whenever a user needs to get feature data either from the Offline or the Online Store they use a feature definition to determine the feature they want to get. In the background:

1. Feast executes a query to the Registry to get all the required metadata (e.g. where the actual feature data is located, the type of the feature, etc.).
2. It executes another query to fetch the actual feature data.

Regardless if a user wants to get historical feature data to create a dataset or get online feature data to perform inference the Feast client (SDK) requires that metadata (1). However, these two operations are completely opposite:

- Getting historical feature data is an I/O bound operation that spends most of the time fetching actual data
- Getting online feature data is a fast operation that is crucial for high-performance inference

In order for Feast SDK to support high performance inference, without needing to fetch that metadata every single time, it keeps a cached registry representation in-memory and makes sure to refresh it when it expires by fetching the upstream registry representation. To do so, it uses a commit & refresh mechanism throughout its operations.

Before explaining the commit & refresh mechanism let's see what a Registry and a RegistryStore object are. They are instances of the two following classes that Feast SDK implements:

- **Registry**: core class responsible for managing the Registry component (adding new features, deleting old ones, etc.) and keeping the cached registry representation in sync. It implements the commit & refresh mechanism.
- **RegistryStore**: extendable class responsible for interacting with a specific registry store backend.

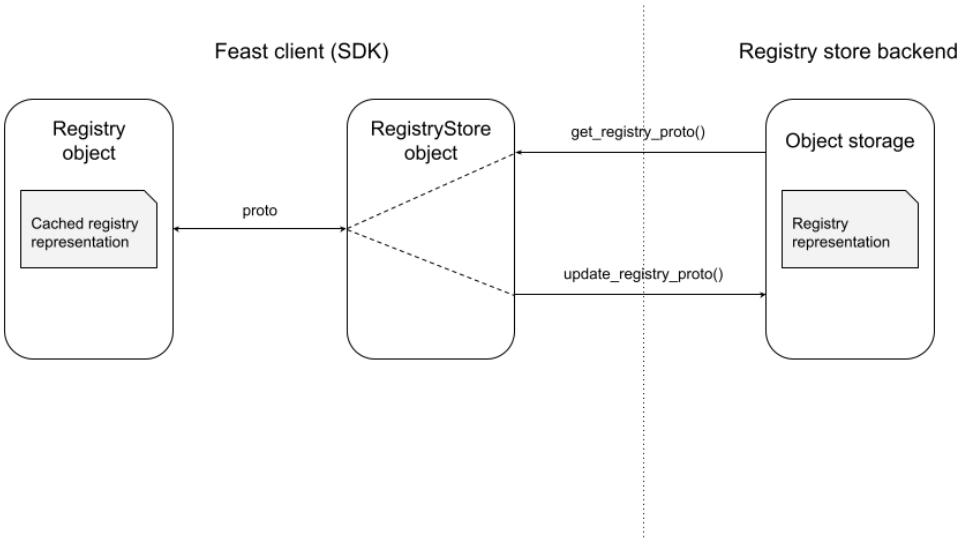


Figure 3.3: *Commit & Refresh Mechanism*

The Registry class contains methods which apply changes to the registry file. These methods provide a boolean argument called commit. For example:

```
def apply_entity(
  self,
  entity: Entity,
  project: str,
  commit: bool = True
): ...
  if commit:
    self.commit()
  return

def delete_entity(
  self,
  name: str,
  project: str,
  commit: bool = True
): ...
  if commit:
    self.commit()
  return
```

This argument specifies whether a change will stay in cache or will be pushed to the

object storage by the Feast client. In case commit = True, the Registry object will pass the entire cached registry representation to the RegistryStore object which will then push it to the actual object storage and replace the old registry representation.

For the same reason, Registry class methods responsible for ”read” operations provide an allow_cache option. If allow_cache = True, the Registry object is allowed to use the cached registry representation. If not, it has to refresh it first by asking the RegistryStore object to fetch it from the object storage.

```
def get_entity(
  self,
  name: str,
  project: str,
  allow_cache: bool = False
):
  registry_proto = self._get_registry_proto(allow_cache=allow_cache)
  ...

def list_entities(
  self,
  project: str,
  allow_cache: bool = False
):
  registry_proto = self._get_registry_proto(allow_cache=allow_cache)
  ...
```

Both of these arguments along with commit() and refresh() methods form the commit & refresh mechanism. Let’s have a closer look at these methods.

```
def commit(self):
  if self.cached_registry_proto:
    self._registry_store.update_registry_proto(
      self.cached_registry_proto
    )

def refresh(self):
  self._get_registry_proto(allow_cache=False)
```

```
def _get_registry_proto(self, allow_cache: bool = False)
-> RegistryProto:
  ...
  registry_proto = self._registry_store.get_registry_proto()
  ...
```

Both of these methods essentially use a RegistryStore object by calling the following two methods that belong to it:

- update_registry_proto
- get_registry_proto

3.2.2 RegistryStore Plugin

The RegistryStore class is the programmatic client that Feast uses to interact with the underlying storage backend of the Feast Registry. It is part of the plugin mechanism that allows different kinds of storage backends to work alongside Feast. To be more specific it is an abstract class that contains the following 3 methods:

- get_registry_proto(self) -> RegistryProto
- update_registry_proto(self, registry_proto: RegistryProto)
- teardown(self)

Developers that want to support a new storage backend need to create a new class that implements these methods. Here is an example:

```
class GCSRegistryStore(RegistryStore):
  ...
  def get_registry_proto(self):
    ...
    if storage.Blob(bucket=bucket, name=self._blob).
        exists(self.gcs_client):
      self.gcs_client.download_blob_to_file(
        self._uri.geturl(), file_obj, timeout=30
      )
      file_obj.seek(0)
```

```
      registry_proto.ParseFromString(file_obj.read())
      return registry_proto
    ...

  def update_registry_proto(self, registry_proto: RegistryProto):
    self._write_registry(registry_proto)

  def teardown(self):
    ...
    gs_bucket = self.gcs_client.get_bucket(self._bucket)
    try:
      gs_bucket.delete_blob(self._blob)
    ...

  def _write_registry(self, registry_proto: RegistryProto):
    ...
    gs_bucket = self.gcs_client.get_bucket(self._bucket)
    blob = gs_bucket.blob(self._blob)
    file_obj = TemporaryFile()
    file_obj.write(registry_proto.SerializeToString())
    file_obj.seek(0)
    blob.upload_from_file(file_obj)
```

It is clear that in all of the three implemented methods there is a GCS client which is responsible for creating, updating, deleting or fetching the registry file from the object storage.

