Large Scale In-Database Machine Learning using Cloud Native Workflows

Héctor Bállega Fernández

Delft University of Technology

Large Scale In-Database Machine Learning Using Cloud Native Workflows

by

Héctor Bállega Fernández

In partial fulfillment of the requirements to obtain the degree of
Master of Science at the Delft University of Technology,
to be defended publicly on Monday August 30th, 2021 at 12:00 PM.

Student number:	5153891	
Project duration:	February 1, 2021 – Agust 30, 2021	
Thesis committee:	Prof. Dr. ir. D. H. J Epema	TU Delft
	Dr. Asterios Katsifodimos	TU Delft
	Dr. Marios Fragkoulis	TU Delft

An electronic version of this dissertation is available at
http://repository.tudelft.nl/.

Contents

Abstract

Preface

1 Introduction 1
- 1.1 Problem statement 2
- 1.2 Research Questions 2
- 1.3 Thesis Outline 2

2 Background 3
- 2.1 Multimedia Database Systems 3
  - 2.1.1 Classification of Multimedia Databases 4
  - 2.1.2 Storing Multimedia Data on Relational Databases 4
- 2.2 Machine Learning for Image Recognition 5
  - 2.2.1 Deep Learning 6
  - 2.2.2 Convolutional Neural Networks 7
  - 2.2.3 CNN Models for Image Recognition 9
- 2.3 In-database Machine Learning 10
  - 2.3.1 Machine Learning & Relational Data 10
  - 2.3.2 In-Database Machine Learning Systems 12
- 2.4 Cloud Native Workflows for Machine Learning 14
  - 2.4.1 Kubernetes 14
  - 2.4.2 Machine Learning Workflows 16
  - 2.4.3 Kubernetes Workflow Engines for Machine Learning 17

3 Design and Implementation 19
- 3.1 Introduction 19
- 3.2 Requirements 20
  - 3.2.1 Functional Requirements 20
  - 3.2.2 Non-Functional Requirements 20
- 3.3 System Architecture 21
  - 3.3.1 SQLFlow Server 22
  - 3.3.2 Argo Workflow Controller & Argo Server 24
  - 3.3.3 MySQL Database 25
  - 3.3.4 Prometheus & Grafana 26

3.4 Dataset Retrieval 27
3.5 Workflow Execution. 29
3.6 Model Implementation 31
3.7 Conclusion 32

4 Evaluation **33**
4.1 Introduction 33
4.2 Experimental Setup 33
4.3 Optimal Dataset Partitioning on Inference Tasks 34
4.3.1 Experiment Design 35
4.3.2 Execution time among multiple partitions 36
4.4 Cost Optimization on different Query Plans 38
4.4.1 Model Repository Construction 39
4.4.2 Query Plan Evaluation 41
4.4.3 Evaluation of the Model Repository on Different Distributions 43

5 Conclusion & Future Work **45**
5.1 Conclusion 45
5.2 Future Work 47

A Argo Workflow **49**

B Model Repository **51**

Bibliography **57**

Abstract

During the last decade, the proliferation of smartphones, social media and streaming services has provoked an explosion of multimedia data. This large amount of image and video sources combined with more powerful and inexpensive computational capabilities brought by the cloud computing paradigm has facilitated the rapid growth of new machine learning models capable of extracting information faster and more accurately. However, the complexity to develop machine learning models has also grown, involving multiple steps, from the acquisition and preparation of data to the training, evaluation and deployment of models. To alleviate this, the leading database providers have started to integrate the predictive capabilities of machine learning directly into their systems. This new approach is known as in-database machine learning, and it brings new interesting properties such as the exploitation of the inherent relational structure of data or the preservation of its privacy and integrity since the inference occurs directly where the data lives. In this work, we present a cloud-native approach to perform in-database machine learning. We have extended SQLFlow, a bridge between SQL engines and machine learning toolkits to support models trained to solve image recognition tasks over image datasets, which meta-information is persisted on a relational database. Furthermore, we have encapsulated the definition of machine learning models on cloud-native workflows that are able to exploit the GPU resources available in a Kubernetes environment. Our research evaluates the scalability of the proposed system regarding the total execution time and GPU utilization. Besides, we are interested in exploring the design of optimized machine learning query plans, where the goal is to choose the optimal among multiple models that cover a range of specific classes to predict from attending its accuracy and execution cost. For that purpose, we have implemented a model repository containing different model variations and evaluated different strategies to optimize the model selection. Our experiments show that optimizing the model selection will lead to more accurate and faster results, especially when a query covers a high number of classes and the number of models that are able to answer them is limited.

Preface

This work puts an end to my journey as part of the EIT Digital Master School, an unforgettable experience that has let me grow academically and personally. I feel fortunate for having the opportunity to live and study in two different countries and for overcoming all the difficulties encountered during this pandemic, which has meant a lesson that I will hardly forget.

Without any doubt, a master thesis cannot be carried out alone, therefore I want to express my gratitude for those who helped me along the way. First of all, I want to thank my supervisor Asterios Katsifodimos for his guidance and support during this time. I am glad for the opportunity you gave me to work on an interdisciplinary project that integrates my interest in cloud computing with machine learning, which is an area outside my academic program but which has allowed me to learn so much from it.

I am also thankful to Marios Fragkoulis, especially for all the discussions and ideas he brought, which helped me move forward when I felt stuck. I want to express special gratitude towards Ziyu Li, who was always available for me when I needed her, and our weekly meetings and feedback helped me not to get lost. Without your encouragement and guidance, I could not have finished this work, and I hope we can continue our friendship when I get back to Delft. I also want to thank Prof.dr.ir. D.H.J. Epema, for being part of my thesis committee.

Last but not least, I want to thank Bram Brink and my parents, Carmen and Neme, for their endless love and patience, especially when we could not be physically together. I never felt alone, and I always knew I could find refuge in you.

Héctor Bállega Fernández
Aranjuez, August 2021

List of Figures

2.1 Tasks to solve on image recognition problems. 5
2.2 Multilayer Perceptron (MLP) [1]. 6
2.3 Convolutional neural network architecture [2]. 7
2.4 Convolution operation over the input data [3]. 7
2.5 2x2 max pooling operation. 8
2.6 Agnostic approach for in-database machine learning [4]. 11
2.7 Structure aware in-database machine learning, adapted from [4]. . 12
2.8 A classification of in-database machine learning systems [5]. . . . 13
2.9 Kubernetes high-level architecture. 14
2.10 A typical machine learning workflow. 16

3.1 In-database machine learning system architecture for image detection tasks. 21
3.2 SQLFlow server steps representation. 22
3.3 From Couler python description to Argo Workflow specification. 24
3.4 Argo workflows user interface. 25
3.5 MySQL database and its persistent volume on Kubernetes. 25
3.6 Grafana dashboard to monitor Argo Workflows. 26
3.7 Class diagram of the database schema after processing the dataset. 28
3.8 Sequence diagram of the workflow execution. 30
3.9 Sequence of steps for the model implementation. 31

4.1 Parallelization of workflows by partitioning the dataset. 34
4.2 Run time execution (in seconds) among multiple partitions and batch sizes. 36
4.3 Total GPU Memory Usage (in MiB). 37
4.4 Tree representation of a machine learning based query plan. . . . 38
4.5 Distribution of classes in the COCO training dataset. 39
4.6 Distribution of constrained models regarding accuracy and cost. . 40
4.7 Comparison between different strategies to generate the query plan. 42
4.8 Model selection performance over objective against different sampling strategy combinations. 43
4.9 Run time of the optimized approaches against the combination of different sampling strategies. 44

List of Tables

2.1 Most relevant CNN models to perform image recognition tasks. . 9
2.2 A resume of the main characteristics available for each system. . . 18

3.1 Image datasets added into our in-database machine learning system. 27
3.2 Machine learning models added into our system. 31

4.1 Overall available resources in the Kubernetes cluster. 33
4.2 GPU Memory utilization for different batch sizes. 35
4.3 Run time execution (s) for multiple partitions and gpu batch sizes 36
4.4 Total GPU Memory Usage (MiB) 37
4.5 Machine learning models trained to answer a set of classes with a fixed cost. 38
4.6 Set of predicates to answer. 38
4.7 Sample predicates to evaluate. 41

B.1 Model variations of YOLO generated for the model repository. . . 55

1

Introduction

In the last decade, we have experienced an explosion of multimedia data. The rise of smartphones, social media and streaming services has facilitated the access to a enormous valuable amount of images and video sources. To this date, a billion of users have uploaded more than 50 billion photos to Instagram [6] and 500 hours of video content are uploaded to YouTube every minute worldwide [7]. The large availability of multimedia data in combination with the more powerful and inexpensive computational capabilities brought by the cloud computing paradigm [8] has made the field of machine learning experience a rapid growth. More concretely, in the area of deep learning, which brings new predictive applications on the domain of image recognition. With the increasing popularity of machine learning, the main database providers have put huge effort to integrate the predictive capabilities of machine learning models directly into relational databases. However, the development of machine learning models has become complex enough, involving multiple steps such as the acquisition and preparation of data, the training of the model, its evaluation and deployment. To facilitate this, the usage of cloud workflows orchestration engines has emerged as a promising solution to automate all the steps involved in this process.

In this work, we present a cloud-native approach to perform in-database machine learning. We have extended SQLFlow [9], a bridge between SQL engines and machine learning toolkits, to support deep learning models able to perform image recognition tasks over image datasets, which meta-information is persisted on a relational database. Our system translates extended SQL code into cloud-native workflows capable of running image classification tasks using the GPU resources available in the cluster. Ultimately, our research aims to bring machine learning functionalities closer to the database, which is where the data truly lives.

1.1 Problem statement

Relational data is the most widely data type used across machine learning practitioners. According to the Kaggle 2017 survey on the state of data science and machine learning [10], the majority of data science tasks involves working around relational data. Consequently, we can observe that in the last years, the main database providers have started to integrate machine learning capabilities on top of their solutions. Recently, Oracle has added support for the deployment and life cycle management of machine learning models via REST APIs hosted on its cloud database. Similarly, Microsoft SQL Server offers the possibility to run machine learning models as Python and R scripts stored as procedures.

Nevertheless, learning on data in relational databases has received little consideration from the deep learning community [11]. This is because deep learning methods normally expect their input as fixed-size vectors and not in a tabular form. Besides, the format of some input data can be presented as images or video files which is a challenge itself to store and maintain on relational database systems.

1.2 Research Questions

In this work, we aim to investigate how to transparently integrate in a cloud environment the predictive capabilities of deep learning models trained to solve image recognition tasks with data of images persisted on a relational database. Based on this challenge we have defined the following research questions:

RQ1: How cloud-native workflows can facilitate machine learning inference tasks over images which meta-information is persisted in a relational database?

RQ2: How can we perform inference tasks over images in a fast, scalable and SQL-driven way?

RQ3: How can we model and process machine learning queries taking into account the trade-offs between execution and accuracy among multiple models?

1.3 Thesis Outline

Our report is structured as follows. Chapter 2 explains the background knowledge and concepts within the scope of our work. Chapter 3, presents the design and implementation of our in-database machine learning system capable of running image classification tasks in a relational way. Chapter 4 focuses on the evaluation and experiments performed to determine the optimal distribution of inference tasks and the design and optimization of machine learning query plans navigating different trade-offs. Finally, Chapter 5, summarize the conclusion of our research and point out some directions for future work.

2

Background

This chapter introduces the relevant background knowledge for the purpose of our work. First, in Section 2.1, we briefly introduce the field of multimedia databases and the challenges and issues encountered when storing multimedia data on a relational database. Next Section 2.2, presents the state of the art of machine learning to solve image recognition tasks. Section 2.3 introduces the topic of in-database machine learning and offers a classification based on a literature review. Finally, Section 2.4 explains the notion of cloud-native workflows and its applications on orchestrating machine learning tasks.

2.1 Multimedia Database Systems

Since the late 80s [12], there have been many efforts from the research community to explore the integration of multimedia information with relational database systems [13]. Relational databases are designed to mainly manage textual and numerical data. However, due to the heterogeneous nature of multimedia data this integration its not as straight forward as it initially seems. A relational database system that supports multimedia data needs to extends its capabilities to consistently store, transport, retrieve and display the information regarding its nature.

Taking into account the differences mentioned above, we can define a Multimedia Database as a collection of interrelated multimedia data items such as text, images, graphic objects, video or audio. Consequently, we can define a Multimedia Database Management System (MDBMS) as the framework who provides support for the creation, storage, access, query, and control of multimedia data [14]. There a multiple criteria on the classification of Multimedia Database Systems, but for the purpose of this work will attend regarding the level of integration of the media file within the database.

2

2.1.1 Classification of Multimedia Databases

We can classify multimedia database systems regarding the level of integration of the media file within the database:

- ***No integration***. This scenario does not store multimedia data on a database. Instead, it maintains the raw data on a separate file system and only retains the inherent meta-information on the database.
- ***Semi-integrated***. In this approach we keep the meta-information of the multimedia data on a relational representation, but also the raw data is stored in the database using a BLOB (Binary Large OBject) data type.
- ***Fully Integrated***. This type of multimedia database covers not only the storage of multimedia data but it also adds effective techniques for indexing and retrieval the information by type the of content or specific domain [15].

2.1.2 Storing Multimedia Data on Relational Databases

We can find in literature examples like Oracle Multimedia [16] that follows a fully integrated approach when storing multimedia data on a relational database. The authors point out some limitations such as the number of files a file system can handle, recovery and backup tasks or security issues coordinating the file system with the database. However, a relational database is not usually considered an ideal repository to store any kind of multimedia information. Historically, the general advise has been not to store multimedia data directly on a database due to performance issues when retrieving high volume of data.

According to Sears et al. [17], the main determinant factor between storing or not in a database is the storage fragmentation. Their study shows that BLOB files smaller than 256KB are efficiently handled by a database, meanwhile a file system is a better option for BLOB files greater than 1MB. If there is a need to handle a high volume of multimedia information, the most common approach is to completely decouple the multimedia data from a relational database and keep them distributed on a file server. Some real examples that illustrates this problematic are Haystack [18] and f4 [19], the storage systems developed by Facebook to handle billions of multimedia BLOB files.

2.2 Machine Learning for Image Recognition

Machine Learning is the study of computer algorithms that enable systems the ability to automatically learn and improve through experience [20]. Machine learning allows systems to make autonomous decisions by finding underlying patterns in data. We can classify machine learning in different sub-fields based on their learning approach:

- **Supervised learning** uses a given set of instances labeled with an expected output value to find a function that maps new input data with the desired result.
- **Unsupervised learning** uses only unlabelled input data to find the underlying structure to group the input data.
- **Reinforcement learning** is concerned with the training of machine learning models to make a sequence of decisions. It studies how an agent can learn to achieve goals in a complex and uncertain environment [21].

Machine learning has applications in many different fields such as recommendation systems, online fraud detection, automatic language translation or image recognition. For the purpose of our work we are going to discuss only the algorithms that are relevant to perform image recognition.

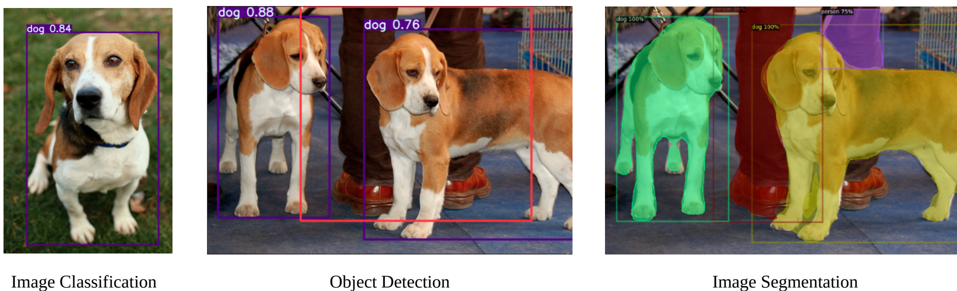


Image Classification Object Detection Image Segmentation

Figure 2.1: Tasks to solve on image recognition problems.

Image recognition is the ability of a system to identify and detect objects or features in a digital image or video [22]. It comprizes the following tasks (see Figure 2.1):

- **Image Classification.** It is the identification of a unique class to which an image belongs.
- **Object Detection.** It deals with detecting one or more instances of semantic objects of a certain class in an image.
- **Object Segmentation.** It consists on locating the elements of an image to its nearest pixel.

2.2.1 Deep Learning

From all the existing machine learning methods, deep learning is the one who has dramatically improved the state-of-the-art on image recognition. Deep learning uses multiple layers of artificial neural networks (ANN) to model a complex non-linear relationships among data [23].

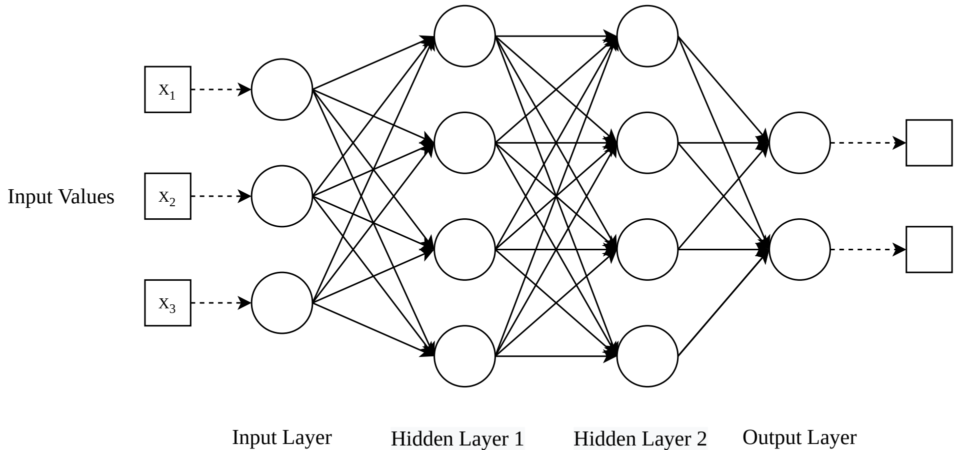


Figure 2.2: Multilayer Perceptron (MLP) [1].