3.2.3 Problematic Designs

Having seen how everything works it's time to expose designs that make the current architecture difficult to use in a multi-user environment, where users need to share specific parts of the registry. Before moving forward, it is important to keep in mind that the registry representation which Feast transfers and stores in a protobuf format contains a list for each of the registry's first-class objects (see section 3.1.1).

Now, storing an entire registry as a single file creates problems with sharing the registry's first-class objects in a multi-user environment. There is no clear way to configure permissions on specific objects as only read or write permissions can be configured on the

registry file level. Adding to this, moving entire registry files between the client and the object storage degrades performance.

Another aspect of the registry that could potentially cause problems is the cached registry representation. In scenarios where multiple users interact with the same registry, keeping it in sync on the client side (Feast SDK), while also making sure changes performed locally by users find their way to the object storage is challenging. At this point, no locking mechanism exists on the "remote" registry file making it impossible to guarantee atomic access on concurrent writes.

3.3 New Architecture

Solving or overcoming the above problematic designs is essential for integrating Feast into Kubeflow. In this section we expose possible new designs and solutions. The focus remains on the Registry and its functionality.

At the moment, Feast is basically a Python SDK that manages and orchestrates other components such as the Offline Store, the Online Store and the Registry. However, all of these three components rely on vendor specific infrastructure such as storage buckets or data warehouses. Having an open source project like Kubeflow requires components that are vendor agnostic and can be used in various Feast deployments.

The purpose of the entire effort is to extend the Feast functionality and flexibility to integrate it into Kubeflow. This will happen by designing a new Registry backend, a complete API server, which adds an extra layer on top of the stored Registry objects. Adding this management layer will make Feast even more extensible and lay the foundations for other improvements as well. A standalone Registry backend that is backed up by an SQL database, which stores Registry objects, and that is able to perform access control is the final goal.

3.3.1 Initial Design

The initial design aims at avoiding using a single file registry as the storage backend. Thus, we will replace the object storage with a relational database.

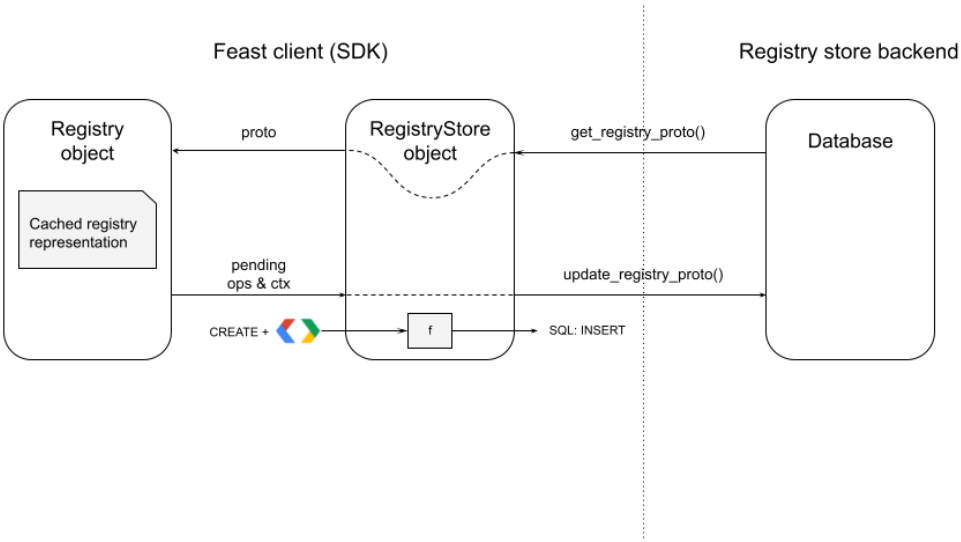


Figure 3.4: *Registry as Database*

This design also improves the way Feast manages the cached registry representation, as typical DBMS offer ACID properties useful for avoiding conflicts in concurrent write operations. When users try to change the state of the registry simultaneously, the use of transactions ensures that every successful write operation is persisted.

In this new design, fetching the registry remains as simple as performing the required SELECT SQL queries against the database. Updating the registry is a bit more complex. The Registry object keeps a FIFO list of pending operations and their context (in protobuf format). This queue contains the changes that the Feast client needs to push since the last commit. The context refers to the actual registry object in its protobuf representation. For example:

```
[("CreateEntity", entityProto), ("DeleteFeatureView", fvProto), ...]
```

Every time Feast triggers the commit mechanism a similar list gets passed to the RegistryStore object which translates it to the corresponding SQL queries and executes them. The list empties after every successful commit.

Database Schema

Since this is a new approach, there is a need to design a database schema that stores the registry objects. The new schema follows the schema definition of the registry and its components [2], the one used for protobuf messages.