Figure 2.2 illustrates a multilayer perceptron (MLP), one of the simplest deep learning models [1]. It is composed by three or more layers: an input and output layer and one or more hidden layers. The input layer does not perform any computation, it just maps the input values and forward them to the next layer. Between the input and output layers we have the hidden layers with an arbitrary number of neurons. It's here where the learning occurs by applying weights to the input and redirect them to an activation function as the output. The activation function introduces non-linearity into the network allowing the model learn more complex tasks. Finally, the output layer is responsible of producing the result regarding what we want to predict. Deep learning is outperforming traditional machine learning algorithms such as logistic regression or support vector machines. In the last decade, we have accumulated vast amounts of data and more inexpensive computational power, enabling training models more efficiently and accurately. The creation of deep learning models is an iterative process where we usually come up with a possible neural network architecture; we implement it and validate it by running an experiment that tells us how well the model performed. If the results are not good enough, we change the architecture and validate it again and again. Therefore, having a large amount of computational power drives us to find a better solution in a much faster time [24].

2.2.2 Convolutional Neural Networks

In this section we are going to introduce the concept of Convolutional Neural Networks (CNN), a deep learning architecture that has become the state-of-the-art to perform image recognition tasks. They are specially good at finding patterns on images, such as lines, circles or gradients and because they are designed to process input values as structured arrays they can work directly over raw images, without having to preprocess them [25].

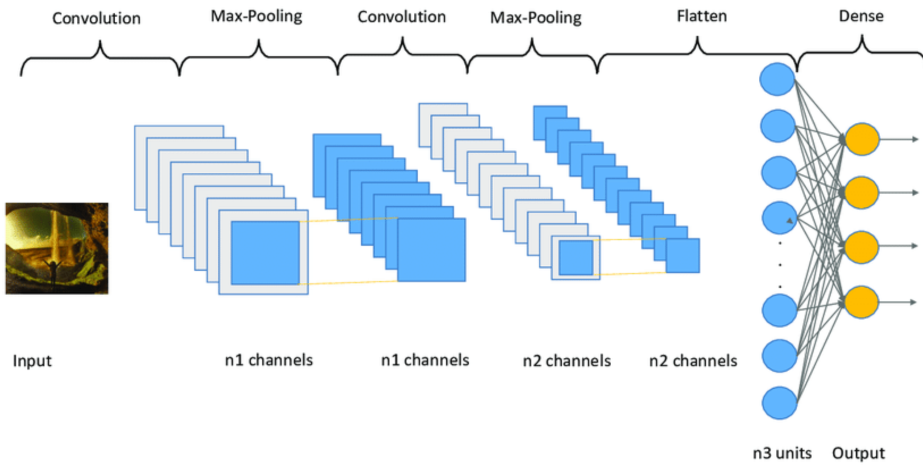


Figure 2.3: Convolutional neural network architecture [2].

The architecture of a Convolutional Neural Network is described in Figure 2.3. It can be described as a multi-layered feed forward neural network made by sequentially stacking several layers on top of each other. In the hidden layers we find convolutional layers, pooling layer, and two or more fully connected layers.

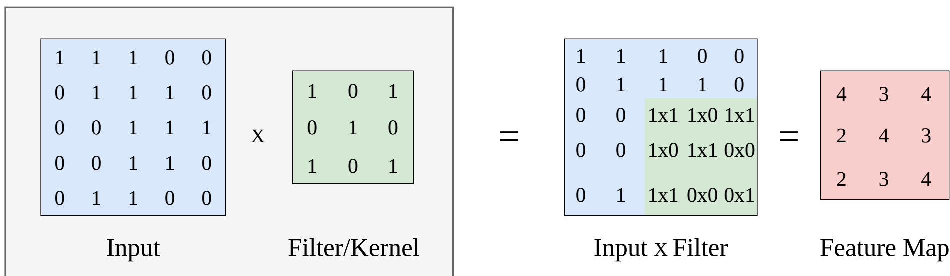


Figure 2.4: Convolution operation over the input data [3].

Convolutional Layer

The Convolutional Layer is the core building block of a Convolutional Neural Network. Convolution is a mathematical operation on two functions that express how the shape of one is modified by other. For the purpose of neural networks, the convolution (See Figure 2.4) is applied on the input data using a kernel or filter to produce a future map. We can apply one or more filters, depending on what we are trying to detect. After applying all the filters we will have a set of activation maps that we will stack along the depth dimension to form the final array. Finally, we will immediately apply an activation function such as rectified linear unit (ReLU) to bring non-linearity properties to the output.

Pooling Layer

After the convolutional layer, a CNN performs pooling to reduce the dimension of the feature maps. By doing this, we are able to reduce the number of parameters, shortening the total training time and preventing overfitting. The most common type of pooling is *max pooling*, which takes the max value in a pooling window (See Figure 2.5), but we can use other types of pooling layers such as *average pooling* to compute the average of the elements in the region of the feature map.

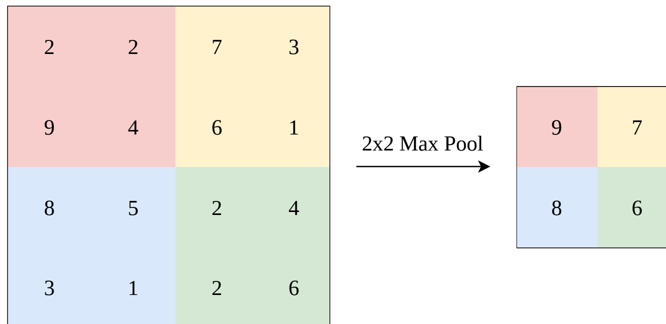


Figure 2.5: 2x2 max pooling operation.

Fully-Connected Layer

Finally, at the end of a CNN and before computing the final output probabilities, its common to add one or more layers fully connected to the previous one in order to learn possible non-linear combinations between the learned features and the sample classes. The fully connected layers acts as a classifier for the features detected on the previous layers [26].

2.2.3 CNN Models for Image Recognition

This section offers an overview of some popular CNN models to solve image recognition tasks. The first model we find in the literature is LeNet-5, proposed in 1998 by LeCun et al. [27], to perform handwritten character recognition. It was composed by several convolutional layers, pooling and three fully connected layers. LeNet-5 is the foundational CNN model, however it was not able to outperform classical machine learning algorithms such as support vector machines.

A decade later, Alex et al. [28] proposed AlexNet, a model that applied the basic principles of CNN towards using ReLu as the activation function to solve the gradient vanishing problem and GPU capabilities to accelerate the training. It won the ImageNet 2012 competition. Two years later, in 2014 the Visual Geometry Group (VGG) proposed a series of CNN model that won the ImageNet 2014 challenge. VGG [29] removed the LRN layer and reduced the number of parameters by half towards using a smaller padding value on the kernels to compute the feature maps. Since then, many other models have emerged. In 2016 He et al. [30] proposed ResNet a CNN model which introduced the concept of residual blocks, which are skip-connection blocks that learn residual functions with reference to the layer inputs [30]. In contrast to conventional CNN models where there are as many connections as layers, Huang et al. proposed DenseNet [31], who introduced the concept of Dense Convolutional Network and connected each layer to every other layer in a feed-forward way. DenseNet achieved a higher performance requiring less memory and computational power to be trained. Based on this models, a series of CNN architectures emerged to perform object detection tasks. Frameworks such as YOLO [32], SSD [33], Faster-RCNN [34] or MobileNet [35] use region proposal methods to general potential candidates to extract features from using VGG, ResNet or DenseNet as their backbone models.

Model	Task	Dataset	Model Size	Layers	Year
AlexNet [28]	Image Classification	ImageNet	238 Mb	8	2012
VGG-16 [29]	Image Classification	ImageNet	540 Mb	16	2014
ResNet [30]	Image Classification	ImageNet	100 Mb	50	2015
DenseNet [31]	Image Classification	ImageNet	64.5 Mb	121	2017
YOLO [32]	Object Detection	COCO	247 Mb	106	2015
SSD [33]	Object Detection	COCO	131 Mb	19	2015
Faster-RCNN [34]	Object Detection	COCO	364 Mb	55	2015
MobileNet [35]	Object Detection	COCO	90 Mb	28	2017

Table 2.1: Most relevant CNN models to perform image recognition tasks.

2.3 In-database Machine Learning

As we introduced in the previous section, the recent progress in the field of machine learning has brought new analytical applications to explore. A novel area that recently is gaining more interest by the academia [36] [37] and the industry [38] [39] is the so-called in-database machine learning, which tries to integrate the predictive capabilities of machine learning algorithms with relational databases.

In many practical situations, machine learning models are trained or used to predict over multiple data sources that are persisted in a relational database. The input of a machine learning model is usually represented in the form of a multidimensional array or tensor, which means than the input data that comes from the different tables might need to be joined into a single relation [40] to process it immediately by running a machine learning algorithm on it.

The main advantages of in-database machine learning are:

1. Preservation of the privacy and integrity of data, since all the operations are perform close to the database engine.

2. Usage of algebraic properties to exploit the inherent relational structure of data and for the optimization of processing multiple queries.

3. Leave the implementation details of machine learning models on a more abstract level, reducing the gap between data scientist and database engineers to operate with them.

2.3.1 Machine Learning & Relational Data

The Kaggle 2017 survey *on the state of data science and machine learning* [10], claims that among of 16,000 machine learning practitioners, 65.5 % of the overall data used to solve predictive tasks is relational data. Relational data inherently benefits from the human effort pursued to enrich and normalize its underlying domain knowledge [4].

We have identified from the academic literature [37] [4] [41] two well defined approaches to solve machine learning problems from the perspective of a database system [42]. The first one is an agnostic approach that treats the machine learning tasks as a black-box and only collects the materialized output from the database to process it. The other one is aware of the structure of the relational data inside the database and exploits it to achieve a better performance on the execution.

Agnostic In-database Machine Learning

The agnostic approach (Fig 2.6) to apply machine learning algorithms over relational data consists on first constructing a training and validation datasets and persist them on a database in the form of tables. Once we have the data accordingly normalized, we can run a feature extraction query to materialize the subset of examples to train or infer from and pass it to a machine learning toolkit to collect results [43].

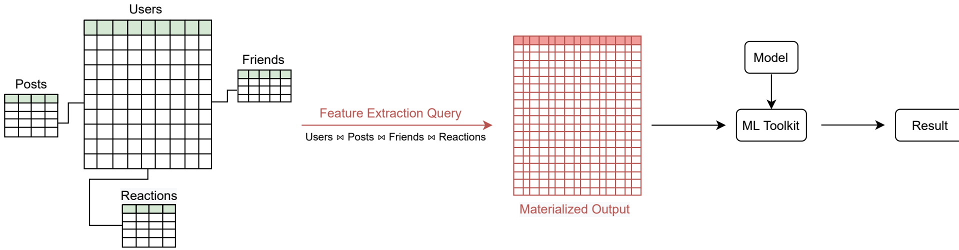


Figure 2.6: Agnostic approach for in-database machine learning [4].

The advantage of the agnostic approach is that combines two well-independent systems, the database engine to retrieve and store the data and the machine learning toolkit to process it. Regarding this, the agnostic approach can ideally work for any dataset or machine learning model. On the other hand, the main disadvantage is that we need to fully materialize the feature extraction query, instead of using the query processing capabilities of the database engine (e.g, compute the feature extraction using aggregate functions) to optimize the database workload [37].

Structure-aware In-database Machine Learning

The second approach (Fig 2.7) to apply machine learning algorithms over relational data is based on the observation that the feature extraction process involves a high degree of redundancy in the computation and representation of the materialized output to learn or infer from [44]. By exploiting the algebraic properties that underlies the relational model, the structure-aware approach aims to reduce the redundancy in the representation and computation of the query results [41].

Figure 2.7 illustrates the structure-aware approach. The system computes the model specification into a set of aggregate functions, one per feature or interaction. This process is known as model reformulation and it take advantage of the data dependencies to re-parameterize the model, so the model learns over a smaller set of determining features and exploits join dependencies to avoid redundancy in the representation of the query result. After, the model aggregates over the batch of queries and iterates over a gradient descent method until the it converges [4].

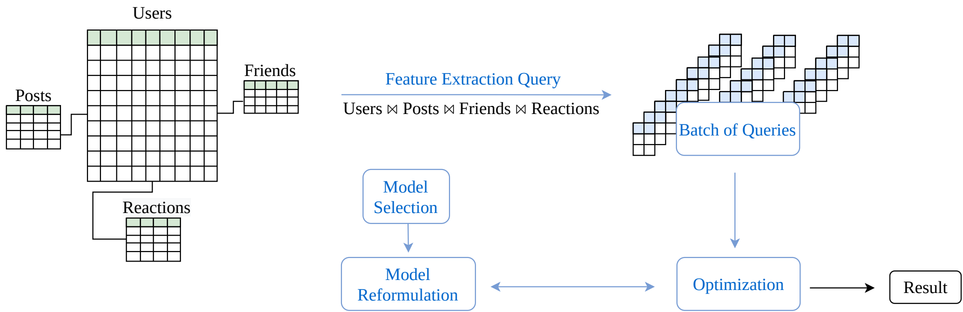


Figure 2.7: Structure aware in-database machine learning, adapted from [4].

The main advantage of an structure-aware approach is the avoidance of the full materialization of the feature extraction query. Examples like LMFAO [45] claims to only take 6.13 seconds to compute the batch of aggregate queries and 50 milliseconds for the model paremeters, in contrast with the agnostic approach on a PostgreSQL database that took 152 seconds to compute the join and 7249 seconds to calculate the model parameters of the fully materialized feature-extraction query [37].

2.3.2 In-Database Machine Learning Systems

In the literature [46] [5], we find that in-database machine learning systems are mainly classified into three categories (see figure 2.8) depending on the server-side development effort needed to integrate machine learning capabilities:

- ***Integrated systems***. In this approach (left part of figure 2.8), the query processing engine and the machine learning system are implemented on top of a common infrastructure. The integrated system process both SQL queries and execute machine learning algorithms within the same computational framework [5].

- ***UDAF-based systems***. Represented at the right part of figure 2.8. Its an in-database machine learning system where the machine learning algorithms are implemented as user-defined aggregate functions (UDAFs). The UDAFs are customized extensions that extend the query processing engine with machine learning capabilities.

- ***Pure SQL systems***. In this approach (right part of figure 2.8), the machine learning algorithms are implemented exclusively on SQL code and therefore the query processing engine remains unchanged.

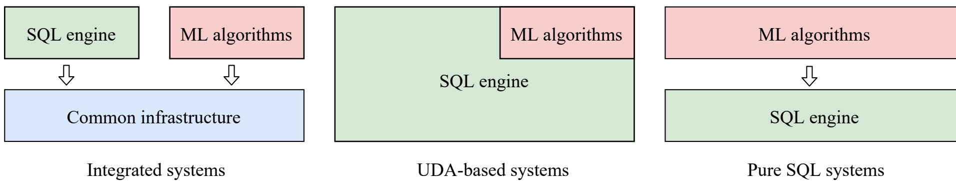


Figure 2.8: A classification of in-database machine learning systems [5].

Integrated systems offer the most granular level of union between machine learning algorithms and database engines. A relevant example found in the literature is Shark [47], the predecessor of the Apache Spark SQL project [48], which follows the principle of pushing computation to data and combines efficient SQL query processing using Resilient Distributed Datasets (RDD) with sophisticated distributed machine learning algorithms such as linear regression, logistic regression and k-means clustering. Another example of an integrated-system is the Teradata Machine Learning Engine [38], which uses a SQL-MapReduce approach to enhanced its SQL Engine with more than 100 pre-built analytical functions to solve different machine learning tasks [49]. The main disadvantage of integrated-system is the highly coupled implementation of the machine learning algorithms with the database engine, which makes it hard to extend. Another disadvantage is the high cost that supposes migrating from an existing database engine to a new one.

The pure-SQL approach is gaining more popularity, since the main cloud database providers have started to add machine learning capabilities on top of their database solutions. The most relevant one is Google BigQuery ML, presented in July 2018 [50]. The authors explain in [51] that re-implementing Big Query to fully integrate machine learning algorithms was not feasible and instead they decided to implement some of the most common machine learning algorithms such as linear regression, logistic regression or support vector machine with pure SQL code [46]. The main advantage is the elimination of UDAFs and that experimentally, the pure-SQL approach scales better without running into single machine bottlenecks compared to the UDAF-based approach [5]. On the contrary, the development of the BigQuery ML project proves that it is not feasible to implement sophisticated machine learning algorithms only in SQL code and its latest versions offer the possibility to import custom Tensorflow models [52] in a similar way as defined on the UDAF-based approach.

2

2.4 Cloud Native Workflows for Machine Learning

One of our goals is to understand the role of cloud native workflows to orchestrate machine learning tasks. In this section, we explain Kubernetes, the main framework in the open source community to develop cloud native applications. We define what a cloud workflow is and we give an overview of the main workflow engines to orchestrate machine learning tasks.

2.4.1 Kubernetes

Kubernetes is an open source container orchestration engine to automatize the deployment, scaling, and management of containerized applications [53]. It has its origins in Borg, a unified container management system develop internally at Google to handle long-running services and batch jobs [54]. Kubernetes follows a master-slave architecture (see Figure 2.9), where the master node act as an entry point that controls workloads across multiple worker nodes.

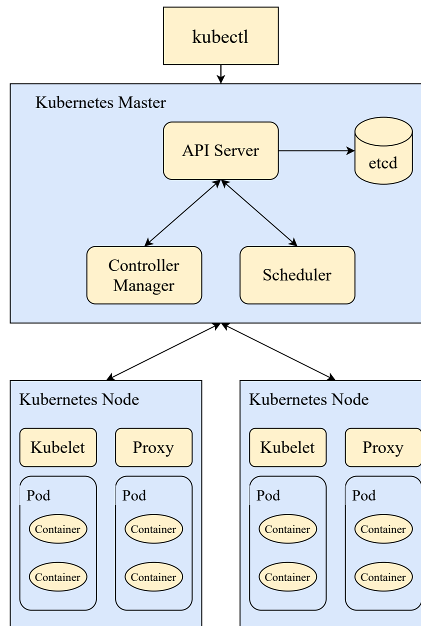


Figure 2.9: Kubernetes high-level architecture.