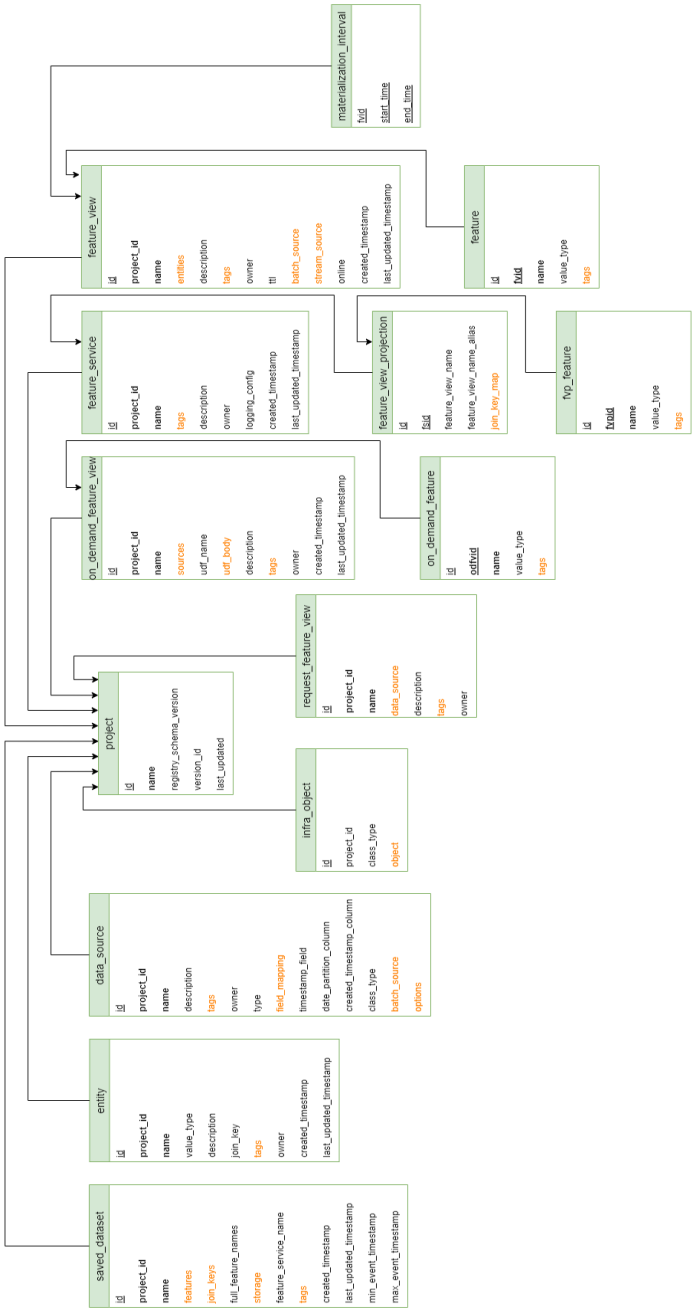


Figure 3.5: *Database Schema*

Comments All tables contain a unique id. In addition, there are other uniqueness constraints. For all first-class objects the combination of project_id and name must be unique. The same thing applies for the projects table, where the project name must be unique as well. We store fields that are colored with orange in the database schema as bytes as they contain complex objects like JSON objects or lists.

3.3.2 Improved Design

Having set a solid background (a database as the registry store backend), it is now possible to build another layer on top of it in order to manage permissions on specific registry objects.

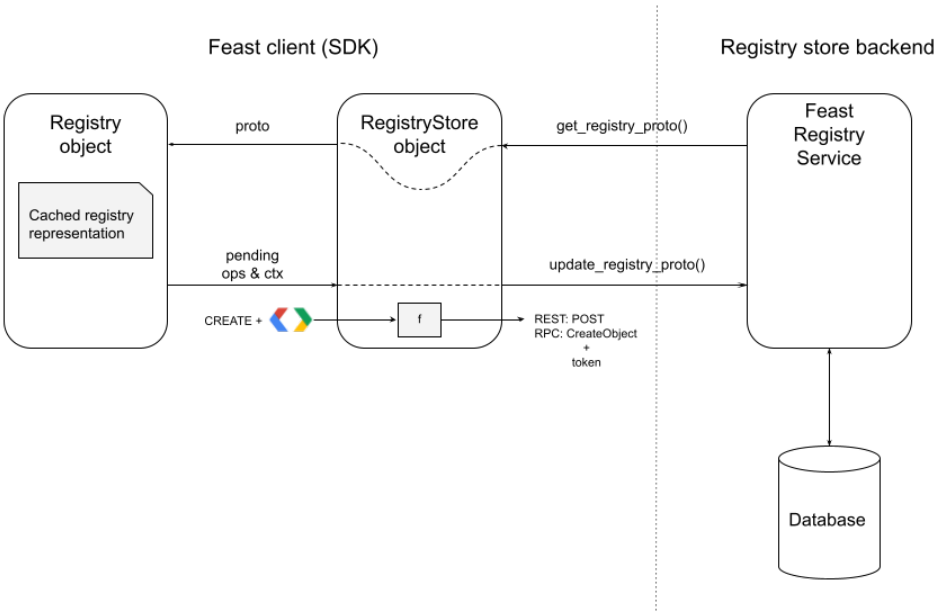


Figure 3.6: *New Architecture*

This layer acts as a middleman between the client side and the actual database enforcing access control. It is a simple REST API server on top of a gRPC server [13]. From now on this layer will be called Feast Registry Service (FRS) and it has the following three functionalities:

- Expose CRUD endpoints for all first-class objects of the registry.
- Perform authentication and authorization of requests.

- Manage interactions with the database.

The CRUD endpoints look like the following:

```
POST: /createEntity body: entity_proto
GET: /getEntity body: entity_name, entity_project
POST: /updateEntity body: entity_proto
DELETE: /deleteEntity body: entity_name, entity_project
```

On the client side the RegistryStore object translates the list of pending operations to API calls and enriches them with a token. This token is then used by the FRS during the authentication and authorization process.

Projects and Isolation

Until now we have introduced all the mechanisms required to manage permissions on registry objects. However, it is also important to explain the levels of isolation that we offer in a Kubeflow environment. As we already explained in the Background chapter (see section 2.2.3) Kubernetes provides a mechanism called Namespaces to isolate resources. Moreover, Feast provides Projects (see section 3.1.2) that offer an isolated environment of features and entities.

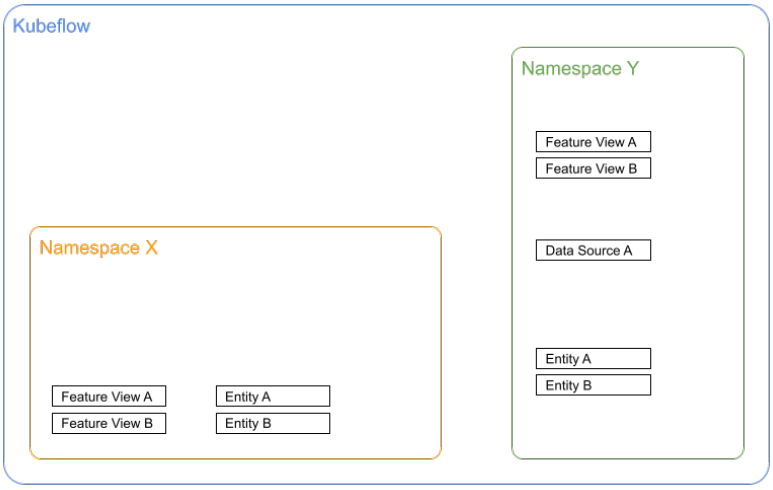


Figure 3.7: *Isolation Levels*

In a collaborative environment like Kubeflow multiple objects need to exist and be shared among users. Project names in Feast must be unique, thus we use the 1:1 relation between a Feast project and a Kubeflow namespace.

To share objects or entire namespaces, admins give permissions to specific users or groups to perform actions on resources in namespaces using K8s Roles (ClusterRoles) and RoleBindings (ClusterRoleBindings).