The main components of the master node are:

1. **Kubernetes API Server**. It exposes an API that let users manipulate the state of Kubernetes resources such as pods, namespaces, configmaps or events.

2. **Etcd**. A consistent and highly-available key value store to handle the state of the cluster.

3. **Schuduler**. It monitors unscheduled pods and their allocation on available nodes among the cluster. It also watches for the availability of resources in the node and anti-affinity specifications defined in the deployment.

4. **Controller Manager**. A daemon that contains a set of core control loops to regulate the desired state of objects in the cluster.

The worker nodes have the following basic components:

1. **Kubelet Agent**. It is an agent that runs on each worker node and communicates with the API server. It also ensures that every container runs as specified in its configuration file.

2. **Kube Proxy**. It maintains the network rules on each worker nodes and facilitates the communication of the pods inside and outside the cluster.

3. **Container Runtime**. It is responsible for running the containers and downloading the images from their repository.

Kubernetes offers different abstractions to build and orchestrate workloads. A workload in Kubernetes is a running application, and it can be defined as single component or several of them. They run as a set of *Pods*, which are a group of one or more containers that share storage and network resources. Workload resources are used to create and manage one or more pods. There are multiple types of workload controllers, e.g, *Deployment* controller which is used to manage replicated applications, *StatefulSet* to deploy scalable pods with persistent storage or *DaemonSet* to ensure that a copy of a pod is running across multiple worker nodes.

Kubernetes pods are created and destroyed regarding the state of the cluster. If we want to expose our pods as a logical set outside our cluster we have to use the *Service* resource. The Service will have an specific name and a unique IP address that will not change meanwhile the service is up and running. The default type of Kubernetes service is the *ClusterIP*, which exposes a service which is only accessible within the cluster. If we want to expose it via a static port from the node where it resides we can use a *NodePort* service, or a *LoadBalancer* type if we want to expose it via a cloud provider load balancer. Ultimately, Kubernetes allow modern applications to easily scale when their requirements grow as well as increase the agility and efficiency of their software development teams.

2

2.4.2 Machine Learning Workflows

A workflow is a procedure that manages repetitive processes and tasks defined in a particular order. The concept of workflow has been used in a wide range of disciplines, e.g, to model processes in engineering and manufacturing areas as well as to define complex business management processes in organizations. For the purpose of our work, we are going to focus on its applications to orchestrate machine learning tasks.

A machine learning workflow consist of sequence of tasks that aim to improve the accuracy of a model. Figure 2.10, illustrates this process. Notice that in most the cases the training of a model is an iterative process, therefore we have to repeat some steps until we achieve a satisfactory result on the model performance.

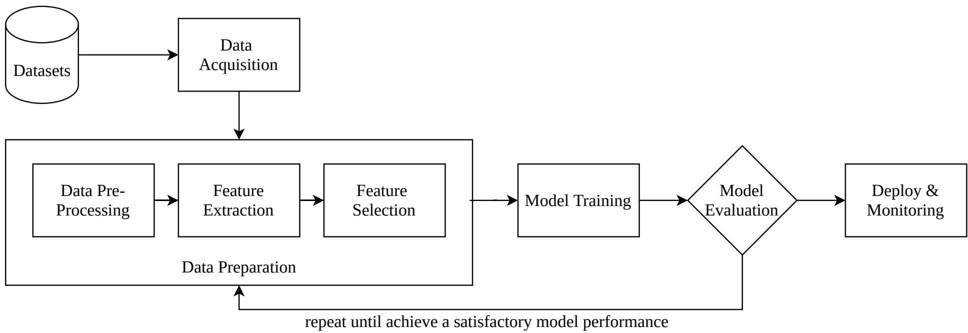


Figure 2.10: A typical machine learning workflow.

A typical machine learning workflow involves the following steps:

Data Acquisition

In this step we perform the extraction and collection of the data to be used to train our model. The data can be from a structured source, e.g a dataset, or from an unstructured source, for example the data collected from scraping a website.

Data Preparation

This step involves the pre-processing of the acquired data to represent it in an usable form, the extraction of specific features or attributes that can be utilized by the model and the selection of the most representative ones to train the model.

Model Training

The training of a machine learning model consist of optimising an specific cost function using the data obtained in the previous steps.

Model Evaluation

After the training a machine learning, we proceed to evaluate the performance of the model and iterate over the previous steps in case the obtained results are not good enough.

Deployment & Monitoring

The last step is an operational task aiming to track the status of the model in a production environment in order to discover unexpected errors. Besides, this step aims to prevent a phenomena known as model drift, which refers to a degradation on the model predictions due to unseen data or changes in the environment and other external factors.

2.4.3 Kubernetes Workflow Engines for Machine Learning

In this section, we are going to review the main orchestration engines for machine learning available in Kubernetes. The main characteristics that we are interested to review are related to the workflow definition, its execution and capability to recover and re-execute in case of failure. Table 2.2 includes a resume of the main characteristics available on each system.

Apache Airflow

Apache Airflow [55] is an open-source workflow management platform written in Python. The workflows are represented as DAGs (Directed Acyclic Graphs) and contains execution units call Tasks. Each DAG describes the execution order of the tasks and its dependencies. Airflow handles the triggering of workflows through an scheduler, and uses a local or remote executor to run them. By default, Airflow does not support dynamic workflows, since it will build the DAG before running it. However, developers can use airflow variables to store the state in which the pipeline will transit on run time. In a similar way, Airflow does not give a default fault tolerance strategy, since it implement different executors, however, by using the Kubernetes executor the tasks can be isolated on pods and easily restarted. As the workflows are written in Python, it can easily support embedding custom scripts on them. Finally, as Airflow is the most used workflow orchestration engine in the industry, it supports a seamless integration with most of the cloud and database providers.

Luigi

Luigi [56] is a Python based workflow engine to build complex pipelines of batch jobs. The pipelines are written in terms of targets, which corresponds to the state of a step of the workflow and tasks, where the computation is done. In contrast to Airflow, Luigi supports dynamic workflow since the tasks are instantiated dynamically. The tasks are targets defined as Python functions, and it supports workflow

parameterization as well as embedding scripts on the tasks. Because the scheduler runs locally toward the workflow, it does not support fault-tolerance by default. This lack of distributed execution makes Luigi much harder to scale compared to Airflow.

2

Argo Workflows

Argo Workflows is an open source container-native workflow engine for orchestrating parallel jobs on Kubernetes [57]. In contrast with Airflow or Luigi, the DAG in Argo is defined as yaml file with a set of steps and the possible dependencies between them. Each step in the workflow is a container which can capture inputs and emit outputs that are passed to other steps. Argo Workflows is implemented as a Custom Resource Definition on Kubernetes, and therefore each step executes as a pod. As it is developed with Kubernetes in mind, it provides an easy mechanism to re-run failed pods, but compared to other solutions it does not integrate seamless with other databases or storage systems, having to perform the connections or other operations in the container itself.

Kubeflow

Kubeflow [58] provides a toolkit to build machine learning workflows on Kubernetes. The workflows are defined using the Kubeflow Pipelines SDK, in terms of input parameters and a list of steps. Internally it uses Argo Workflows as its workflow engine, having the same benefits than Argo plus an specific interface written in Python focused on solving typical machine learning tasks, e.g, experiment tracking, hyperparameters tuning or model deployment.

	Airflow	Luigi	Argo	Kubeflow
Dynamic Workflows	No	Yes	Yes	Yes
Workflow Parameters	No	Yes	Yes	Yes
Fault Tolerance	Not by default	No	Yes	Yes
Embedded Scripts	Yes	Yes	Yes	Yes
Language Definition	Python	Python	Yaml	Python
Artifact Storage	PostgreSQL	PostgreSQL	Kubernetes	MySQL
Intregation With Other Systems	Yes	Yes	No	No

Table 2.2: A resume of the main characteristics available for each system.

3

Design and Implementation

In this chapter, we discuss the design and implementation of our in-database machine learning system capable of running image classification tasks in a relational way. First, we introduce the system requirements to proceed explaining in depth the architecture and its different components. Later, we discuss the dataset retrieval process, the model implementation and the execution of the cloud workflows.

3.1 Introduction

In the previous chapter, we introduced the field of in-database machine learning, a new approach to integrate machine learning algorithms over relational databases. Besides, we explained the concept of cloud native workflows and their applications to orchestrate machine learning tasks. Therefore, we acknowledge new possibilities in the research of in-database machine learning systems that leverage prediction tasks over multimedia data persisted in a relational database using cloud native workflows.

At the moment of writing this dissertation, most of the in-database machine learning systems only support basic machine learning algorithms such as linear regression, k-means clustering or simple neural network architectures. None of these are suitable to address image classification or object detection tasks, where the state of the art approaches make use of deep learning models based on convolutional neural networks (CNN). Recently, in-database machine learning systems like BigQuery ML offer the possibility to run custom models based on Tensorflow. However, it has some technical limitations such as the models are limited to 250MB, they must be stored in Google Cloud Storage and only support core TensorFlow operations. These restrictions prevent us from using some of the most popular architectures for image recognition tasks such as Mask R-CNN, SSD or yolov5.

3.2 Requirements

In line with the limitations mentioned in the previous section, we have identified the characteristics of an in-database machine learning system capable of running image classification tasks in a relational way. In this section, we transcribe these features in terms of functional and non-functional requirements.

3.2.1 Functional Requirements

(FR1) **Image storage and relational query processing.** We address the problem of running images recognition tasks over a set of images persisted in a relational way. Therefore, the system needs to handle the storage and retrieval of images from the perspective of a relational database problem.

(FR2) **In-database machine learning for image recognition.** Besides the storage and retrieval of images, our in-database machine learning system needs to leverage the raw information of the images with machine learning algorithms to perform inference tasks. It needs to implement a common interface to materialize a subset of images, run pre-trained image classifications models and collect the predictions on a result table. All this process needs to be done from a single entry point, such an extended SQL query language or in a programmatic way using an Domain Specific Language (DSL).

(FR3) **Parallelization of inference tasks.** The system needs to handle multiple images recognition tasks independently of the amount of users connected to the database.

(FR4) **Accelerate Inference Tasks on GPU.** A key property of an in-database machine learning system should be to exploit GPU resources to accelerate machine learning tasks. Moreover, for the purpose of image recognition, most of the deep learning implementations make use of the GPU parallelization model to maximize the amount of images inferred on a single batch.

3.2.2 Non-Functional Requirements

(NFR1) **Extensibility for other machine learning models.** To benefit from the multiple existing models to solve image classification tasks, the system needs to offer a structured and straightforward way to define its incorporation.

(NFR2) **Cloud platform independent.** The deployment of the in-database machine learning system needs to be independent of specific cloud platforms.

3.3 System Architecture

Our in-database machine learning system (Figure 3.1) is built upon different components to integrate multiple operations. From keeping the images and its meta-information on a relational way to executing inference tasks as Kubernetes workflows and monitoring their state. In this section, we provide an in-depth explanation of each component, its role on the overall implementation and the tasks it performs.

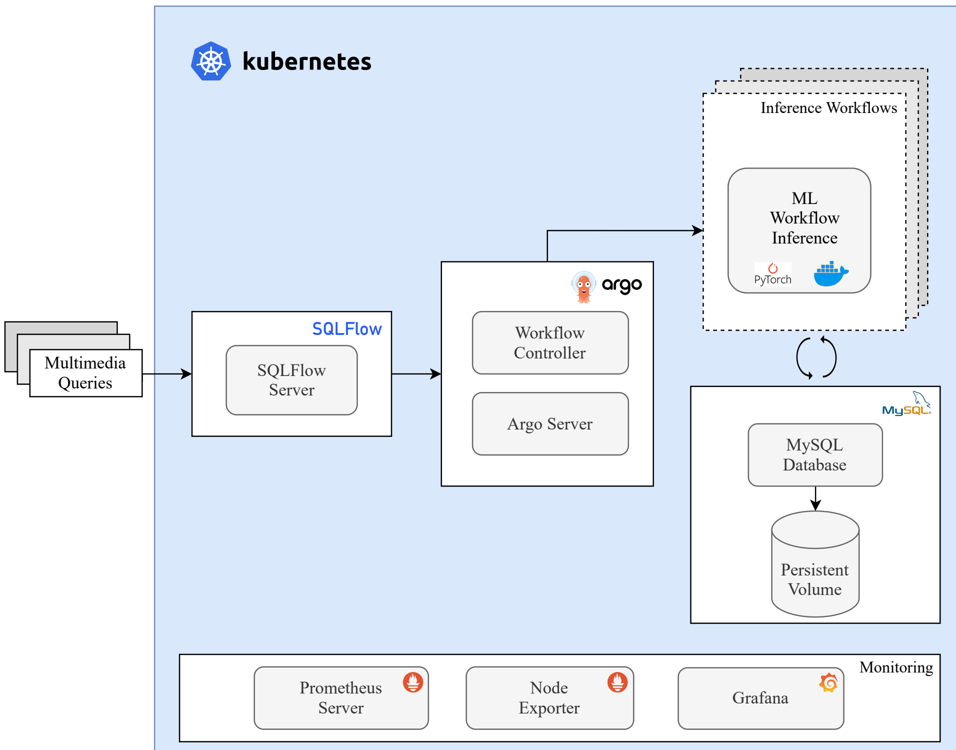


Figure 3.1: In-database machine learning system architecture for image detection tasks.

From a general perspective, the entrypoint of our system are the multimedia queries sent from a client to the SQLFlow server. The server translates the queries into Argo workflows that are executed over the cluster. In the workflows, we encapsulate the machine learning tasks to infer from the set of images given by the query. Finally, we keep the prediction results on a table in the database and monitor the state and performance of all the architectural components using Prometheus server to store metrics, Node Exporter to collect them and Grafana for its visualization.

3.3.1 SQLFlow Server

The SQLFlow server is the core component of our architecture and the framework we have based most of our work on. SQLFlow is a bridge that connects a SQL engine, e.g, MySQL or Apache Hive, with TensorFlow and other machine learning toolkits [9]. SQLFlow extends the classical SQL syntax to enable model training and inference. It keeps a loose integrating between the machine learning algorithms and the underlying SQL engine by translating a SQL query into a workflow program.

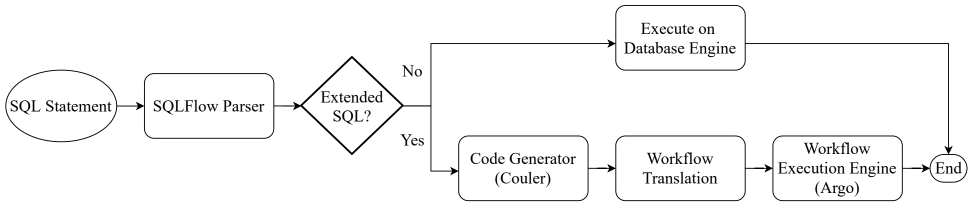


Figure 3.2: SQLFlow server steps representation.

The process of translating the SQL queries into workflow programs is represented in Figure 3.2. The SQLFlow server receives an SQL statement, parses it and check if it corresponds to an extended SQL statement defined by a train, predict, explain or run machine learning clause (see Example **??**). If the statement is a standard one, it use its corresponding engine parser noted as a third party parser and immediately send the statement to be executed on the database engine. If the statement is an extended SQL statement it combines the third party parser (so the syntax remains consistent) with an extended syntax parser to parse the machine learning clause [59]. After, the parsed SQL program will generate an intermediate representation that will be translated into a Couler program who provides a unified interface for constructing Argo workflows. Finally, the workflow specification will be submitted into the kubernetes cluster for its execution.

The example 3.1 illustrates the extended machine learning clauses available in SQLFlow. This approach of creating special SQL clauses to perform machine learning tasks is inspired by Google BigQuery ML. As we discussed on the previous chapter, BigQuery ML follows a pure SQL approach and it uses extended SQL clauses like CREATE MODEL to specify the characteristics of a machine learning model. In a similar way, SQLFlow defines the TRAIN clause to train a model using the result from a SELECT statement. The PREDICT clause makes a prediction using a previously trained model. The EXPLAIN clause display a visualization of the output of a machine learning model using the SHAP project (SHapley Additive exPlanations) and the RUN clause extends the SQL syntax to support complex end-to-end machine learning implementations which involves custom data trans-

formations.

```
SELECT ...  TO TRAIN  model_definition
            WITH parameters
            COLUMN features LABEL target
            INTO model_name;
SELECT ...  TO PREDICT field USING model_name;
SELECT ...  TO EXPLAIN model_name WITH parameters;
SELECT ...  TO RUN docker_image
            CMD parameters
            INTO result_table;
```

Example 3.1: Extended machine learning SQL clauses in SQLFlow [9].

3

In the previous chapter, we gave a classification for in-database machine learning systems based on the level of integration between the machine learning algorithms and the database engine (Section 2.3.2). Regarding SQLFlow, we can classify it as a hybrid since it borrows concepts from the pure SQL, UDAF-based and integrated approach. From the pure SQL, although in SQLFlow the machine learning algorithms are not implemented only in SQL, it uses an extended SQL language to define and trigger the life-cycle of the machine learning tasks. From the UDAF-based approach, the TO RUN clause encapsulates the model implementation on Docker images for complex end-to-end system. And from the integrated approach because the machine learning models and the database engine are loosely coupled but they both run on a common infrastructure in the Kubernetes cluster.

For the purpose of our work, we have modified the SQLFlow code generator and workflow translations steps to support machine learning models trained for image classification tasks. Due to the complexity of these models, the TO TRAIN and TO PREDICT clauses were not the most feasible approach since these clauses work as a high level abstraction of the Keras Modes API [60]. Instead, we prefer the flexibility of the TO RUN clause, which offer us a structure way of implementing pre-trained models to solve multiple image classification tasks and give us the flexibility to build a model repository based on the tagging strategy we establish when building and releasing the docker images.

As a summary and prelude for the next sections, we have used the SQLFlow extended syntax as a query language to collect the meta-information of our images. We have changed the code generator and workflow translations steps to fit models pre-trained to solve image classification tasks. Finally, we take the prediction results and store them on a result table defined in the original extended SQL query.

3.3.2 Argo Workflow Controller & Argo Server

The second component of our architecture is the Argo workflow controller that manages the life-cycle of the workflows and the Argo server to expose a user interface (see Figure 3.4) and REST API to interact with the controller. As we explained in the previous section, the SQLFlow server parses the extended SQL statements, generates an intermediate representation of the query and then fills a template to create a Couler description. After it will uses that description to generate an Argo workflow yaml specification (see Figure 3.3). The Argo workflows encapsulates the different steps to run the machine learning models over a subset of the image data from the database.

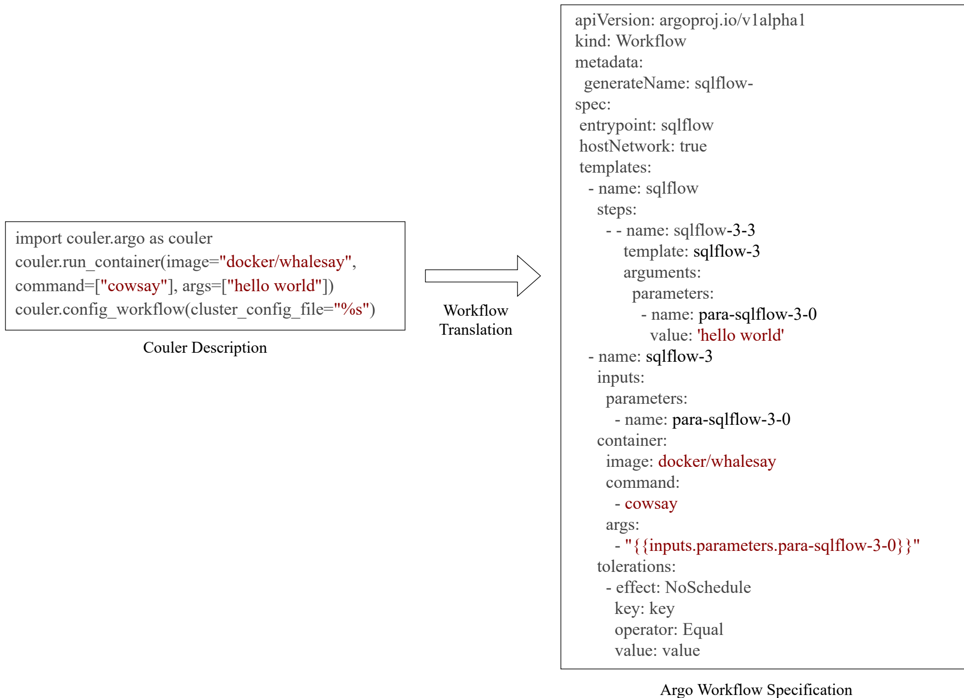


Figure 3.3: From Couler python description to Argo Workflow specification.

The right part of Figure 3.3 illustrates an Argo workflow specification. It is defined as a Kubernetes custom resource definition, and it consists of a template of steps with the instructions defined in terms of containers that receive inputs as arguments and optionally send outputs to other steps.

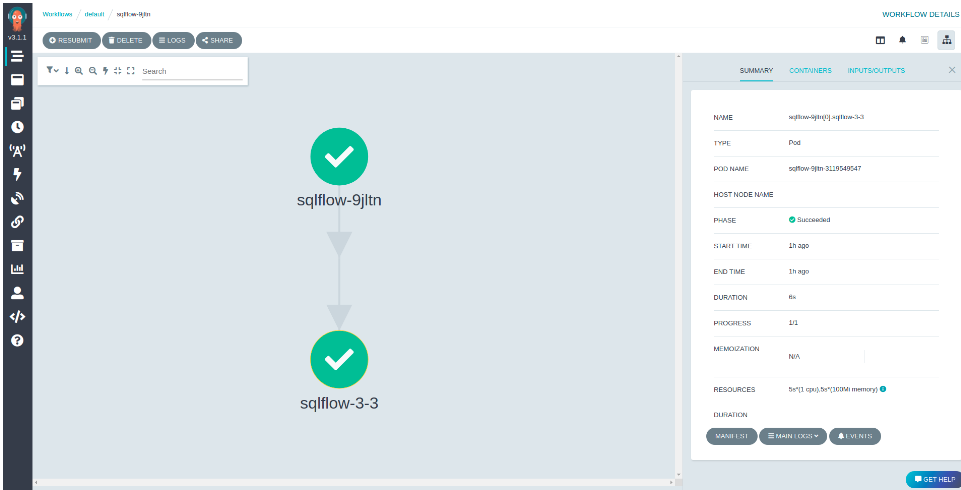


Figure 3.4: Argo workflows user interface.

3.3.3 MySQL Database

The third component of our architecture is the MySQL database (see Figure 3.5). The database serves the purpose of keeping the access and storage to the image datasets. For the deployment of our database we have followed an Infrastructure as code approach [61] where the entire system is described in a declarative way.

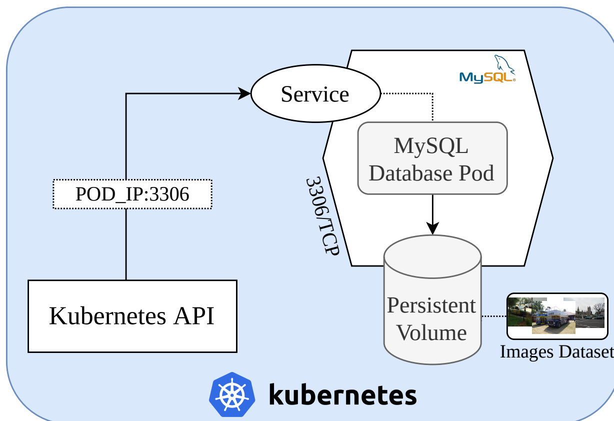


Figure 3.5: MySQL database and its persistent volume on Kubernetes.

In our case, all the image datasets are embedded as SQL scripts on the docker image, and we recreate them on an initialization script defined as the default entrypoint in the container. For this reason, we don't need to manually import the datasets after the initialization of the database and therefore all the data is available every time

the container restarts. We have decided not to integrate the raw images as BLOB files in the dataset, and instead we follow a more cloud native approach where the raw images are stored on a persistent volume resource on the Kubernetes cluster and in the database we only keep references to the paths where they are located. Due to this, the whole database scales much better since initializing it with BLOB files would be very time consuming and would require a lot of memory on the node where the database is allocated. In this way, the persistent volume can be defined as a cloud network resource, e.g, a S3 or Google Cloud Storage bucket or any distributed object storage server implemented as a Kubernetes operator such as MinIO [62].

3.3.4 Prometheus & Grafana

The last component of our architecture corresponds with a set of services that facilitates the monitoring and collection of metrics in our cluster. The Argo workflows emit by default a set of metrics related with the state of the controller, by installing Prometheus server, towards Prometheus node exporter we can easily scrap them from an specific endpoint defined in the Argo server. Figure 3.6, shows the Grafana dashboard to visualize the state of the controller and the running workflows.

Figure 3.6: Grafana dashboard to monitor Argo Workflows.

3.4 Dataset Retrieval

After introducing the different components of our architecture, we need to collect multiple image datasets and import them into our in-database machine learning system to manipulate them on a relational way. The datasets that we have collected are not only relevant for the purpose of image classification but also in the context of object-recognition, broadening the question of scene understanding to more complex contexts.

Dataset	Year	Instances	Classes	Size	Format
VOC [63]	2012	17112	20	500x500	PNG
COCO [64]	2017	123287	80	640x640	JPEG

Table 3.1: Image datasets added into our in-database machine learning system.

Regarding this, we have incorporated two of the most relevant datasets (see Table 3.1) in the literature that are based on images containing complex scenarios and that follow a multi-object classification approach. The first dataset that addresses these issues is the PASCAL VOC [63], which provides a common set of tools for accessing the dataset and its annotations and offers a standardized way to evaluate new methods through the PASCAL VOC Evaluation Server [65]. The second dataset and the one we have based most of our work on is the Common Objects in COntext (COCO) [64]. The COCO dataset represents an evolution from the Pascal VOC project, it contains 80 different classes and more than 120,000 instances, aiming to train more capable models in terms of what the model can predict and more accurate performance on their evaluation since it counts with much more samples.

After identifying the relevant datasets for our research, we have to transform them from its raw format into one compatible with our database system. The end goal is to have the data represented in a relational format so it can be manipulated using a SQL language. The procedure to import the datasets into the database system consists on the following steps:

1. **Download raw images and its annotation.** The first step consist on downloading the VOC and COCO raw images and its respective annotations. For each training, validation and test partition we will create a separate database. Afterwards, as we previously explained in section 3.3.3, we upload the raw image files into our file server.

2. **Convert annotations to COCO JSON format.** For each training, validation and test dataset we ensure that the annotations are in the COCO JSON

format, which is an standard notation used to facilitate the training of the models. As it is obvious the COCO dataset comes in this format, but not VOC. We made an script to convert annotations written in PASCAL VOC XML notation to COCO JSON.

3. **Transform COCO JSON objects to database tables.** After we have the annotation in COCO JSON, we proceed to incorporate the annotations for each train, validation and test dataset as a separate database in our MySQL instance. We made use of pandas and the sqlalchemy library for the creation of the tables. Figure 3.7 shows a class diagram with the database schema of the annotations.

4. **Clean and export train/validation databases as sql scripts.** In order to have the data available just right after deploying the database in the cluster, we clean and export all the train/validation databases as sql scripts, so the database service can be initialize with them for each deployment of the system.

5. **Add sql scripts to the database Dockerfile.** The last step consists on adding the sql scripts into our Dockerfile, build, push and re-deploy the database service to the kubernetes cluster.

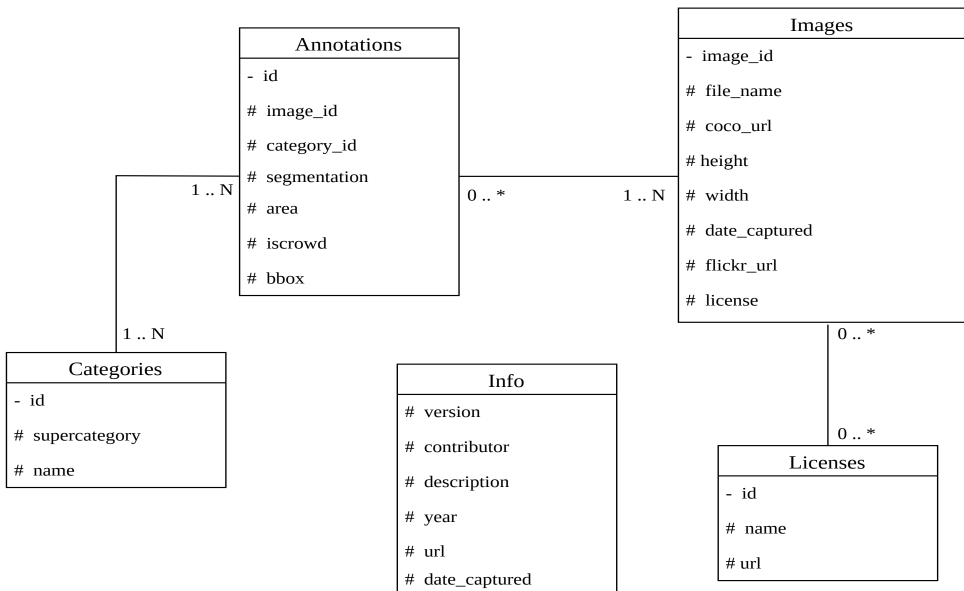


Figure 3.7: Class diagram of the database schema after processing the dataset.

3.5 Workflow Execution

Once the image datasets are transformed into tables we can manipulate them in a relational way and run extended SQL statements to perform inference tasks. In this section, we describe in more detail the whole process of translating the extended SQL statements into the Argo workflows and its execution in the Kubernetes cluster.

As we explained in Section 3.3.1, our work extends the TO RUN clause from SQLFlow to support the execution of machine learning models trained to solve image detection and object classification tasks. Therefore, the entrypoint of our system are extended SQL statements as shown in Example 3.2. We can differentiate three parts on them, the first one (lines 1-2) corresponds with the instance selection of the samples we are going to infer from. We can take advantage of the inherent relational structure of our data and apply common SQL clauses like ORDER BY or ASC on them. The second part (lines 3-8) corresponds with the model execution, here we specify a Docker image and an entrypoint script that will trigger the machine learning inference task. Finally, the last part (line 9) corresponds with a table in the database that will persists the prediction results.

```
1 SELECT * FROM coco_val.images
2 ORDER BY images.image_id ASC
3 TO RUN hebafer/yolov5-sqlflow:latest
4     CMD "run_yolov5.py",
5     "--dataset=coco_val",
6     "--image_dir=/datasets/coco/val/val2017",
7     "--repository=ultralytics/yolov5",
8     "--model=yolov5s"
9 INTO result_table;
```

Example 3.2: Extended SQL statement with TO RUN clause.

The process of translating the extended queries into workflows is illustrated on Figure 3.8. The client receives extended SQL statements with the TO RUN clause (see Example 3.2), and before reading them it checks that it can establish a connection with the database. If the database is up and running, the client connects to the SQLFlow server and submits an Argo workflow. The SQLFlow server parses the extended SQL statement, generates an Intermediate Representation and creates a Couler code description. As we explained in Section 3.3.2, Couler is a library written in python to specify Argo workflows. With the yaml specification of the workflow, the SQLFlow server calls the Kubernetes API to submit it into our cluster. The workflow runs based on the Docker image and the entrypoint from the extended SQL statement. The workflow connects to the database, retrieve the desired samples and call the machine learning models to retrieve predictions on them. Finally, it persist the prediction results on the result table defined in the extended SQL statement and returns the workflow result to the client.

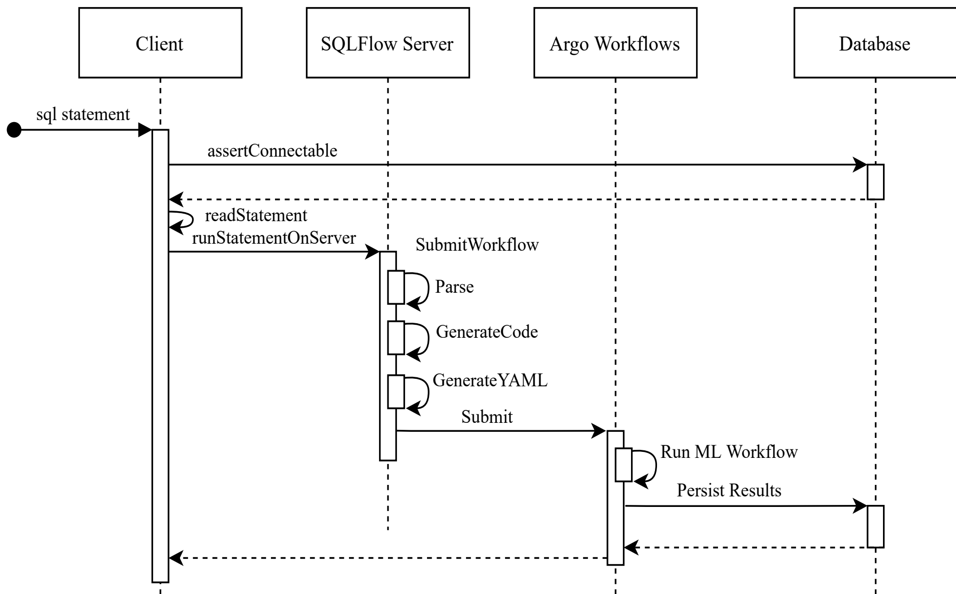


Figure 3.8: Sequence diagram of the workflow execution.

We have extended the workflow translation of the SQLFlow server to support image classification and object detection tasks. As we discussed in Section 3.3.3, we decided not to store our images as BLOB files to enhance the scalability of the database. Instead, our image datasets only contains a reference to the absolute path inside a Kubernetes Persistent Volume that maintains the image raw files. In order to deal with the raw image files we have to develop an strategy so the machine learning models within the workflow have access and can infer from them.

For this purpose, we have modified the Couler library which is in charge of generating the Argo workflows and the internal Couler template filled with Intermediate Representation of the extended SQL statetements to support attaching Persistent Volumes into the Argo workflows, so the base container defined in the extended SQL statement can load and manipulate in a convenient way the image files without having to manipulate them directly from the database. We have included the complete workflow yaml specification of the Example 3.2 in the Appendix A.

3.6 Model Implementation

In the previous section, we explained in detail the execution of the workflows that run image classification and object detection tasks. As we have seen, the machine learning models are encapsulated in Docker containers. This approach address a double goal: first, we want to abstract the implementation details of the machine learning models from users who might not have the required knowledge on this field but they know how to work around data. In our case, the containers act as a black box that run machine learning models given an input and an output defined by the extended SQL statements. The second goal is to facilitate the incorporation of new models into our system. We offer a public repository [66] containing our model implementations so other users can collaborate and add future machine learning models that outperforms the ones we currently have.

Model	Labels	Framework	Backbone
Faster-RCNN	COCO	Tensorflow	ResNet101
SSD	COCO/VOC	Tensorflow/Pytorch	ResNet50
yolov3	COCO	Pytorch	Darknet-53
yolov5	COCO	Pytorch	CSPDarknet

Table 3.2: Machine learning models added into our system.

Since our database has incorporated VOC and COCO as its main image datasets, we have collected machine learning models (see Table 3.2) available in the Tensorflow Model Garden [67] and Pytorch Hub [68] that have been pretrained on them. All the model implementations follow the same sequence of step described in Figure 3.9. Once the workflow is initialized, it connects to the database and runs the SELECT clause from the extended SQL statement to retrieve the instances we want to infer from. Later, it loads the set of images into the model and collects the prediction result. We manipulate the results on a Dataframe to keep a similar tabular structure as we have in the database. Finally, it connects to the database again and persists the results into the result table given in the extended SQL statement.