3.3.3 Registry Service

The design and architecture of the Feast Registry Service is heavily inspired by Kubeflow Pipelines (KFP) backend apiserver [4]. It follows a very similar structure and is as follows:

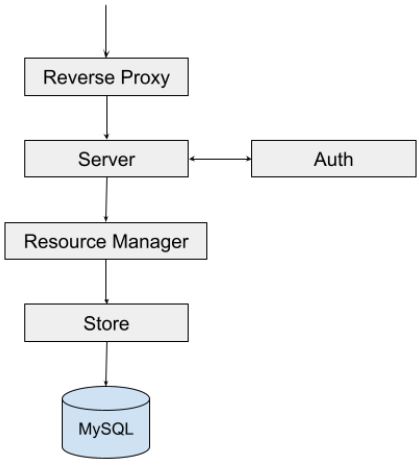


Figure 3.8: *Registry Service Architecture*

It has 5 main components that have different roles and perform different actions during each request.

Reverse Proxy It receives HTTP requests from clients, transforms them into gRPC requests and sends them to the Server. The reverse process happens after a gRPC response is generated from the Server.

Server It is a typical gRPC server [20]. It is the entrypoint of gRPC requests and implements functions that handle them. It executes validation checks, it uses the Auth component to perform authentication and authorization and passes the request to the underlying Resource Manager.

Auth It receives a token and a set of permissions from the Server component. It then passes the token to a TokenReview client which performs authentication. A TokenReview client is a K8s client [22] which interacts with a native K8s API. Authorization is executed by a SubjectAccessReview client using both the token and the set of permissions received by the Server. This is another K8s client that interacts with features provided by the authorization.k8s.io API [21].

Resource Manager It receives a request from the Server and transforms it to an internal representation (model) that can be handled by lower level components. It manages resources at a high level and is responsible for engaging the appropriate low level Stores in order to execute the correct changes in the underlying database.

Store It receives a model and performs an action on it that gets persisted in the database. Thus, it interacts with the actual storage system by executing queries or transactions. In this case the storage system refers to a MySQL database. However, it can interact with other types of SQL databases since we choose to use an ORM (Object-Relational Mapping) mechanism.

3.3.4 Registry Client

The design of the Registry Client is pretty simplistic. The goal is to be a component that leverages the plugin mechanism Feast offers. At its core it programmatically executes HTTP requests to the Feast Registry Service.

RegistryStore It is responsible for setting up and managing the client that communicates with the FRS. It fills requests with the proper configuration and the required token. If the token is not valid any more, it refreshes it.

Proto to API As we mentioned earlier the Registry object is passing a list of operations and their context to the RegistryStore object. The context though is in protobuf format. Before sending an HTTP request we need to convert the context to a proper API object.

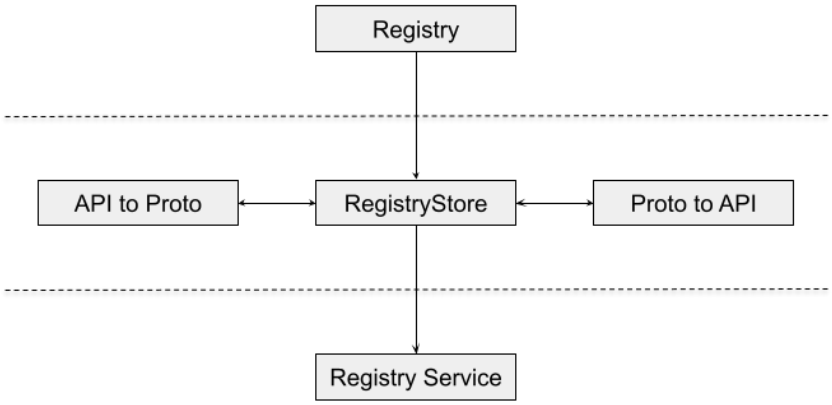


Figure 3.9: *Registry Client Architecture*

API to Proto The Registry Service responds with one or multiple API objects. However, the RegistryStore object must pass a complete registry representation to the Registry object. Thus, we need to properly transform the received objects to a protobuf representation.

4 Implementation

This chapter focuses on presenting all the technical details and nuances needed to implement the newly designed architecture. At first, it describes all the minor changes in the existing Feast code base. Then, it showcases the entire effort regarding the new Feast Registry Service. Last but not least, it presents the new pluggable Registry client. Along with all implementation specific details, it thoroughly explains deployment or installation steps wherever needed.

4.1 Overview

Having already described all the new components that we need to create, it's important to highlight that the goal of the implementation is to introduce new functionality without compromising compatibility of the existing one. Thus, we perform minimal changes on the existing code base. In addition, entirely new components try to provide solutions that create abstractions and we can easily extend and use them outside a Kubeflow environment.

Since, Feast SDK client is written in Python and this is the most popular and easy way to get started with Feast, we use Python for both core changes of Feast and the new Registry client. However, we implement the Feast Registry Service using Go as plenty of cloud native applications are already developed in Go and the end result is highly performant. Moreover, we reuse part of KFP backend code and reduce the total time spent on implementing the same mechanisms in another programming language.

4.2 The Feast Core

We need to extend the Registry class so as to properly handle the pending ops list. As explained previously this is a FIFO list. To implement it we will use the queue [19] module.

```
class Registry:
  cached_registry_proto: Optional[RegistryProto] = None
  cached_registry_proto_created: Optional[datetime] = None
  cached_registry_proto_ttl: timedelta
  pending_ops: queue.Queue
```

When executing "write" operations we fill this queue with dicts that contain the type of operation and the necessary context. Here are some examples:

```
# CreateEntity and UpdateEntity example
def apply_entity(
  self, entity: Entity, project: str, commit: bool = True
):
  update = False
  entity.is_valid()
  ...

  for idx, existing_entity_proto in enumerate(
    self.cached_registry_proto.entities
  ):
    if (
      existing_entity_proto.spec.name == entity_proto.spec.name
      and existing_entity_proto.spec.project == project
    ):
      del self.cached_registry_proto.entities[idx]
      self.pending_ops.put({
        "op": "UpdateEntity", "proto": entity_proto
      })
      update = True
      break

  self.cached_registry_proto.entities.append(entity_proto)
  if not update:
```

```
    self.pending_ops.put({
      "op": "CreateEntity", "proto": entity_proto
    })
  if commit:
    self.commit()
```

```
# DeleteEntity example
def delete_entity(
  self, name: str, project: str, commit: bool = True
):
  self._prepare_registry_for_changes()
  assert self.cached_registry_proto

  for idx, existing_entity_proto in enumerate(
    self.cached_registry_proto.entities
  ):
    if (
      existing_entity_proto.spec.name == name
      and existing_entity_proto.spec.project == project
    ):
      del self.cached_registry_proto.entities[idx]
      self.pending_ops.put({"op": "DeleteEntity", "name": name})
      if commit:
        self.commit()
      return

  raise EntityNotFoundException(name, project)
```

At some point we need to commit changes in the Registry. For this reason, we pass the pending ops list to the RegistryStore object.

```
def commit(self):
  if self.cached_registry_proto:
    self._registry_store.update_registry_proto(
      self.cached_registry_proto, self.pending_ops
    )
```

However, the RegistryStore abstract class contains an abstract method that updates the registry proto with only one argument. The new method has one extra argument with

a default value (an empty queue) in case the pending ops list is not useful for other RegistryStore class implementations.

```
# Old definition
def update_registry_proto(self, registry_proto: RegistryProto):

# New definition
def update_registry_proto
  self,
  registry_proto: RegistryProto,
  pending_ops: queue.Queue = queue.Queue()
):
```

4.3 Registry Service

To create the Registry Service we use patterns similar to the KFP backend apiserver. The process that we describe in the rest of this section refers to Entity objects and acts as an easy to follow example. However, the same process applies for all other first-class Registry objects.

Protobuf Definitions

In the first step of the process we define the endpoints by creating the required protobuf definitions. The following example presents an EntityService with five distinct endpoints: Create, Get, Update, Delete and List.

```
service EntityService {
  rpc CreateEntity (CreateEntityRequest) returns (Entity) {
    option (google.api.http) = {
      post: "/CreateEntity",
      body: "entity"
    };
  }

  rpc GetEntity (GetEntityRequest) returns (Entity) {
    option (google.api.http) = {
      get: "/GetEntity"
```

```
    };
  }

  rpc UpdateEntity (UpdateEntityRequest) returns (Entity) {
    option (google.api.http) = {
      post: "/UpdateEntity",
      body: "entity"
    };
  }

  rpc DeleteEntity (DeleteEntityRequest) returns (google.protobuf.Empty) {
    option (google.api.http) = {
      delete: "/DeleteEntity"
    };
  }

  rpc ListEntities (ListEntitiesRequest) returns (ListEntitiesResponse) {
    option (google.api.http) = {
      get: "/ListEntities"
    };
  }
}
```

Since the gRPC framework relies on clients that can directly call a method on a server application on a different machine, the above definitions act as the source of truth. We use a compiler to translate these definitions into actual code which we use as a stub client providing the same methods as the server. On the server side, it is the developer's responsibility to implement these methods and create a server that handles client requests.

Server

So, the next step is actually implementing these methods. To begin with, let's compare how these methods look both on the client and the server side.

On the client side, here is an example of the generated code a compiler creates:

```go
func (c *entityServiceClient) CreateEntity(
    ctx context.Context, in *CreateEntityRequest, opts ...grpc.CallOption
  ) (*Entity, error) {
  out := new(Entity)
  err := c.cc.Invoke(
    ctx, "/api.EntityService/CreateEntity", in, out, opts...
  )
  if err != nil {
    return nil, err
  }
  return out, nil
}
```

On the server side, this is how we implement the method:

```go
func (s *EntityServer) CreateEntity(
    ctx context.Context, request *api.CreateEntityRequest
  ) (*api.Entity, error) {
  resourceAttributes := &authorizationv1.ResourceAttributes{
    Namespace: request.Namespace,
    Verb:      common.RbacResourceVerbCreate,
  }
  err := s.haveAccess(ctx, resourceAttributes)
  if err != nil {
    return nil, util.Wrap(err, "Failed to authorize the request.")
  }

  entity, err := s.resourceManager.CreateEntity(
    request.Entity, request.Namespace
  )
  if err != nil {
    return nil, util.Wrap(err, "Create entity failed.")
  }

  return ToApiEntity(entity), nil
}
```

Authentication and Authorization

As a request reaches the server, the server performs access control before actually executing the requested operation. At first, it extracts the token from the request and

initiates a TokenReview. A TokenReview is actually a request to the TokenReview API which validates the token and ensures it is intended for the specified audience, in this case *features.kubeflow.org*.

```
func (tra *TokenReviewAuthenticator) doTokenReview(
    ctx context.Context, userIdentity string
  ) (*authv1.UserInfo, error) {
  review, err := tra.client.Create(ctx,
    &authv1.TokenReview{
      Spec: authv1.TokenReviewSpec{
        Token:     userIdentity,
        Audiences: tra.audiences,
      },
    },
    v1.CreateOptions{},
  )
  ...
  return &review.Status.User, nil
}
```

After this, the server authorizes the request. This time it uses the user identity along with a set of resource attributes to perform a SubjectAccessReview. A SubjectAccessReview is again a request to the respective API which ensures that a user can perform an action (verb) on a specific resource of a namespace.

```
func (r *ResourceManager) IsRequestAuthorized(
    ctx context.Context, userIdentity string, userGroups []string,
    resourceAttributes *authorizationv1.ResourceAttributes
  ) error {
  result, err := r.subjectAccessReviewClient.Create(ctx,
    &authorizationv1.SubjectAccessReview{
      Spec: authorizationv1.SubjectAccessReviewSpec{
        ResourceAttributes: resourceAttributes,
        User:               userIdentity,
        Groups:             userGroups,
      },
    },
    v1.CreateOptions{},
  )
  ...
  return nil
}
```

For example, a */CreateEntity* request which uses the following resource attributes:

- Namespace: kubeflow-user
- Verb: create
- Group: features.kubeflow.org
- Version: v1beta1
- Resource: entities

is authorized by the server only if the user who makes the request is allowed to execute *create* on *entities* of *features.kubeflow.org* version *v1beta1* on *kubeflow-namespace* namespace.

Resource Manager

The next step of the process is to instruct the Resource Manager to perform the requested operation.

```
func (r *ResourceManager) CreateEntity(
    apiEntity *api.Entity, namespace string
  ) (*model.Entity, error) {
  project, err := r.projectStore.GetProject(apiEntity.Project, namespace)
  if err != nil {
    return nil, util.Wrap(err, "Failed to find project")
  }

  entity, err := r.ToModelEntity(apiEntity, "", project.Id)
  if err != nil {
    return nil, util.Wrap(err, "Failed to convert entity model")
  }

  return r.entityStore.CreateEntity(entity)
}
```

Since the Resource Manager can manage all of the Registry objects, it starts by finding the project this new Entity belongs to and uses its projectId along with request's Entity

to create a model (an internal representation of an object). It then passes this model to the appropriate underlying Store where the projectId will be used by the Store as a foreign key.