Figure 3.9: Sequence of steps for the model implementation.

3.7 Conclusion

In this chapter, we have presented an in-database machine learning system capable of dealing with the storage and retrieval of images in a relational way and performing image classification and object detection tasks. In section 3.2, we defined a set of requirements that our system had to fulfill. We are going to summarize our conclusions with regards to those requirements:

FR1. Image storage and relational query processing. In Section 3.3.3 and Section 3.4, we explained the approach we have followed to store and retrieve information from the image datasets. We have transformed the image datasets into tables to query them using SQL statements and in order to scale up our system we decide to store the raw image files in a kubernetes persistent volume that its attached to the workflows when they run machine learning tasks.

FR2. In-database machine learning for image classification. As we have seen in Section 3.5 and Section 3.6 the extended SQL statements are translated into workflows that runs on the Kubernetes cluster. The workflow execution is defined by a base docker image that runs the image classification task an persists the prediction results in a database table.

FR3. Parallelization of inference tasks. Each workflow execution (see Section 3.5) runs independently from each other, therefore the only limitation on the number of workflows we can run is based on the number a workflows the Kubernetes scheduler can allocate in a node.

FR4. Accelerate Inference Tasks on GPU. The Kubernetes nodes where our workflows runs have allocated GPU resources that the workflows can exploit to accelerate the machine learning inference tasks.

NFR1. Extensibility for other machine learning models. As we have seen in Section 3.6, the machine learning models are encapsulated in Docker images. Therefore, anyone can extend our approach and implement in the future other models that outperform the ones we currently have.

NFR2. Cloud platform independent. Since our system architecture (Section 3.3) and all its components are deployed in Kubernetes, our solution is completely independent from the main cloud providers such as Amazon Web Services, Google Cloud Platform or Microsoft Azure.

4

Evaluation

4.1 Introduction

In this section, we evaluate the performance of our in-database machine system in two different scenarios. First, we want to determine the optimal dataset partitioning on multiple inference tasks to classify the whole image dataset. Secondly, we want to evaluate the system performance under the execution of multiple cost-optimized query plans.

4.2 Experimental Setup

We have deployed our Kubernetes cluster on a virtual machine instance with GPU capabilities from the High Performance Computing (HPC) platform provided by SURFsara. The characteristic of the cluster are described in Table 4.1. In order to have GPU enabled nodes we have built a custom docker imaging packaging k3s (a lightweight kubernetes distribution) and the NVIDIA Container Runtime [69].

OS Image	Ubuntu 18.04.5 LTS
Kernel Version	4.15.0-151-generic
Kubernetes Version	1.21.2
Master Nodes	2
Worker Nodes	3
CPU	5
Memory	40Gb
Ephemeral Storage	100Gb
GPU	1x RTX 2080 ti (11Gb)

Table 4.1: Overall available resources in the Kubernetes cluster.

4.3 Optimal Dataset Partitioning on Inference Tasks

As we discussed in Chapter 2, the growing volume of data benefited the development of new machine learning techniques. However, the scale and complexity involved in building machine learning systems increased as well. One of the great challenges in the research of large-scale machine learning system is handling the massive amount of data that is involved [70].

When it comes to distribute large chunks of data on machine learning models across multiple nodes there are two well known strategies:

- **Data Parallelization.** In this approach, the same machine learning model is loaded among multiple workers, then the data is splitted into multiple chunks and forwarded to each worker node to be processed.
- **Model Parallelization.** In model parallelization the same dataset is replicated into the worker nodes which operate with different parts of the model. The final model is the result of aggregating all the model parts.

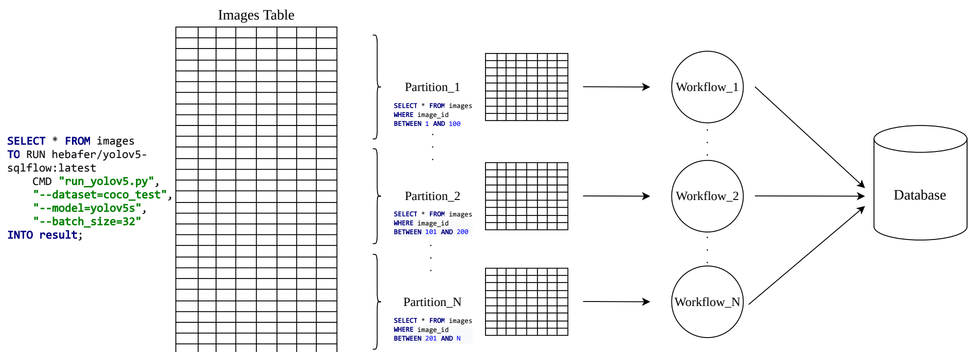


Figure 4.1: Parallelization of workflows by partitioning the dataset.

Our evaluation, described in Figure 4.1, will follow a Data Parallelization approach to optimize the execution time on classifying an entirely image dataset. The goal is to find the best balance between partitioning the dataset and the time it takes for each workflow to solve its inference task. Since our cluster has a limited amount of resources (in terms of CPU, memory and GPU), we cannot heavily increase the number of workflows because each of them require a fixed amount of GPU memory to load the model (around 500 MiB) and this value dynamically grows depending on the amount of images (per batch) that we load on the available GPU. If a workflow cannot allocate enough resources, it will fail and we will need to reschedule it on the cluster negatively impacting the overall performance.

4.3.1 Experiment Design

In this section we describe how we proceed with the experiment. We have used the COCO test dataset [64], previously described in Section 3.4 and we have divided the 40670 instances among 2, 4, 8 and 16 partitions. To overcome the GPU memory limitation, we have defined different fixed batch sizes (32, 64, 128 and 256) for each partition and analyzed how the amount of allocated GPU memory grows when more images are loaded into the GPU.

The Example 4.1 illustrates an extended SQL query that retrieves half of the COCO testing dataset (partitioning the dataset in 2) with a batch size of 32 samples. The model to infer from that we are going to use to evaluate our work on is YOLO.


```
1 SELECT * FROM images.coco_test
2 WHERE image_id BETWEEN 1 AND 290580
3 TO RUN hebafer/yolov5-sqlflow:latest
4 CMD "run_yolov5.py",
5     "--dataset=coco_test",
6     "--image_dir=/datasets/coco/test/test2017",
7     "--model=yolov5s",
8     "--write_mode=append",
9     "--batch_size=32"
10 INTO result;
```

Example 4.1: Extended SQL statement to retrieve half of COCO test dataset and predict from Yolov5 with a batch size of 32.

As we explained in Chapter 3 (Section 3.5), the extended SQL queries are translated into Kubernetes workflows that perform image classification tasks. The partitioning of the dataset is done by retrieving a subset of samples from the database (see SELECT clause of Example 4.1), and therefore the amount of partitions will determine the amount of workflows to execute. However, the amount of workflows that we can execute in parallel is limited by the available GPU memory at run time. Taking into account this limitation, we have proposed different batch sizes regarding the maximum GPU memory available.

Batch Size	32	64	128	256
GPU Memory Consumption (MiB)	2133	2512	4869	8519

Table 4.2: GPU Memory utilization for different batch sizes.

We have run single workflows with image batches of 32, 64, 128 and 256 to observe the amount of GPU memory consumed. The values are reflected in Table 4.2, we can observe that the amount of GPU memory scales linearly when the batch size does. Our system is capable of processing batches up to 256 images, if we increase it to a higher value, the workflow will fail since it will try to allocate more gpu memory than the one we have available (see Table 4.1).

4.3.2 Execution time among multiple partitions

The results from the experiment are described in Table 4.3 and Figure 4.2. We can clearly see that partitioning the whole dataset on multiple workflows and increasing the batch size improves the execution time. However, we can observe that the best execution time does not correspond with the highest partition value and batch size. Increasing the GPU batch size increments the amount of GPU memory allocated by a workflow but decreases the number of workflows we can run in parallel.

		GPU Batch Size			
		32	64	128	256
Partitions	2	781.50	677.50	656.0	693.0
	4	303.75	318.25	437.0	710.0
	8	308.50	***235.25***	411.0	798.0
	16	296.75	272.25	498.5	1094.0

Table 4.3: Run time execution (s) for multiple partitions and gpu batch sizes

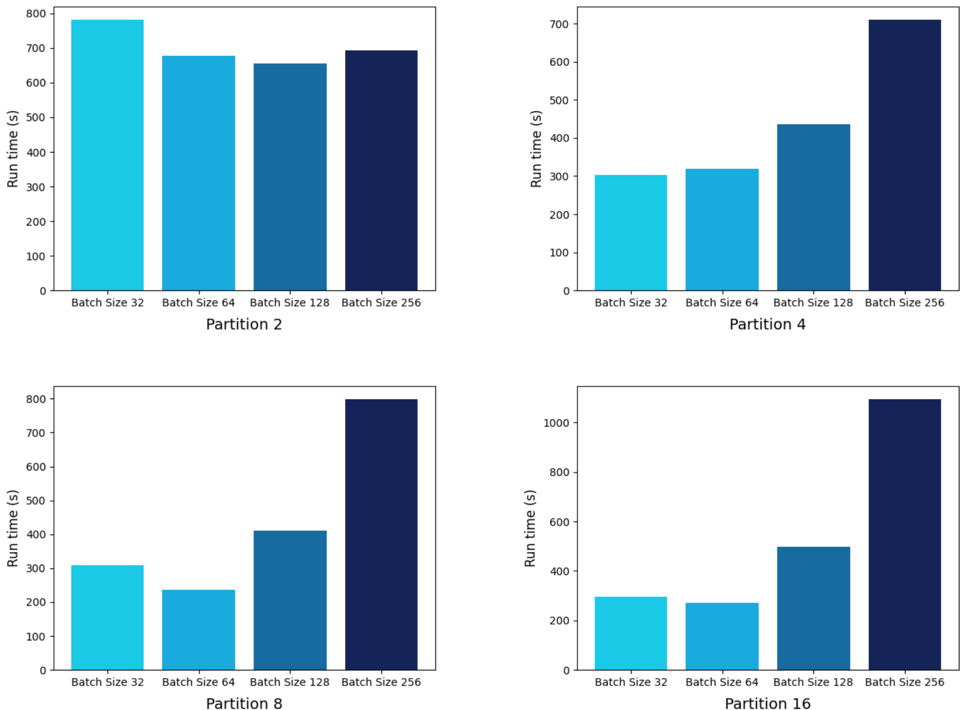


Figure 4.2: Run time execution (in seconds) among multiple partitions and batch sizes.

We have obtained the best execution time (lower amount of time to classify the 40670 samples from the COCO test dataset) for a partition of 8 workflows and a batch size of 64. As we explained before, the number of workflows we can run in parallel is limited by the amount of GPU memory available in the cluster. The batch size of 64 maintains a good ratio between the GPU memory used to load the model and the images to retrieve the predictions from and the number of workflows that can run at the same time to classify the whole dataset faster.

		GPU Batch Size			
		32	64	128	256
Partitions	2	4266	5024	9738	17038
	4	8532	10048	19476	34076
	8	17064	20096	38952	68152
	16	34128	40192	77904	136304

Table 4.4: Total GPU Memory Usage (MiB)

Figure 4.3 and Table 4.4 show the total GPU memory consumed by the workflows. We can see a direct increment when the batch size and number of partitions grow. However, if we compare this results with the total execution time from Table 4.1 we clearly see that due to the limited GPU memory in our cluster not by partitioning more and increasing the results we are going to lower down the overall execution time. Our approach aims to find the best ratio between the number of partitions and the batch size so we get the best out of our limited resources.

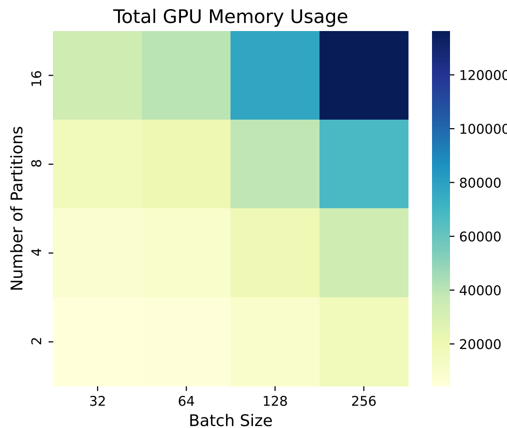


Figure 4.3: Total GPU Memory Usage (in MiB).

4.4 Cost Optimization on different Query Plans

In this section we are going to evaluate the capabilities of our system to run machine learning based query plans navigating the trade-offs between accuracy and cost (execution time) among multiple machine learning models. The end goal is to find an optimized query plan capable to answer predicates by choosing from a repository of models which are constrained regarding the cost that takes to run a model and the subset of classes it can answer.

For example, considering the following scenario, where we have different machine learning models trained to answer a set of predicates or classes with an assigned cost to execute that model (see Table 4.5).

Model	Classes	Cost
M1	car, truck, bicycle	100
M2	dog, cat, bird	100
M3	person, car	50
M4	person, bicycle, backpack	75

Table 4.5: Machine learning models trained to answer a set of classes with a fixed cost.

Predicates
person AND bicycle
(person AND dog) OR (person AND cat)
(car OR truck) AND person
(backpack AND bicycle) OR person

Table 4.6: Set of predicates to answer.

If we have a list of predicates (Table 4.6) with clauses made out of the classes the models can answer, the goal is to model an optimal machine learning based query plan (Figure 4.4) to find out which models we have to select in order to retrieve a set of images that fulfill the predicate condition minimizing the total execution cost.

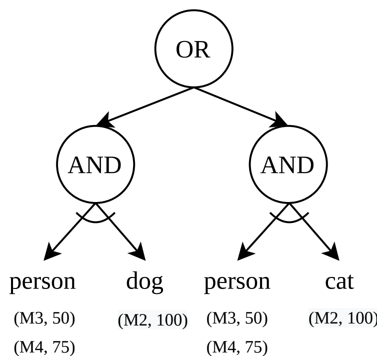


Figure 4.4: Tree representation of a machine learning based query plan.

4.4.1 Model Repository Construction

We construct a model repository by introducing modified submodels of YOLO trained on the COCO dataset. For our evaluation, we add variations of YOLO regarding the following properties:

- **Cost.** It refers to the execution time needed to classify a single image. The goal is to minimize it, since we want to reduce the total time needed to classify the set of images for a given predicate. To increase or decrease the cost, we introduce random latency values ranging from 1 to 50 seconds that will delay the total run time execution of the model.
- **Accuracy**. Its a value from 0 to 1 that denotes the probability of a model missclassifying an image. The higher the accuracy is, the more confidence we will have on the prediction result. The model with accuracy equal to one is our chosen base model (YOLO) which we are introducing the variations from.
- **Class Coverage**. It refers to a set of classes the model can identify. Since we have trained YOLO on the COCO dataset, our class coverage set will be a subset from the 80 classes available in YOLO. Figure 4.5 illustrates the 80 classes of COCO and its distribution among the training dataset.

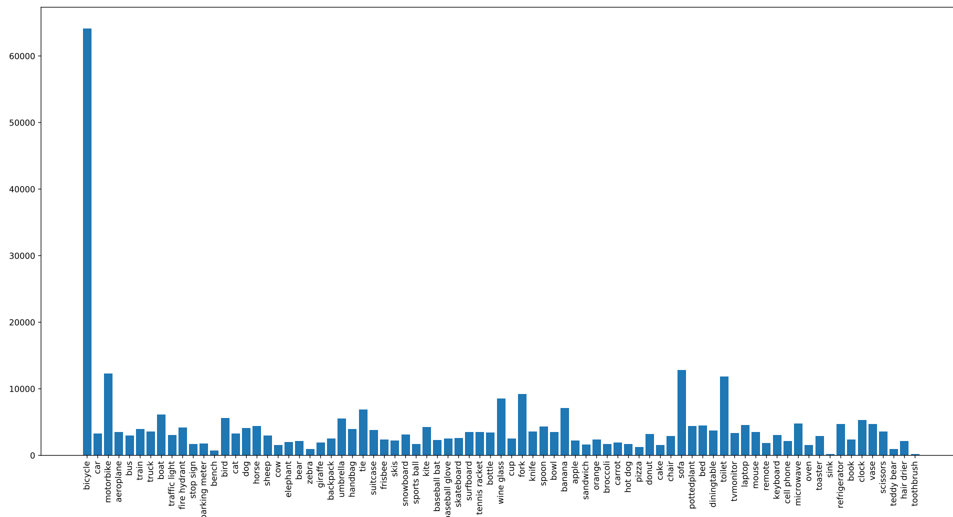


Figure 4.5: Distribution of classes in the COCO training dataset.

To implement the models regarding our constraints, we have extended the workflow execution process described in Chapter 3. We have modified the base Docker image to introduce the constraints for the latency, cost and class coverage.

```
1  SELECT * FROM coco_val.images
2  TO RUN hebafer/yolov5-sqlflow-variation:latest
3      CMD "run_yolov5_variation.py",
4      "--dataset=coco_val",
5      "--image_dir=/datasets/coco/val/val2017",
6      "--repository=ultralytics/yolov5",
7      "--model=yolov5s",
8      "--latency=50",
9      "--accuracy=0.75",
10     "--class_coverage=1,2,3,4,5"
11 INTO result_table;
```


Example 4.2: Extended SQL statement to run a Workflow with the model constraints.

The Example 4.2 shows an extended SQL statement that runs the modified model with new parameters. The details of the implementation and the base Docker image are available in our public repository [66]. For the latency, we inject a delay in seconds during the execution of the container, for the accuracy, we adjust the final value of the predicted class regarding the given probability. Finally, the class coverage parameter establish the number of classes the model is internally capable to predict. Based on this parameters, we have created different variations of YOLO, we randomly assign values for the latency (from 1 to 50), the accuracy or missprediction (from 0.8 to 1) and a random subset of classes from COCO. Figure 4.6 shows the distribution of the different variations of YOLO that we have randomly generated regarding the cost and the accuracy. The specific parameters for each model from the repository are available in the Appendix B.