Store

The final step of the process involves the Store which receives an Entity model and creates the respective SQL query. Assuming that the SQL query is valid, it executes it and checks for possible errors.

```
func (s *EntityStore) CreateEntity(
    e *model.Entity
  ) (*model.Entity, error) {
  newEntity := *e
  id, err := s.uuid.NewRandom()
  ...
  newEntity.Id = id.String()

  sql, args, err := sq.
    Insert("entities").
    SetMap(
      sq.Eq{
        "id":                      newEntity.Id,
        "project_id":              newEntity.ProjectId,
        "name":                    newEntity.Name,
        "value_type":              newEntity.ValueType,
        "description":             newEntity.Description,
        "join_key":                newEntity.JoinKey,
        "tags":                    newEntity.Tags,
        "owner":                   newEntity.Owner,
        "created_timestamp":       newEntity.CreatedTimestamp,
        "last_updated_timestamp": newEntity.LastUpdatedTimestamp,
      }).
    ToSql()
  ...
  _, err = s.db.Exec(sql, args...)
  if err != nil {
    ...
  }

  return &newEntity, nil
}
```

The process completes by sending a response back to the client that made the request. Before doing so, the Server transforms the model back to a proper API object.

This end-to-end process is almost identical for the rest of the Registry objects. However, there are objects such as Feature Views that need to handle e.g. their child Features and their corresponding database entries. Implementation wise it gets a bit more complex as we need to use typical database transactions to ensure consistency. Here is an example:

```
func (s *FeatureViewStore) CreateFeatureView(
    fv *model.FeatureView
  ) (*model.FeatureView, error) {
  newFeatureView := *fv
  ...

  tx, err := s.db.Begin()
  if err != nil {
    ...
  }

  _, err = tx.Exec(sql, args...)
  if err != nil {
    tx.Rollback()
    ...
  }

  for _, feature := range newFeatureView.Features {
    _, err = s.featureStore.CreateFeature(
      tx, feature, newFeatureView.Id
    )
    if err != nil {
      tx.Rollback()
      ...
    }
  }

  err = tx.Commit()
  if err != nil {
    tx.Rollback()
    ...
  }

  return &newFeatureView, nil
}
```

In case an error occurs during the transaction, we perform a rollback to ensure a consistent database state. If all queries execute successfully, we finally persist the result by committing the transaction.

4.4 Registry Client

Creating the Registry Client is all about implementing the three abstract methods of the RegistryStore class. Let's see how everything works step by step.

Initializing the RegistryStore includes setting up an ApiClient and its Configuration. The frs_api Python package we use, is already automatically generated by the protobuf definitions of the Registry Service (see section 4.3). As a result it takes away all the heavy work of sending a request and handling the response. The Configuration object we pass to the ApiClient provides a function that is responsible for refreshing the token which we need to include in the requests' authorization header. We can find the token either on a file or an env variable.

```
class KubeflowRegistryStore(RegistryStore):
  def __init__(self, registry_config: RegistryConfig, repo_path: Path):
    ...
    config = frs_api.configuration.Configuration()
    config.host = registry_config.path

    config.api_key['authorization'] = 'token'
    config.api_key_prefix['authorization'] = 'Bearer'
    config.refresh_api_key_hook =
      ServiceAccountTokenVolumeCredentials().refresh_api_key_hook

    api_client = frs_api.api_client.ApiClient(configuration=config)

    self._entity_api = frs_api.api.entity_service_api.
      EntityServiceApi(api_client)
    ...
```

Purpose of the get_registry_proto() method is to fetch the entire registry representation. At first, it fetches project specific metadata which verifies that the project exists. Then it performs a list request for all first-class Registry objects to gather the stored definitions of the project's objects.

```
@log_exceptions_and_usage(registry="kubeflow")
def get_registry_proto(self):
  try:
    if not self.readMode:
      proj = self._project_api.project_service_get_project(
        project=self.project,
        namespace=self.namespace
      )
  except frs_api.exceptions.ApiException as e:
    raise FileNotFoundError(
      f'Project named "{self.project}" not found.'
    )
  self._get_entities(registry_proto)
  ...
  return registry_proto

def _get_entities(self, registry_proto: RegistryProto):
  res = self._entity_api.entity_service_list_entities(
    project=self.project,
    namespace=self.namespace
  )

  if not res.entities:
    return

  for e in res.entities:
    entity_proto = entity_to_proto(e)
    registry_proto.entities.append(entity_proto)
```

Purpose of the update_registry_proto() method is to change the state of the stored registry representation and it does it in a fine-grained way by performing only necessary changes. Based on the list of operations it receives from the Registry objects, it performs the respective API calls along with the required context. After the update successfully completes, the list empties.

```
@log_exceptions_and_usage(registry="kubeflow")
def update_registry_proto(
    self,
    registry_proto: RegistryProto,
    pending_ops: queue.Queue = queue.Queue()
  ):
```

```
    while not pending_ops.empty():
      op = pending_ops.get()
      ...
      elif op['op'] == 'CreateEntity':
        entity = entity_to_api(op['proto'])
        self._entity_api.entity_service_create_entity(
          body=entity,
          namespace=self.namespace
        )
      elif op['op'] == 'UpdateEntity':
        entity = entity_to_api(op['proto'])
        self._entity_api.entity_service_update_entity(
          body=entity,
          namespace=self.namespace
        )
      elif op['op'] == 'DeleteEntity':
        self._entity_api.entity_service_delete_entity(
          name=op['name'],
          project=self.project,
          namespace=self.namespace
        )
      ...
```

The goal of the teardown() method is to delete everything related to the specified project. A simple API call is enough as the Registry Service executes the entire logic.

```
@log_exceptions_and_usage(registry="kubeflow")
def teardown(self):
  self._project_api.project_service_delete_project(
    project=self.project,
    namespace=self.namespace
  )
```

Transform Objects

As we already explained in the Design chapter, the Registry object passes a protobuf registry representation to the RegistryStore object. Thus, we need a way to transform protobuf objects to proper API objects and vice versa. We implement functions such as entity_to_api() or entity_to_proto() for this reason:

```
def entity_to_api(proto: EntityProto) -> ApiEntity:
  return ApiEntity(
    name=proto.spec.name,
    project=proto.spec.project,
    value_type=proto.spec.value_type,
    description=proto.spec.description,
    join_key=proto.spec.join_key,
    tags=dict(proto.spec.tags),
    owner=proto.spec.owner,
    created_timestamp=proto.meta.created_timestamp.ToDatetime().
      replace(tzinfo=timezone.utc),
    last_updated_timestamp=proto.meta.last_updated_timestamp.
      ToDatetime().replace(tzinfo=timezone.utc)
  )

def entity_to_proto(api: ApiEntity) -> EntityProto:
  meta = EntityMetaProto()
  meta.created_timestamp.FromDatetime(getattr(
    api,
    "created_timestamp",
    datetime(1970, 1, 1, tzinfo=timezone.utc))
  )
  meta.last_updated_timestamp.FromDatetime(getattr(
    api,
    "last_updated_timestamp",
    datetime(1970, 1, 1, tzinfo=timezone.utc))
  )

  spec = EntitySpecProto(
      name=api.name,
      project=api.project,
      value_type=ValueType[api.value_type].value,
      description=api.description,
      join_key=api.join_key,
      tags=api.tags,
      owner=api.owner,
    )

  return EntityProto(spec=spec, meta=meta)
```