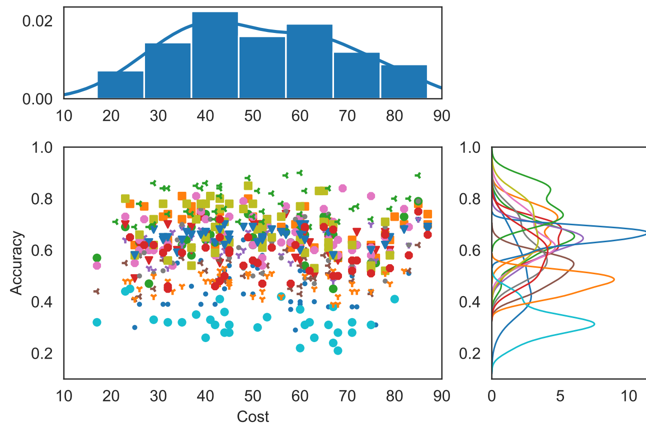


Figure 4.6: Distribution of constrained models regarding accuracy and cost.

4.4.2 Query Plan Evaluation

After building our model repository, we are going to evaluate different strategies to generate query plans that will select the optimal machine learning models that satisfy a predicate. As explained in the previous section, each machine learning model from the repository can answer a subset of the classes that form a predicate. The goal is to find the best combination of machine learning models that answer the predicate, considering that each of them will have a different cost and accuracy. To find those values, we run the 125 models from the repository on the COCO valuation dataset and collect its performance metrics to see assign respectively its cost (execution time) and accuracy. For the purpose of our experiment, we are going to evaluate four different predicates (see Table 4.7), we can observe that by increasing the number of classes on a predicate it reduces the total possible query plans we can combine.

4

Index	Predicate	Result
Q1	book OR person	2824
Q2	(cup AND person) AND (remote OR bird)	140
Q3	(dining table OR person) AND (clock OR handbag)	820
Q4	(dog AND person) OR (traffic light AND sport ball) AND ((boottle OR knife) AND (horse OR kite))	164

Table 4.7: Sample predicates to evaluate.

Once we set the cost and accuracy for each model from the model repository, we proceed to formulate the query plans in terms of a linear optimization problem. Having the predicate and model repository as inputs, the goal is to retrieve the models with the best accuracy minimizing the cost. To design the query plans, we apply three different strategies:

- **Baseline.** The baseline strategy follows a greedy approach where it only prioritizes choosing the models with the highest possible accuracy. It takes into account the restriction on the cost but it will not explore the combination of different models to minimize it.

- **Basic optimization.** The second strategy explores the combination of multiple models that satisfy the given predicate and minimize the total cost.

- **Selectivity and order optimization.** Our last strategy applies the basic optimization but also takes into account the selection and order execution of the chosen models. In this scenario, we aim to reduce the total cost by

forwarding to the next model selected in the query plan a the subset of images that have been already answered by the previous one.

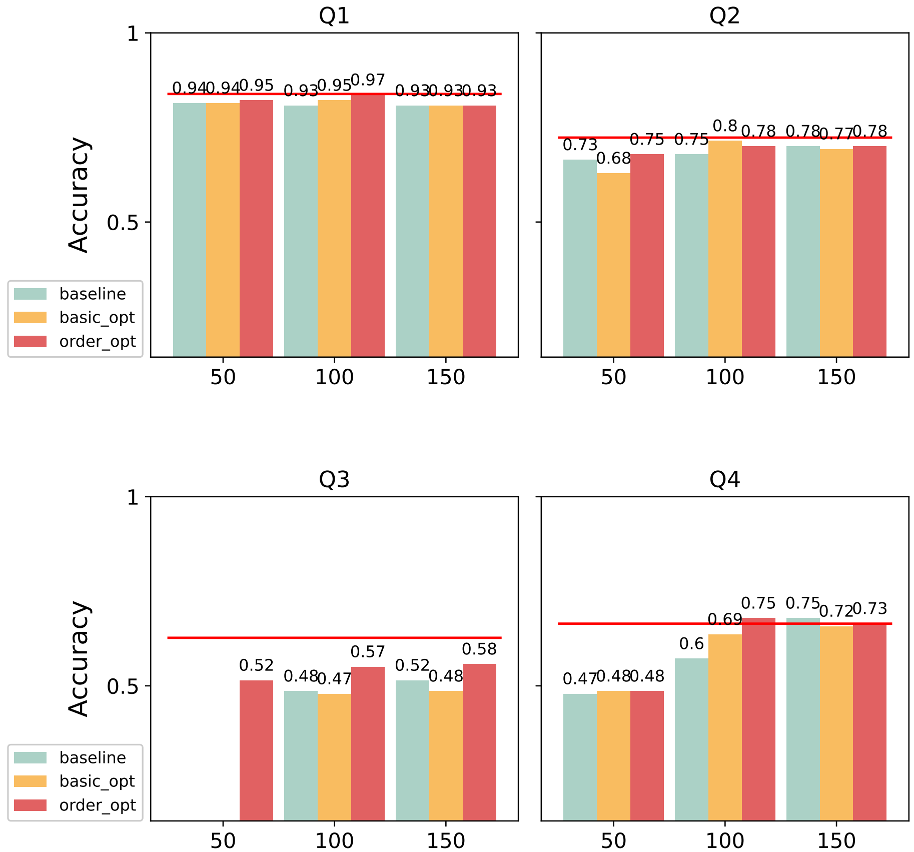


Figure 4.7: Comparison between different strategies to generate the query plan.

Figure 4.7 shows the results of evaluating the query plans generated for each predicate defined in Table 4.7. We can observe that the order and selectivity strategy always retrieves the most accurate models for each constrained cost (50, 100 and 150), outperforming the baseline and basic optimizations. For the case where we have a predicate both containing multiple classes (Q3 and Q4) and having a very strict cost (50), it makes more sense applying an optimization technique. Since otherwise, the baseline strategy will always be as good as the optimized one because there are no restrictions on the cost and both approaches will select the same models.

4.4.3 Evaluation of the Model Repository on Different Distributions

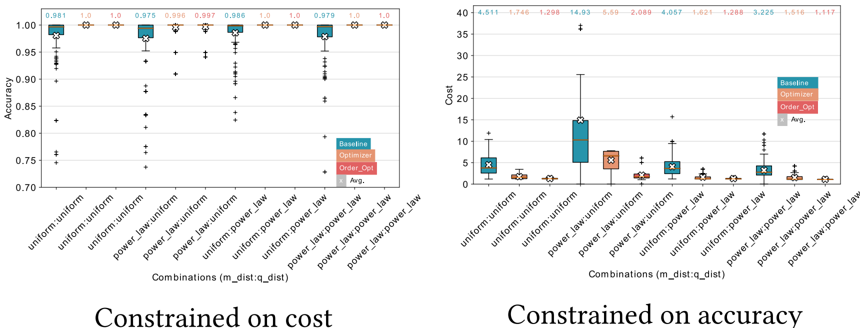
Finally, we are going to evaluate if the distribution of the model repository impacts the model selection when generating the machine learning query plan. We want to discover which models will be selected by the optimizer and what are their characteristics. We have synthesized the model repository and previous queries applying two sampling strategies:

- **Uniform sampling.** This is the strategy we used to generate the model repository in Section 4.4.1. We uniformly sampled the classes each model can predict and then we assign the accuracy based on the Gaussian distribution.
- **Power-law sampling.** Our second strategy samples the classes using the power-law distribution:

  $$y = ax^{a-1}$$

  Where a is equal to 5. The accuracy and cost are assigned as in the uniform sampling strategy.

When the classes are sampled uniformly, the majority of the classes are detected by most of the models. By using the power-law sampling strategy, we limit the classes a model can answer to, resulting into only a small number of models that are able to predict a wide range of classes. Figure 4.8 illustrates the model selection performance over our two constraints (accuracy and cost) using both sampling strategies. Each bar is a box plot where the end of the box plot are the 25th and 75th percentile. The blue color represents the baseline and the orange and red the optimized ones.

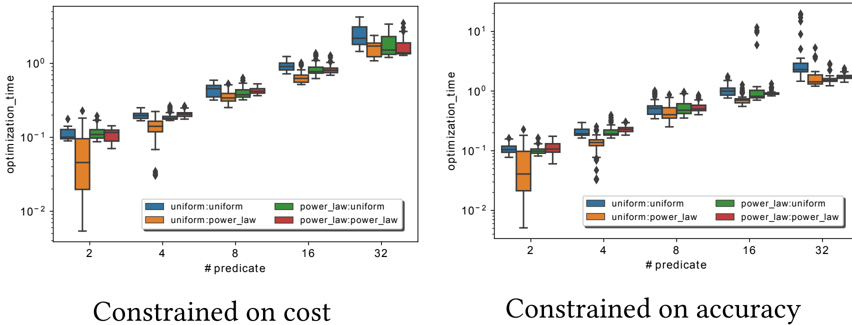


Constrained on cost

Constrained on accuracy

Figure 4.8: Model selection performance over objective against different sampling strategy combinations.

We see that when the cost is constrained, the optimized approaches are able to select models with a higher accuracy than the baseline. Similar results are shown when the accuracy is constrained, because the optimized approaches select models with a lower accumulated cost. For each combination of sampling strategies, while using the power-law sampling strategy to build the model repository, the baseline and optimized approaches show poorer results in terms of accuracy and cost. This is because the power-law sampling strategy limits the number of models that can answer a wide range of predicates, and therefore offers poorer results. -



Constrained on cost Constrained on accuracy

Figure 4.9: Run time of the optimized approaches against the combination of different sampling strategies.

Figure 4.9 illustrates the run time of query plans when the number of classes in a predicate scales. We can observe that the total execution time increases for a higher number of classes in a predicate. It is worth mentioning that a power-law sampling strategy leads to a skew distribution of the predicate coverage of models. If some predicates are answered only by a small range of model it will save time when optimizing the query plan, since there will be only a few number of models to choose that can satisfy it.

5

Conclusion & Future Work

This section summarizes our main findings by answering the research questions stated in Chapter 1 and discusses some possible future lines of work.

5.1 Conclusion

To conclude, we are going to revisit and answer our initial research questions:

RQ1. How cloud-native workflows can facilitate machine learning inference tasks over images which meta-information is persisted in a relational database?

In Chapter 2, we discuss the role of cloud-native workflows to orchestrate machine learning tasks. We can observe that machine learning workflows bring a standardized way of managing the life-cycle of a machine learning models, covering its definition, training, evaluation and deployment. To better understand its role in a cloud native environment, we give a classification of the main workflow engines with its particular characteristics available on Kubernetes. Later, in Chapter 3 we explain the role of Argo, our workflow controller and its impact in the overall architecture, giving our system the capability to systematically perform inference tasks over our image datasets. Our workflow definition encapsulates in Docker images all the necessary dependencies to perform its task, from the connection with the database, to the model execution and the retrieval of the inference results and its persistence in form of a database table. Besides, using a standardized approach for the workflow definition facilitates the extension of our system with future machine learning models.

RQ2. How can we perform inference tasks over images in a fast, scalable and SQL-driven way?

This research question is mainly answered in Chapter 3 and evaluated in our first experiment in Chapter 4. Our work extends the SQLFlow project to support deep learning models trained to solve image recognition tasks. First, we incorporate and represent our image datasets in a relational format, structuring them as a set of interrelated tables. To overcome scalability issues, we decided not to keep the images as BLOB files inside the database and instead store them on a Persistent Volume in the cluster. To perform the inference tasks, we use the extended SQL syntax defined in SQLFlow. As explained in Section 3.5, each inference task corresponds with an extended SQL query that is transformed into an Argo workflow. We have modified the workflow translation to attach persistent volumes containing the images files and to exploit the GPU capabilities of our cluster. The run time execution and scalability of the system are evaluated in Section 4.3. We designed an experiment to find out the optimal partition of the dataset on multiple inference tasks. Each partition runs an inference task over a subset of images with a specific GPU batch size. In the experiment, we observe that the run time execution and scalability of the inference tasks are constrained by the total amount of GPU memory available in the cluster.

RQ3. How can we model and process machine learning queries taking into account the trade-offs between execution and accuracy among multiple models?

In order to answer our last research question, we conduct an experiment in Section 4.4 that evaluates the capabilities of our system to run machine learning based query plans navigating the trade-offs between accuracy and cost among multiple machine learning models. We construct a model repository by introducing variations of a base model regarding the amount of classes it can predict and its execution time and accuracy (both measure on classifying the validation dataset). We extend the workflow execution process to introduce those constraints as new parameters in our base Docker image. After that, we apply three different strategies to generate the query plans that will select which models from the repository are capable of answering a given predicate. Finally, to evaluate the generation of the query plans we change the distribution of the models in the repository to discover which models will be selected over others.

5.2 Future Work

The present work has focused on the design and implementation of an in-database machine learning system capable of running image classification tasks on a relational way. During the implementation and evaluation of the system, new questions arose, which are not answered in favor of focusing on the purpose of our work. In this section, we explain some future ideas to explore:

Model training over relation data

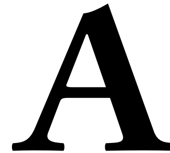
The present work discusses the characteristics of an in-database machine learning system capable of running image classification tasks on a relational way. We did not focus on exploiting the possibilities of directly training machine learning models from relational data. However, as we are encapsulating the machine learning inference tasks on containers, we could follow a similar approach and define workflows capable of training and pushing models to a public repository to use them to infer from.

Evaluation of our in-database machine learning system at the Edge

While writing this dissertation, we identified an increasing interest in the research community to evaluate deep learning models for image classification tasks on resource constraint devices [71] [72]. Besides of existing technologies such as KubeEdge [73], aiming to integrate Kubernetes with the Edge computing paradigm, our work is designed to run aware of a cluster environment, and it could serve as the foundation to evaluate the performance of an in-database machine learning system running at the Edge.

Optimize the model selection of query plans using chaos engineering

The last part of our work, evaluates the capability of an in-database machine learning system to execute query plans choosing among multiple models constrained by its execution time and accuracy to solve an inference task. While doing the evaluation, we found out that our constraints are closely related to the principles of chaos engineering [74], which is the discipline of experimenting on a system in order to build confidence in the capability to withstand turbulent conditions in production [75]. Platforms such as Chaos Mesh [76], offer the possibility to inject chaos into Kubernetes infrastructure in a manageable way. We could use Chaos Mesh to introduce latency and accuracy disturbance in the cluster instead of in an application level as it is currently done.



Argo Workflow

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
- name: sqlflow
  inputs: {}
  outputs: {}
  metadata: {}
  steps:
    - - name: sqlflow-1-1
        template: sqlflow-1
        arguments: {}
- name: sqlflow-1
  inputs: {}
  outputs: {}
  metadata: {}
  container:
    name: ''
    image: 'hebafer/yolov5-sqlflow:latest'
    command:
      - bash
      - '-c'
      - |-
        step -e "SELECT * FROM coco_val.images
        ORDER BY images.image_id ASC
        TO RUN hebafer/yolov5-sqlflow:latest
        CMD \"run_yolov5.py\",
            \"--dataset=coco_val\",
            \"--image_dir=/datasets/coco/val/val2017\",
            \"--repository=ultralytics/yolov5\",
            \"--model=yolov5s\"
        INTO result_table;"
```

```
32      env:
33        - name: SQLFLOW_DATASOURCE
34          value: 'mysql://root:root@tcp(DB_IP:3306)/?maxAllowedPacket=0'
35        - name: SQLFLOW_DB_PORT
36          value: 'tcp://SQLFLOW_IP:3306'
37        - name: SQLFLOW_DB_PORT_3306_TCP
38          value: 'tcp://SQLFLOW_IP:3306'
39        - name: SQLFLOW_DB_PORT_3306_TCP_ADDR
40          value: SQLFLOW_IP
41        - name: SQLFLOW_DB_PORT_3306_TCP_PORT
42          value: '3306'
43        - name: SQLFLOW_DB_PORT_3306_TCP_PROTO
44          value: tcp
45        - name: SQLFLOW_DB_SERVICE_HOST
46          value: SQLFLOW_IP
47        - name: SQLFLOW_DB_SERVICE_PORT
48          value: '3306'
49        - name: SQLFLOW_DB_SERVICE_PORT_DATABASE
50          value: '3306'
51        - name: SQLFLOW_PARSER_SERVER_PORT
52          value: '12300'
53        - name: SQLFLOW_SERVER_PORT
54          value: 'tcp://SQLFLOW_SERVER_IP:80'
55        - name: SQLFLOW_SERVER_PORT_80_TCP
56          value: 'tcp://SQLFLOW_SERVER_IP:80'
57        - name: SQLFLOW_SERVER_PORT_80_TCP_ADDR
58          value: SQLFLOW_SERVER_IP
59        - name: SQLFLOW_SERVER_PORT_80_TCP_PORT
60          value: '80'
61        - name: SQLFLOW_SERVER_PORT_80_TCP_PROTO
62          value: tcp
63        - name: SQLFLOW_SERVER_SERVICE_HOST
64          value: SQLFLOW_SERVER_IP
65        - name: SQLFLOW_SERVER_SERVICE_PORT
66          value: '80'
67        - name: SQLFLOW_SERVER_SERVICE_PORT_SERVER
68          value: '80'
69        - name: NVIDIA_VISIBLE_DEVICES
70          value: all
71        - name: NVIDIA_DRIVER_CAPABILITIES
72          value: 'compute,utility'
73      resources: {}
74      volumeMounts:
75        - name: sqlflow-pv
76          mountPath: /datasets
```

B

Model Repository

	model	latency	accuracy	tasks
1	yolov3	21	0.98	7
2	yolov3	36	0.82	45 68 57 13 10 23 7 24 74 20 32 12 65 60 24 2 10 54 70 66 71 48 54 15 5 17 42 20 48 22 13 53 10 55 61 56 21 25 14 13 43 6 77 56 59 15 24 9 66 71 53 69 36
3	yolov3	1	0.81	21 40 77 49 47 77 40 78 45 16 28 45 67 66 78 46
4	yolov3	15	0.83	0 29 63 75 35 53 33 2 48 54 32 28 55 31 28 74 8 32 8 77
5	yolov3	8	0.98	50 79 41 64 24 20 44 15 30 14 19 26 7 53 47 60 34 32 19 67 24 38 47 5 79 63 32 42 74 66 30 17 68 64 60 78 17 39
6	yolov3	5	0.82	35 28 22 38
7	yolov3	10	0.88	41 74 77 70 25 48 50 62 44 54 0 16 19 9 51 10 68 23 14 63 21 46 3 56 46 54 79 71 14 77 15 25 53 58 29 44 37 22 54 76 12 59 26 76 71
8	yolov3	17	0.99	39 43 76 38 69 33 43 26 56 69 73
9	yolov3	20	0.91	52 27 43 1 26 30 64 22 52 3 70 12 39 48 61 70 13 36 23 22 66 53 9 41 57 76 11 36 16 30 57 35 41 2 36 46 27 72 28 2 13 41 36 40 18 38
10	yolov3	27	0.94	63 46 2 28 78 75 58 42 26 21 58 16 37 11 22 6 14 75 35 63 32 49 0 22 70 36 18 27 28 52 50 46 7 16 29 11 61 14 30 8 40 48 30 23 66 64 12 17 33 10 28 75 5 5 42
11	yolov3	21	0.86	52 57 56 78 10 72 48 19
12	yolov3	34	0.94	12 25 77 16 4 27 50 68 3 58 12 2 76 61 15 74 12 18 30 59 5 16 60 43 31 44 24 62 79 21 19 77
13	yolov3	11	0.97	33 4 39 67 34 34 42 78 38 48 5 8 40 69 5 7 78 37 78 33 27 51 77 30 43 46 9 67 3 67 54 63 28 37 64 37 18 78 70 52 10 32 46 63 18 36 29 39 28 54 29 55 27 60
14	yolov3	44	0.8	12 17 71 7 56 51 32 76 62 51 30 22 5 22 76 22 18 29 18 52 31 49 45 51 15 71 48 28 27 43 19 79
15	yolov3	2	0.95	70 11 65
16	yolov3	33	1.0	15 51 45 63 42 22 40 44 16 17 0 71 14 36 26 32 1 48 22 34 21 68 67 48 18 11 11 52 76 68 45 64 40 7 7 33 58 7 34 4 5 61
17	yolov3	21	0.95	50 76 45 21 71 25 1 13 38 27 38 33 1 31 38 28 75 31 51 66 67 32 74 32 5 0 37 72 43 24 16 41 24 21 27 46 35 62 68 62 32 45 42 57 25 32 33 6 65 51 31 7
18	yolov3	28	0.86	13 3 28 15 29 66 61 58 34 30 57 49 46 41 20 40 50 4 9 0 21 74 21 33 38 37 43 33 41 71 69 70 23 39 31 37 38 56 28 36
19	yolov3	7	0.96	66 10 0 16 30 3 78 59 53 69 49 74 46 27 33 72 68 39 32 7 33 62 14 77 43 61 36 12 53 26 12 25 1 41 55 55 64 15 40 60 70 49 50 32 20 52 38 65 57 52 19 64 42 47 21 1 38 12 18 77 77 39 15 37 36 1 32 30 7 50 50 32 43 41
20	yolov3	10	0.82	69 1 51 30 49 71 21 18 46 44 9 4 77 63 23 59 64 31 1 78 68 54 25 27 56 8 55 76 46 49 11 8 61 74 23 5 42 46 21 51 73 42 13 18 23 45

21	yolov3	40	0.89	52 33 54 14 57 61 40 72 65 22 2 1 72 51 71 77 62 48 0 42 39 3 57 43 53 43 74 75 59 66 74 5 26 67 31 11 3 54 1 13 59 56 8 38 4 70 76 73 5 79 59 31 7 66 32 46 75 44 64 37 47 73 15 65 52 69 66 28 19 17 5
22	yolov3	48	0.98	79 10 29 17 46 7 17 61 64 7
23	yolov3	16	0.86	29 55 28 43 55 5 50 63 52 34 77
24	yolov3	34	0.86	37 54 4 43 11 29 76 32 50 51 42 23 43 28 64 58 51 22 10 31 9 6 30 67 18 12 73 74 65 53 42 6 9 36 59 8 53 18 21 67 14 20 9 42 66 20 16 58 77 42 35 75 28 34 44 8 30 64 77 18 38 11 56
25	yolov3	43	0.83	52 76 68 25 69 31 59 62 29 66 32 71 60 72 27 68 38 60 9 67 78 63 20 35 55 28 45 56 51 4 79
26	yolov5x	21	0.98	24
27	yolov5x	36	0.82	20 44 15 30 14 19 26 7 53 47 60 34 32 19 67 24 38 47 5 79 63 32 42 74 66 30 17 68 64 60 78 17 39 35 28 22 38 41 74 77 70 25 48 50 62 44 54 0 16 19 9 51 10
28	yolov5x	1	0.81	68 23 14 63 21 46 3 56 46 54 79 71 14 77 15 25
29	yolov5x	15	0.83	53 58 29 44 37 22 54 76 12 59 26 76 71 39 43 76 38 69 33 43
30	yolov5x	8	0.98	26 56 69 73 52 27 43 1 26 30 64 22 52 3 70 12 39 48 61 70 13 36 23 22 66 53 9 41 57 76 11 36 16 30 57 35 41 2
31	yolov5x	5	0.82	36 46 27 72
32	yolov5x	10	0.88	28 2 13 41 36 40 18 38 63 46 2 28 78 75 58 42 26 21 58 16 37 11 22 6 14 75 35 63 32 49 0 22 70 36 18 27 28 52 50 46 7 16 29 11 61
33	yolov5x	17	0.99	14 30 8 40 48 30 23 66 64 12 17
34	yolov5x	20	0.91	33 10 28 75 5 5 42 52 57 56 78 10 72 48 19 12 25 77 16 4 27 50 68 3 58 12 2 76 61 15 74 12 18 30 59 5 16 60 43 31 44 24 62 79 21 19
35	yolov5x	27	0.94	77 33 4 39 67 34 34 42 78 38 48 5 8 40 69 5 7 78 37 78 33 27 51 77 30 43 46 9 67 3 67 54 63 28 37 64 37 18 78 70 52 10 32 46 63 18 36 29 39 28 54 29 55 27 60
36	yolov5x	21	0.86	12 17 71 7 56 51 32 76
37	yolov5x	34	0.94	62 51 30 22 5 22 76 22 18 29 18 52 31 49 45 51 15 71 48 28 27 43 19 79 70 11 65 15 51 45 63 42
38	yolov5x	11	0.97	22 40 44 16 17 0 71 14 36 26 32 1 48 22 34 21 68 67 48 18 11 11 52 76 68 45 64 40 7 7 33 58 7 34 4 5 61 50 76 45 21 71 25 1 13 38 27 38 33 1 31 38 28 75
39	yolov5x	44	0.8	31 51 66 67 32 74 32 5 0 37 72 43 24 16 41 24 21 27 46 35 62 68 62 32 45 42 57 25 32 33 6 65
40	yolov5x	2	0.95	51 31 7
41	yolov5x	33	1.0	13 3 28 15 29 66 61 58 34 30 57 49 46 41 20 40 50 4 9 0 21 74 21 33 38 37 43 33 41 71 69 70 23 39 31 37 38 56 28 36 66 10
42	yolov5x	21	0.95	0 16 30 3 78 59 53 69 49 74 46 27 33 72 68 39 32 7 33 62 14 77 43 61 36 12 53 26 12 25 1 41 55 55 64 15 40 60 70 49 50 32 20 52 38 65 57 52 19 64 42 47
43	yolov5x	28	0.86	21 1 38 12 18 77 77 39 15 37 36 1 32 30 7 50 50 32 43 41 69 1 51 30 49 71 21 18 46 44 9 4 77 63 23 59 64 31 1 78
44	yolov5x	7	0.96	68 54 25 27 56 8 55 76 46 49 11 8 61 74 23 5 42 46 21 51 73 42 13 18 23 45 52 33 54 14 57 61 40 72 65 22 2 1 72 51 71 77 62 48 0 42 39 3 57 43 53 43 74 75 59 66 74 5 26 67 31 11 3 54 1 13 59 56 8 38 4 70 76 73
45	yolov5x	10	0.82	5 79 59 31 7 66 32 46 75 44 64 37 47 73 15 65 52 69 66 28 19 17 5 79 10 29 17 46 7 17 61 64 7 29 55 28 43 55 5 50 63 52 34 77 37 54
46	yolov5x	40	0.89	4 43 11 29 76 32 50 51 42 23 43 28 64 58 51 22 10 31 9 6 30 67 18 12 73 74 65 53 42 6 9 36 59 8 53 18 21 67 14 20 9 42 66 20 16 58 77 42 35 75 28 34 44 8 30 64 77 18 38 11 56 52 76 68 25 69 31 59 62 29 66
47	yolov5x	48	0.98	32 71 60 72 27 68 38 60 9 67
48	yolov5x	16	0.86	78 63 20 35 55 28 45 56 51 4 79
49	yolov5x	34	0.86	75 31 76 52 38 74 70 26 57 30 64 38 18 41 57 48 42 52 22 64 70 61 60 27 0 26 50 20 26 65 77 18 60 59 62 38 23 68 69 61 66 19 36 17 21 77 26 32 8 8 20 68 54 74 23 13 11 42 10 24 47 68 43
50	yolov5x	43	0.83	13 6 2 67 4 28 35 77 46 78 6 20 23 63 71 6 21 64 50 9 67 33 71 8 51 33 61 7 59 41 27
51	yolov5x6	27	0.93	51 40 71 45 9 46 58 21 54 0 19 19 27

52	yolov5x6	36	0.83	21 18 67 35 29 22 56 10 42 72 31 12 7 8 61 23 34 3 55 68 15 24 62 26 47 35 16 78 57 24 55 49 52 24 15 67 73 46 20 66 12 25 0 77 11 24 54 47 77 40 59 79 31 56 33 38 20 14 9 67 79 3 47 70 42 5 20 57 59 54 77 24 58
53	yolov5x6	30	0.99	79 36 58 51 63 73 55 78 70 27 23 53 69 68 23 46 39 66 77 26 24 17 1 45 58 21 17
54	yolov5x6	27	0.9	56 33 43 75 2 26 26 52 55 7 62 31 79 76 46 70 69 55 74 15 14 56 1 75 76 18 62 45 65 52 28 38 20 51 59 64 14 52 44 16 64 30 10 45 78 59 42 64 5 76 55 42 20 68 18 49 15 17 61
55	yolov5x6	21	0.88	7 64 55 2 4 16 77 68 30 52 36 59 31 35 32 50 51 5 33 0 9 31 35 60 38 11 44 43 7 62 56 52 24 43 51 10 49 18 10 61 39 6 7 79 12 0 3 51 15 67 59 66 15 61 64 19 44
56	yolov5x6	32	0.85	61 21 0 63 1 60 76 22 14 24 46 32 70 54 24 77 35 57 31 39 14 48 13 19 15 41 35 25 51 34 22 33 61 78 7 32 3 67 57 46 21 58 64 40 78 78 62 62 47 27 71 42 14 51 18 78 10 35 49 79 4 5 29 53 72 29 38 78 54
57	yolov5x6	22	0.95	42 64 68 44 73 50 50 35 72 15 39 2 35 79 15 15 66 34 18 74 18 25 17 49 37 20 40 52 23 18 79 51 32 37 30 40 50 73 68 15 63 64 20 54 23 6 24 0 48
58	yolov5x6	44	0.89	28 1 73 21 0 76 32 40 30 53 33 22 3 34 49 70 37 33 12 37 79 65 6 79 3 32 44 29 46 67 66 28 57 43 20 52 31 25 63 52 59 17 22 54 13 30 23 50 21 71 3 23 12 77 7 52 30 26 8
59	yolov5x6	48	0.91	73 39 56 60 25 65 27 18 1 34 2 17 11 53 59 11 3 0 51 32 5 11 69 9 46 74 32
60	yolov5x6	19	0.8	38 42 63 23 40 63 63 31 18 40 5 79 74 40 25 68 49 46 13 21 49
61	yolov5x6	39	0.92	51 14 30 10 4 7 54 59 41 35 56 13 11 52 66 21 34 61 60 74 8 75 27 25 36 14 3 72 43 8 55 17 60 32 27 52 50 79 26 37 24 56 40 52 38 19 44 20 63 21 65 68 18 68 16 78 59 56 44 35 39 61 32 67 53 62 75 8 34 33
62	yolov5x6	26	0.92	25 67 1 8 27 29 77 10 61 16 79 74 76 0 30 3 64 38 67 30 21 26 10 69 33 36 9 0 55 69 10 3 39
63	yolov5x6	28	0.92	77 37 16 25 46 7 43 0 56 14 55 55 52 78 53 60 0 14 60 66 39 71 7 16 66 61 69 50 58 79 14 77 12 11 79 34 6 69 44 70 72 12 46 7 29 60 73 46 47 73 76 45 36 39 37 6 43 58 1 29 48 55 45 8 34 29 0 79 19 4 71 1 8 26 40 32
64	yolov5x6	46	0.99	70 7 38 29 3 74 50 16 49 42 70 8 31 25 34 54 70 5 63 24 46 20 22 0 69 22 27 25 64 45 38 57 52 52 32 69 79 5 33 64 53 53 53 12 33 51 55 54 12 10 78 71
65	yolov5x6	4	0.94	29 12 63 7 70 63 64 44 69 75 34 14 59 33 24 3 41 30 22 53 36 6 38 31 62 47 62 34 71 42 20 50 22 2 63 0 5 72 52 9 11 40 76 61 2 22 47 29 1
66	yolov5x6	5	0.87	75 1 52 64 75 78 62 76 48
67	yolov5x6	1	0.89	36 49 14 73 21 32 71 0 46 24 37 16 52 68 7 60 9 34 63 66 68 13 66 0 34 70 16 37 17 38 14 37 62 0 33 49 77 31 57 58 73 76 28 76 53 58 31 13 38 59 46 62 61 39 71 2 59 25 57 32 30 52 41 78 57 45 25 73 74 6 2 30 77 78 42
68	yolov5x6	41	0.94	18 56 42 4 11 24 22 6 4 79 28 53 30 21 27 5 35 72 48 51 24 76 40 28 40 47 23 28 37 5 23 68 20 13 56
69	yolov5x6	39	0.81	29 28 54 28 63 60 69 72 79 48 52 57 39 78 61 57 5 76 25 79 37 44 60 15 41 63 7 2 46 75 52 66 25 37 3 25 64 0 52 46 17 54 35 78 60
70	yolov5x6	43	0.93	50 46 23 65 0 32 12 75 25 55 69 5 14 2 64 58 24 45 14 77 32 72 38 20 24 71 27 37 16 54 52 73
71	yolov5x6	48	0.93	5 66 50 2 32 7 36 66 73 11 35 42 39 52 20 73 7 52
72	yolov5x6	40	0.84	69 57 62 16 28 45 68 67 61 69 38 74 63 72 12 34 24 34 22 40 75 77 38 75 21 15 55 68 68 11 36 57 27 79 1 12 1 8 63 15 27 7 14 75 12 61 20 37 62 66 33 57 74 72 5 0 2 4 64 57 35 30 4 75 14 2 52 54 15 0 59
73	yolov5x6	23	0.83	62 1 67 41 44 6 56 49 47 72 59 38 10 65 59 73 34 50 51 51 24 55 68 65 37 57 44 57 41 32 27 55 63 76 56 26 62 53 22 38 20 58 43 19 45
74	yolov5x6	39	0.86	77
75	yolov5x6	6	0.87	75 37 34 58 1 40 37 58 7 24 24 41 27 32 14 25 13 78 36 24 3 56 31 28 76 2 60 59 72 40 15 37 7 38 47 60 60 48 8 68 39 63 54 13 61 40 73 74
76	yolov5m	21	0.98	68 24 25 33 18 78 46 78 7 77 8 38 49 14 17 45 57 77 26 31 58 23 37 42 26
77	yolov5m	36	0.82	46 18 24 39 75 39 6 68 39 46 21 67 73 24 71 32 12 65 7 2 20 3 35 17 17 55 43 52 25 29 51 67 47 20 71 79 8 23 60 56 9
78	yolov5m	1	0.81	46 60 65 62 21 13 22 27 25 17 32 14 52 25 60 68 24 57 63 12 18 26 77 25 0 22 14 74 39 75 19 9 71 53 66 27 2 71 19 14 13 22 61 11 28 44 2 3 10 31 29 55 29 33 41 65 49 25 48 53 48 0 38 7 21 5 15 77 63
79	yolov5m	15	0.83	30 32 72 44 40 67 24 78 67 50 48 35 52 31 1 64 31 18 66 15 43 39 20 1 0 45 29 26

80	yolov5m	8	0.98	7 52 3 21 79 62 69 67 8 72 43 76 72 58 68 58 63 52 37 47 51 23 70 0 23 22 59 8 2 18 3 14 79 67 34 64 44 42 34 78 44 60 67 42 22 41 68 41 39 42 67 13 25 38 75 5 22 39 73 14 47 71 1 33 70 60 19 15 53 69 20
81	yolov5m	5	0.82	35 38 49 58 12 75 53 24 22 33 24 30 0 35 45 63 8 41 14 62 10 29 30 37 39 10 67 54 77 45 73 68 23 64 32 14 30 38 0 37 66 42 61 18 0 45 71 70 53
82	yolov5m	10	0.88	0
83	yolov5m	17	0.99	42 40 39 24 40 72 31 4 29 35 35 48 30 47 0 79 7 52 9 24 51 75 14 61 40 12 27 44 0 54 64 16 17 59 37 72 39 43 70 23 30 73 63 48 44 16 59 44 48 17 51 48 6 76 66 35 44 78 0 38 18 36 48 3 51 48 3 57 16 63 21 33 31
84	yolov5m	20	0.91	68 38 71 69 53 12 46 40 17 78 61 35 58 31 30 48 30 20 6 74 35 36 73 38 71 56 71 67 49 61 47 0 65 21 21 39 76 60 2 40 77 43 59 66 78 50 22 37 13 15 46 38 54 27
85	yolov5m	27	0.94	29 48 15 73 29 51 49 64 13 52 59 34 3 64 57 0 9 38 17 13 42 52 9 38 25 57 51 69 53 69 29 62 76 20 55 1 62 54 10 31 69 47 48 10 40 25 72 22 49 55 3 28 24 20 65 9 11 21 58 64 78 71 0 42 34 66 57 6 70 2 12 4 0 34 76 8 23 61
86	yolov5m	21	0.86	27 15 11 53 47 60 29 50 69 38 74 47 5
87	yolov5m	34	0.94	59 11 35 51 73 12 35 16 9 58
88	yolov5m	11	0.97	48 21 47 71 56 51 67 3 50 53 79 39 2 30 47 65 41 56 41 79 44 79 33 54 15 27 37 5 64 34 42 44 42 44 52 57 2 64 73 49 31 31 41 39 14 2 47 35 50 32 32 41 42 16 28 31 11 38 16 56 9 64 26 2 38 30 53 31 30 36 15 21 47
89	yolov5m	44	0.8	73 44 4 66 43 18 59 29 77 9 23 8 14 47 72 37 41 30 39 5 33 74 19 37 15 12 64 0 69 50 47 18 68 67 66 74 69 55 65 50 48 4 50 56 78 38 10 24 39 58 55 0 38 70 73