4.5 Cluster Configuration

4.5.1 Registry Service

After developing the FRS, the next step is to actually deploy the service in Kubeflow. Kubeflow deploys all services under kubeflow namespace. We will follow the same approach for the FRS.

Deploying the MySQL database the FRS is using is as simple as creating a StatefulSet and a Service in K8s.

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql-feast-sts
spec:
  replicas: 1
  serviceName: mysql-feast-svc
  selector:
    matchLabels:
      app: mysql-feast
  template:
    metadata:
      name: mysql-feast
      labels:
        app: mysql-feast
    spec:
      containers:
      - name: mysql
        image: bitnami/mysql:8.0
        volumeMounts:
          - name: data
            mountPath: /bitnami/mysql/data
        env:
          - name: MYSQL_USER
            value: "feastadmin"
          - name: MYSQL_PASSWORD
            value: "feastadmin"
          - name: MYSQL_DATABASE
            value: "feast"
        ports:
          - name: mysql
            containerPort: 3306
```

```
            protocol: TCP
        ...
---
apiVersion: v1
kind: Service
metadata:
  name: mysql-feast-svc
spec:
  type: ClusterIP
  ports:
  - name: mysql
    port: 3306
    targetPort: mysql
  selector:
    app: mysql-feast
```

A StatefulSet ensures that no data will be lost if a Pod goes down unexpectedly and a new one takes its place.

In order to deploy the API server, we need to create a ConfigMap [8] first. This contains a json file with all the configuration, e.g. database config, the server needs to function. Then, we create a typical Deployment and a Service.

```
apiVersion: v1
data:
  # apiserver assumes the config is named config.json
  config.json: |
    {
      "DBConfig": {
        "DriverName": "mysql",
        "DBName": "feast",
        "GroupConcatMaxLen": "4194304",
        "ConMaxLifeTime": "120s",
        "Host": "mysql-feast-svc",
        "User": "feastadmin",
        "Password": "feastadmin"
      },
      "InitConnectionTimeout": "10s",
      "MULTIUSER": true
    }
kind: ConfigMap
metadata:
  name: frs-config
```

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: feast-registry-d
  labels:
    app: feast-registry
spec:
  selector:
    matchLabels:
      app: feast-registry
  template:
    metadata:
      labels:
        app: feast-registry
    spec:
      containers:
      - name: feast-registry-api-server
        env:
        - name: POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
        image: apiserver
        imagePullPolicy: Never
        command:
          - /tmp/apiserver
          - --config=/etc/frs-config
        ports:
        - containerPort: 8888
        - containerPort: 8887
        volumeMounts:
        - name: config-volume
          mountPath: /etc/frs-config
      volumes:
        - name: config-volume
          configMap:
            name: frs-config
---
apiVersion: v1
kind: Service
metadata:
  name: feast-registry-svc
spec:
  type: ClusterIP
```

```
  ports:
  - name: http
    port: 8888
    protocol: TCP
    targetPort: 8888
  - name: grpc
    port: 8887
    protocol: TCP
    targetPort: 8887
  selector:
    app: feast-registry
```

The server exposes two ports (8888, 8887), one for the HTTP server and another one for the gRPC server.

4.5.2 Authentication Support

In order to support authentication, we use a token based approach. For this reason, we require a token that we can find in a file or an env variable inside a Pod. This token is a ServiceAccountToken [12] which is tied with a specific ServiceAccount.

A ServiceAccount in K8s is one way of providing identity for Pods. Multiple service accounts are tied with a namespace and have different sets of permissions. We can mount a ServiceAccountToken into Pods at specific locations using a ProjectedVolume [11] and use it by just reading the file that contains the token. Configuring a new PodDefault ensures to mount the volume to every new Pod.

```
apiVersion: kubeflow.org/v1alpha1
kind: PodDefault
metadata:
  name: access-features
spec:
  desc: Allow access to Kubeflow Features
  selector:
    matchLabels:
      access-ml-pipeline: "true"
  volumeMounts:
  - mountPath: /var/run/secrets/kubeflow/features
    name: volume-features-token
```

```
    readOnly: true
  volumes:
  - name: volume-features-token
    projected:
      sources:
      - serviceAccountToken:
        path: token
        expirationSeconds: 7200
        audience: features.kubeflow.org
  env:
  - name: KF_FEATURES_SA_TOKEN_PATH
    value: /var/run/secrets/kubeflow/features/token
```

The token file lives under /var/run/secrets/kubeflow/features/token and it is automatically refreshed by the system every 7200 seconds. This token is valid only for the features.kubeflow.org audience.

4.5.3 Authorization Support

To make authorization work we use K8s native RBAC mechanism. This mechanism relies on typical K8s API objects (ClusterRoles/Roles, RoleBindings, ServiceAccounts) and the SubjectAccessReview (SAR) mechanism. The general idea is to specify resources, actions that can be performed on these resources, as well as tie these actions to service accounts or users. Then, we use SAR to check whether a user is allowed to perform an action on a resource of a namespace.