90	yolov5m	2	0.95	55 15 49 55 6
91	yolov5m	33	1.0	32 29 78 19 28 78 41 24 10 69 15 46 29 44 19 74 79 44 12 77 49 31 53 78 2 69 22 42 2 20 56 39 37 26 21 3 62 28 9 54 1 28 6 44 67 32 29 69 12 15 74 16 23 15 11 4 71 24 37
92	yolov5m	21	0.95	63 60 49 78 11 0 47 23 12 56 28 35 8 16 6 21 34 47 45 42 25 35 75 26 53 74 66 77 60 52 49 36 60 47 69 34 61 32 47 18 73 12 41 57 70 2 34 66 71 14 26 39 58 45 52 13 9 25 47
93	yolov5m	28	0.86	64 39 21 45 44 68 67 70 35 34 24 61 73 20 76 72 58 55 15 41 63 27 71 73 60 79 22 34 55 7 37 0 75 73 63 57 64 39 14 22 26 15 76 26 49 32 25 52 60 77 5 46 64 10 1 56 72 56 21 37 72 74 14 14 49 45 30 77 71 58 62 42
94	yolov5m	7	0.96	18 58 75 12 28 45 49 22 76 56 38 11 19 50 74 40 59 2 8 75 71 29 10 72 63 13 35 61 27 5 48 36 67 6 38 18 71 53 15 52 46 74 53 3 24 43 25 26 1 6 18 34 31 43 51 36
95	yolov5m	10	0.82	8 2 19 62 75 25 35 79 62
96	yolov5m	40	0.89	76
97	yolov5m	48	0.98	76 74
98	yolov5m	16	0.86	70 47
99	yolov5m	34	0.86	78 31 33 43 9 56 46 37 36 10 50 77 19 45 34 35 7 71 1
100	yolov5m	43	0.83	3 46 61 6 3 8 36 48 14 15 48 43 43 57 67 46 50 2 0 58 18 45 65 27 58 20 43 11 46 37 40 6 44 30 58 33 65 21 57 0 77 67 15 70 3 47 42 1 47 16 68 21 61 29 44 66 34 77 38 10 21 16 78 7 53 53 67
101	yolov5m6	21	0.98	77 12 28 13 61 6 57 45 61 51 6 57 40 7 4 7 18 51 72 2 46 18 29 35 11 52 76 37 78 69 24 69 29 77 24 42 10 65 53 20 13 26
102	yolov5m6	36	0.82	6 31 49 65 54 0 50 37 76 70 20 54 15 45 29 70 14 13 76 47 43 75 62 25 25 72 42 61 44 44 53 32 28 5 20 61 7 51 50 21 9 77 11
103	yolov5m6	1	0.81	67 35 37 41 9 15 76 29 52 25 37 15 22 36 45 66 3 58 72 72 16 74 32 62 6 73 37 2 45 13 41 33 3 4 30 65 69 74 36 65 56 77 70 16 42 57 5 29 69 41 51 46 58 39 15 75 52 56 25 79 36 4 11 42 22 28
104	yolov5m6	15	0.83	56 32 58 68 66 22 2 73 1
105	yolov5m6	8	0.98	33 7 63 69 5 42 15 35 46 23 20 4 23 25 46 31 25 18 11 53 45 6
106	yolov5m6	5	0.82	20 5 67 11 44 44 26 56 53 37 46 11 64 56 22 56 76 74 71 36 50 31 23 28 0 39 61 32 66 58 46 79 37 53 33 34 25 78 19 52 75 63 79 41 38 39
107	yolov5m6	10	0.88	6 8 0 42 18 44 43 5 2 62 36 59 19 23 30 17 39 55 23 16 9 30 48 49 37 51 21 0 31 18 39 44 49 44 23 36 69 74 68 61 43 46 53 74 12 35 17 11 4 26 65 16 75 25 63 24 16 79 56 19 33 65 29 38 53 69 64 19 54 75 69 70 22 28 73 57
108	yolov5m6	17	0.99	50 20 29 32 43 47 46 57 79 1 12 39 19 55 59 57 66 57 53 55 2 33 44 53 37 62 73 60 69 72 12 48 8 64 26 52 17 27 42 48 53 14 45 28
109	yolov5m6	20	0.91	15

110	yolov5m6	27	0.94	33 70 14 79 65 13 26 51 23 10 72 51 29 19 2 36 22 56 58 21 34 39 54 28 11 59 2 3 10 27 1 61 70 12 30 67 79 61 57 7 42 3 53 12 74 75 59 34 69 50 21 75 43 3 6 1 65 18 53 7 70 46 25
111	yolov5m6	21	0.86	34 55 71 63 11 12 49 79 67 54 69 41 31 33 73 9 74 1
112	yolov5m6	34	0.94	19 11 72 41 23 35 58 29 14 1 49 39 27 57 19 32 11 51 64 76 60 8 39 18 3 46 40 31 42 62 23 16 31 75 6 18 45 64 53 43 50 42 13 7 0 5 13 8 38 66 59 66 5 25 6 15 74 59 19 50 22 2 75
113	yolov5m6	11	0.97	75 50 41 8 47 27 9 70 18 78 69 61 35 35 56 53 22 13 48 21 7 6 77 28 43 70 28 73 6 53
114	yolov5m6	44	0.8	13 72 54 50 48 38 57 34 68 65 51 24 47 38 38 31 78 78 8 36 72 37 77 34 39 19 67 31 70 64 55 59 56 26 77 30 52 31 43 0 51 10 72 39 76 13 69 69 49 11 12 59 13 31 19 25 64 66 18 36 48 14 40 45 78 63 0 54
115	yolov5m6	2	0.95	20 23 55 59 40 37 2 34 69 43 40 60 45 18 68 26 60 45 58 23 1 40 13 77 62 51 47 61 57 64 23 1 2 67 4 8 77 14 13 6 68 28 73 34 58 48 9 14 31 24 24 22 74 25 19 74 61 7 53
116	yolov5m6	33	1.0	61 75 68 17 43 28 17 52 48 59 21 71 9 10 20 15 46 38 19 60 6 23 31 0 71 9 67 65 2 65 47 25 19 4 52 70 4 22 62 59 40 69 70
117	yolov5m6	21	0.95	23 76 31 58 27 72 30 18 56 44
118	yolov5m6	28	0.86	36 59 56 56
119	yolov5m6	7	0.96	1 48 8 54 23 53 70 23 42
120	yolov5m6	10	0.82	68 61 30
121	yolov5m6	40	0.89	72 26 35 21 60 32 18 9
122	yolov5m6	48	0.98	29 74 30 58 39 25 73 73 53 75 77 60 26 6 67 77 33
123	yolov5m6	16	0.86	51 36 33 40 13 64 51 73 6 56 21 59 30 45 68 57 19 31 70 60 75 33 17 78 0 41 67 62 34 24 72 12 50 63 64 33 59 35 19 6 23 60 18 62 9 23 40 76 6 79 34 6 32 12 70 51
124	yolov5m6	34	0.86	45 17 6 57 57 72 6 18 30 6 13 53 61 40 38 16 47 38 0 65 48 69 31 65 0 75 76 19 63 38 54 12 21 18 45 59 60 12 28 11 13 60 59 2
125	yolov5m6	43	0.83	60

Table B.1: Model variations of YOLO generated for the model repository.

Bibliography

References

[1] Aston Zhang, Zachary C Lipton, Mu Li, and Alexander J Smola. Dive into deep learning. *arXiv preprint arXiv:2106.11342*, 2021.

[2] María García-Ordás, José Benítez-Andrades, Isaías García, Carmen Benavides, and Hector Alaiz Moreton. Detecting respiratory pathologies using convolutional neural networks and variational autoencoders for unbalancing data. *Sensors*, 20, 02 2020.

[3] Arden Dertat. Applied Deep Learning - Part 4: Convolutional Neural Networks. *Medium*, Jun 2018.

[4] Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q. Ngo, and XuanLong Nguyen. Learning models over relational data: A brief tutorial. *CoRR*, abs/1911.06577, 2019.

[5] Umar Syed and Sergei Vassilvitskii. Large-scale in-database machine learning with pure sql, November 19 2019. US Patent 10,482,394.

[6] How Instagram hit 1 billion users, Jun 2020.

[7] More Than 500 Hours Of Content Are Now Being Uploaded To YouTube Every Minute - Tubefilter, May 2019.

[8] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.

[9] Yi Wang, Yang Yang, Weiguo Zhu, Yi Wu, Xu Yan, Yongfeng Liu, Yu Wang, Liang Xie, Ziyao Gao, Wenjing Zhu, et al. Sqlflow: A bridge between sql and machine learning. *arXiv preprint arXiv:2001.06846*, 2020.

[10] Kaggle. 2017 kaggle machine learning & data science survey, Oct 2017.

[11] Milan Cvitkovic. Supervised learning on relational databases with graph neural networks, 2020.

[12] Hideyuki Tamura and Naokazu Yokoya. Image database systems: A survey. *Pattern Recognition*, 17(1):29–43, 1984. Knowledge Based Image Analysis.

[13] Darrell Woelk, Won Kim, and Willis Luther. An object-oriented approach to multimedia databases. *ACM Sigmod Record*, 15(2):311–325, 1986.

[14] O. Kalipsiz. Multimedia databases. In *2000 IEEE Conference on Information Visualization. An International Conference on Computer Visualization and Graphics*, pages 111–115, 2000.

[15] Arjen P De Vries, Menzo Windhouwer, Peter MG Apers, and Martin Kersten. Information access in multimedia databases based on feature models. *New Generation Computing*, 18(4):323–339, 2000.

[16] Marcelle Kratochvil. *Managing multimedia and unstructured data in the Oracle database*. Packt Publishing Ltd, 2013.

[17] Russell Sears, Catharine Van Ingen, and Jim Gray. To blob or not to blob: Large object storage in a database or a filesystem? *arXiv preprint cs/0701168*, 2007.

[18] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, Peter Vajgel, et al. Finding a needle in haystack: Facebook's photo storage. In *OSDI*, volume 10, pages 1–8, 2010.

[19] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. F4: Facebook's warm blob storage system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 383–398, USA, 2014. USENIX Association.

[20] Tom M Mitchell et al. Machine learning. 1997.

[21] Kenji Doya. Reinforcement learning: Computational theory and biological mechanisms. *HFSP journal*, 1(1):30, 2007.

[22] Frank Y Shih. *Image processing and pattern recognition: fundamentals and techniques*. John Wiley & Sons, 2010.

[23] Li Deng and Dong Yu. Deep learning: methods and applications. *Foundations and trends in signal processing*, 7(3–4):197–387, 2014.

[24] Andrew Ng, K Katanforoosh, and YB Mourri. Neural networks and deep learning. *Deeplearning. ai on Coursera*, 2017.

[25] Convolutional Neural Network, May 2019. [Online; accessed 31. Jul. 2021].

[26] SH Shabbeer Basha, Shiv Ram Dubey, Viswanath Pulabaigari, and Snehasis Mukherjee. Impact of fully connected layers on performance of convolutional neural networks for image classification. *Neurocomputing*, 2019.

[27] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[29] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[31] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[32] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[33] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[34] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28:91–99, 2015.

[35] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[36] Christopher Ré, Divy Agrawal, Magdalena Balazinska, Michael Cafarella, Michael Jordan, Tim Kraska, and Raghu Ramakrishnan. Machine learning and databases: The sound of things to come or a cacophony of hype? In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 283–284, 2015.

[37] Dan Olteanu. The relational data borg is learning. *arXiv preprint arXiv:2008.07864*, 2020.

[38] Sandeep Singh Sandha, Wellington Cabrera, Mohammed Al-Kateb, Sanjay Nair, and Mani Srivastava. In-database distributed machine learning: demonstration using teradata sql engine. *Proceedings of the VLDB Endowment*, 12(12), 2019.

[39] Arash Fard, Anh Le, George Larionov, Waqas Dhillon, and Chuck Bear. Vertica-ml: Distributed machine learning in vertica database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 755–768, 2020.

[40] Factorised Databases (FDB) Research Group University of Zurich. Factorised in-database analytics, Aug 2020.

[41] Maximilian Schleich. Structure-aware machine learning over multi-relational databases. In *Proceedings of the 2021 International Conference on Management of Data*, pages 6–7, 2021.

[42] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Machine learning over static and dynamic relational data. In *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*, pages 160–163, 2021.

[43] Amir Shaikhha, Maximilian Schleich, Alexandru Ghita, and Dan Olteanu. Multi-layer optimizations for end-to-end data analytics. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pages 145–157, 2020.

[44] Dan Olteanu and Maximilian Schleich. Factorized databases. *ACM SIGMOD Record*, 45(2):5–16, 2016.

[45] Maximilian Schleich and Dan Olteanu. Lmfao: An engine for batches of group-by aggregates. *arXiv preprint arXiv:2008.08657*, 2020.

[46] Umar Syed and Sergei Vassilvitskii. Sqml: large-scale in-database machine learning with pure sql. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 659–659, 2017.

[47] Reynold S Xin, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*, pages 13–24, 2013.

[48] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394, 2015.

[49] Choudur Lakshminarayan, Thiagarajan Ramakrishnan, Awny Al-Omari, Khaled Bouaziz, Faraz Ahmad, Sri Raghavan, and Prama Agarwal. Enterprise-wide machine learning using teradata vantage: An integrated analytics platform. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 2043–2046, 2019.

[50] Release notes of BigQuery ML Google Cloud, Jul 2018.

[51] Umar Syed. XLDB-2019: Design of BigQuery ML, May 2019.

[52] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[53] Kubernetes, Production-Grade Container Orchestration, 2021.

[54] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes: Lessons learned from three container-management systems over a decade. *Queue*, 14(1):70–93, January 2016.

[55] Apache Airflow, 2021.

[56] Luigi, 2021.

[57] argoproj. Argo Workflows, 2021.

[58] Kubeflow, Aug 2021. [Online; accessed 12. Aug. 2021].

[59] SQLFlow Parser, Oct 2020. [Online; accessed 13. Jul. 2021].

[60] Keras Team. Keras documentation: Models API, Jul 2021. [Online; accessed 13. Jul. 2021].

[61] Kief Morris. *Infrastructure as code: managing servers in the cloud.* " O'Reilly Media, Inc.", 2016.

[62] MinIO Inc. MinIO | High Performance, Kubernetes Native Object Storage, Jul 2021.

[63] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes challenge: A retrospective. *International Journal of Computer Vision*, 111(1):98–136, January 2015.

[64] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[65] Jian Guo and Stephen Gould. Deep cnn ensemble with data augmentation for object detection. *arXiv preprint arXiv:1506.07224*, 2015.

[66] Hector Ballega Fernandez & Ziyu Li. sqlflow-object-detection, Jul 2021.

[67] Hongkun Yu, Chen Chen, Xianzhi Du, Yeqing Li, Abdullah Rashwan, Le Hou, Pengchong Jin, Fan Yang, Frederick Liu, Jaeyoun Kim, and Jing Li. TensorFlow Model Garden. `https://github.com/tensorflow/models`, 2020.

[68] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch Hub. `https://github.com/pytorch/hub`, 2020.

[69] K3d: Running CUDA workloads on Kubernetes, Jul 2021.

[70] Ruben Mayer and Hans-Arno Jacobsen. Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools. *ACM Computing Surveys (CSUR)*, 53(1):1–37, 2020.

[71] MG Murshed, Christopher Murphy, Daqing Hou, Nazar Khan, Ganesh Ananthanarayanan, and Faraz Hussain. Machine learning at the network edge: A survey. *arXiv preprint arXiv:1908.00080*, 2019.

[72] Salma Abdel Magid, Francesco Petrini, and Behnam Dezfouli. Image classification on iot edge devices: profiling and modeling. *Cluster Computing*, 23(2):1025–1043, 2020.

[73] Ying Xiong, Yulin Sun, Li Xing, and Ying Huang. Extend cloud to edge with kubeedge. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 373–377, 2018.

[74] Ali Basiri, Niosha Behnam, Ruud De Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. Chaos engineering. *IEEE Software*, 33(3):35–41, 2016.

[75] Casey Rosenthal and Nora Jones. *Chaos engineering*, volume 1005. O'Reilly Media, Incorporated, 2020.

[76] Chaos mesh: A powerful chaos engineering platform for kubernetes, Jul 2021. [Online; accessed 28. Jul. 2021].