Service accounts are tied to ClusterRoles or Roles [10] via RoleBindings which specify permissions of a ServiceAccount on a specific namespace. A ClusterRole or Role contains a set of rules that combine apiGroups, resources, resourceNames and verbs. As a result, a rule specifies which actions (verbs) can a ServiceAccount perform on specific resources.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: kubeflow-feature-edit
rules:
- apiGroups:
```

```
  - features.kubeflow.org
  resources:
  - data_sources
  - entities
  - feature_services
  - feature_views
  - infra_objects
  - on_demand_feature_views
  - projects
  - request_feature_views
  - saved_datasets
  verbs:
  - create
  - update
  - delete
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: kubeflow-feature-view
rules:
- apiGroups:
  - features.kubeflow.org
  resources:
  - data_sources
  - entities
  - feature_services
  - feature_views
  - infra_objects
  - on_demand_feature_views
  - projects
  - request_feature_views
  - saved_datasets
  verbs:
  - get
  - list
```

For example, service accounts that are tied to kubeflow-feature-edit role can create, update and delete all registry objects, whereas the ones that are tied to kubeflow-feature-view role can only get or list registry objects.

5 Conclusion

This final chapter, presents a brief summary of our work and highlights our main contributions. It assesses some principal elements of our design and implementation process and finally mentions possible extensions and further improvements that could be developed in the future.

5.1 Concluding Remarks

The initial goal of this thesis was to integrate a component that manages features, a feature store, in an ML platform, in Kubeflow. We managed to achieve this goal by turning an open-source feature store, Feast, with poor cloud support into a cloud-native client-server system. To do so we:

1. Created a REST API server that manages feature definitions and their metadata.
2. Developed a client that interacts with the API server in a Kubernetes environment.
3. Improved Feast by adding authentication and authorization support.

Now, data scientists are able to securely share features and collaborate. They can also use all the tools a feature store offers and radically improve ML workflows by decreasing time spent on developing new features and by making sure their models receive consistent high-quality features.

During the design and implementation phases we had to dynamically iterate and constantly improve and document our work. We spent a lot of time delivering demos internally that acted as a proof of concept and resulted in valuable feedback. However, the highlight of the entire process was delivering a live demo in the Feast community during the biweekly community call. This really showed us that we are heading in the right direction, it proved the interest of the community in our work and helped us gather useful feedback.

5.2 Future Work

Having worked on radically changing how Feast looks and works acts as the first step of creating a truly powerful open source feature store that is able to support different kinds of use cases and environments. This means that there is great room for improvement and a lot of capabilities to extend Feast. The following ideas are just the tip of the iceberg in terms of fully leveraging the new architecture:

- Extend the Feast SDK client to use all the capabilities of the new API server. Allow fetching specific feature definitions without needing to fetch the entire project first.

- Use Kubernetes mechanisms to enhance the reach of Feast. For example, use Kubernetes RBAC to enforce access control to underlying data sources (both on the Offline and the Online Store) or make sure to delete namespaced resources in case a namespace is destroyed.

- Use Rok or other dataset management systems to share entire datasets in an efficient way inside a Kubeflow environment.

- Allow data scientists to run automated feature pipelines using KFP by extending the API server into an actual pipeline scheduler.

- Build a data quality monitoring system around Feast or integrate an existing one. This is actually a direction that the Feast community is currently heading to.

We really hope that these ideas are enough to get people to think on further improvements and extensions. For us, the goal right now is to push and actually contribute our work to the upstream project and allow it to flourish along with the Feast community.

Bibliography

[1] The Docker Authors. Use containers to Build, Share and Run your applications. https://www.docker.com/resources/what-container/.

[2] The Feast Authors. Feast Protobuf Definitions. https://github.com/feast-dev/feast/tree/master/protos/feast/core.

[3] The Feast Authors. What is Feast? https://docs.feast.dev/#what-is-feast.

[4] The Kubeflow Authors. Backend API server Repository. https://github.com/kubeflow/pipelines/tree/master/backend/src/apiserver.

[5] The Kubeflow Authors. An introduction to the goals and main concepts of Kubeflow Pipelines. https://www.kubeflow.org/docs/components/pipelines/introduction/.

[6] The Kubeflow Authors. Model serving using Kserve. https://www.kubeflow.org/docs/external-add-ons/kserve/kserve/.

[7] The Kubeflow Authors. An overview of Kubeflow's architecture. https://www.kubeflow.org/docs/started/architecture/.

[8] The Kubernetes Authors. ConfigMaps. https://kubernetes.io/docs/concepts/configuration/configmap/.

[9] The Kubernetes Authors. Kubernetes Documentation. https://kubernetes.io/docs/home/.

[10] The Kubernetes Authors. Role and ClusterRole. https://kubernetes.io/docs/reference/access-authn-authz/rbac/#role-and-clusterrole.

[11] The Kubernetes Authors. Service Account Token Volume Projection. https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/#service-account-token-volume-projection.

[12] The Kubernetes Authors. Service Account Tokens. https://kubernetes.io/docs/reference/access-authn-authz/authentication/#service-account-tokens.

[13] Phillips Brandon. gRPC with REST and Open APIs. https://grpc.io/blog/coreos/.

[14] Sculley D., Holt Gary, Golovin Daniel, Davydov Eugene, Phillips Todd, Ebner Dietmar, Chaudhary Vinay, and Young Michael. Machine Learning: The High-Interest Credit Card of Technical Debt. *NIPS*, 2014.

[15] The Google Developers. Introduction to Vertex AI Feature Store. https://cloud.google.com/vertex-ai/docs/featurestore/overview.

[16] The Google Developers. MLOps: Continuous delivery and automation pipelines in machine learning. https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning.

[17] The Google Developers. Protocol buffers. https://developers.google.com/protocol-buffers/docs/overview.

[18] IBM Cloud Education. What are containers? https://www.ibm.com/cloud/learn/containers.

[19] Python Software Foundation. A synchronized queue class. https://docs.python.org/3/library/queue.html.

[20] The gRPC Authors. An introduction to gRPC and protocol buffers. https://grpc.io/docs/what-is-grpc/introduction/.

[21] Red Hat. SubjectAccessReview Documentation [authorization.k8s.io/v1]. https://docs.openshift.com/container-platform/3.11/rest_api/authorization_k8s_io/subjectaccessreview-authorization-k8s-io-v1.html.

[22] Red Hat. TokenReview Documentation [authentication.k8s.io/v1]. https://docs.openshift.com/container-platform/4.8/rest_api/authorization_apis/tokenreview-authentication-k8s-io-v1.html.

[23] Hermann Jeremy and Del Balso Mike. Meet Michelangelo: Uber's Machine Learning Platform. https://eng.uber.com/michelangelo-machine-learning-platform/.

[24] Zinkevich Martin. Training-Serving Skew. https://developers.google.com/machine-learning/guides/rules-of-ml#training-serving_skew.

[25] Del Balso Mike and Pienaar Willem. What Is a Feature Store? https://www.tecton.ai/blog/what-is-a-feature-store/.

[26] Nigel Poulton. *The Kubernetes Book*. LeanPub, 2020.

[27] Amazon Web Services. What is DevOps? https://aws.amazon.com/devops/what-is-devops/.

[28] Tecton. The Fastest Way to Build and Deploy Data Pipelines for Machine Learning. https://www.tecton.ai/product/.