Aalto University
School of Science
Master's Programme in ICT Innovation

Luca Scotton

Engineering framework for scalable machine learning operations

Master's Thesis
Espoo, December 30, 2020

Supervisor: Professor Hong-Linh Truong, Aalto University
Advisor: Dr. Seamus Moloney (Tech), F-Secure Corporation

Aalto University
School of Science
Master's Programme in ICT Innovation

ABSTRACT OF
MASTER'S THESIS

Author: Luca Scotton

Title:
Engineering framework for scalable machine learning operations

Date:	December 30, 2020	**Pages:**	77
Major:	Data Science	**Code:**	SCI3095
Supervisor:	Professor Hong-Linh Truong		
Advisor:	Dr. Seamus Moloney		

With the evolution of algorithms and solutions in the artificial intelligence field, new and modern methods and practices are required to successfully leverage these technologies. Therefore a new field named Machine Learning Operations (MLOps) has grown rapidly in the last five years. The goal is to increase integration between the pure research in the artificial intelligence domain and the traditional software engineering domain to generate business value faster, by rapidly shifting machine learning algorithms to the production stage.

The thesis aims to provide a novel machine leaning framework to exploit the business potential of solutions to artificial intelligence and data mining problems in the industry by introducing novel tools and practises which ensure automation and maintenance are guaranteed on the long run. In particular, the thesis introduces the Model-as-a-Package concept. The design considers a machine learning pipeline as a highly specialized package. Based on this, a novel framework is proposed and implemented. The ultimate goal is to enable robust automation in the model versioning, model deployment and monitoring aspects of machine learning operations.

Compared to the state-of-the-art end-to-end open source frameworks in the market, the features the framework introduces outperform the alternatives in the context of model life cycle processes. In particular, major improvements have been proposed for dependency management, automated logging and automatic model update. Moreover, the framework improves the state of the art by decoupling logic into two highly integrated components, namely Trainer and Predictor. This expands the set of inference services supported by the framework by adding low resources serving channels such as serverless functions and IoT devices.

Keywords: Machine Learning Operations, Machine Learning, DevOps, Data Science, Software engineering, Framework, Serverless

Language: English

Acknowledgements

First I want to thank my employer, F-Secure, for giving me the opportunity to research and build this project during these unusual months. In particular, I want to thank my advisor Seamus Moloney and my teammates Lasse Hyyrynen, David Karpuk and Alejandro Antillon. I want to express my gratitude toward my supervisor Prof. Hong-Linh Truong for his guidance and precious feedback received during these months. I want to thank my flatmate and friend Adam for having supported me during the lockdown months with his tacos care packages and his herbal tea. Finally, I thank Sol for keeping me motivated until the thesis has been finalized.

Espoo, December 30, 2020

Luca Scotton

Abbreviations and Acronyms

AICE	Artificial Intelligence Center of Excellence
API	Application programming interface
AWS	Amazon Web Services
CI	Continuous integration
CD	Continuous delivery (or when specified, Continuous deployment)
CLI	Command line interface
DevOps	Development and operations
EDA	Exploratory data analysis
ETL	Extract-Transform-Load
FaaS	Function-as-a-Service
IaC	Infrastructure as code
KPI	Key Performance Indicator
ML	Machine learning
MLOps	Machine learning operations
NLP	Natural language processing
OLTP	Online Transactional Processing
RDBMS	Relational database management system
VCS	Version control system
YAML	YAML Ain't Markup Language

Contents

Chapter 1

Introduction

In the last decade, large firms and rising start-ups have brought increasing attention to their data collected through time. As a consequence of this, several aspects of the business, such as customer acquisition [47] or cost savings [38], have been revised and innovated to make more informed decisions thanks to the data available. This so-called "data-driven" culture [27] has been highly beneficial if applied rigorously. As a proof of this, there are newly-founded start-ups, such as Uber, Spotify or Lyft, which have at their core values highly specialized data-driven approaches which use machine learning approaches to exploit open and closed domains data sources to generate new services for both business-to-business and business-to-consumer clients.

The opportunities the market has offered in the past years have been increasingly expanding thanks to the successful innovations in the fields of cloud computing [12, 48] and parallel and distributed computation [16]. These have introduced more efficient services and tools to bring data closer to industrial applications in every domain. As a consequence, the largest cloud services providers, Amazon (AWS) [9], Microsoft[35] and Google [28] among all, have invested in distributed infrastructures to attract an increasing amount of companies from on-premises architectures to their managed cloud-based products [26].

The last ten years have also been crucial for the artificial intelligence (AI) and machine learning (ML) fields. In particular, major advances in parallel computing and more robust enterprise-level frameworks have enabled the rapid growth of deep learning and several other sub-fields of the machine learning domain. One of the strongest business decisions which permitted machine learning frameworks like Tensorflow [1] by Google Brain and Pytorch [39] by Facebook's AI Research to become a standard in both the research

and industrial environments is the fact that these have become publicly available as open-source software. This game-changing decision has led to strong positive implications in the development of machine learning-based solutions since these frameworks are both highly optimized for fast computation and architecture agnostic. This means they can be extremely flexible for multiple hardware configurations and scalable to production-level environments, thus thanks to these and open-source, several companies have started to experiment with these at affordable costs.

As a consequence of these two factors, namely the transition to cloud architectures and the rapid evolution of the artificial intelligence field, the last four years have seen the rise of a new field named Machine Learning Operations (MLOps) [4] which aims to reduce the gap between the pure research in the machine learning domain and the traditional software engineering domain. As artificial intelligence-driven solutions have been rapidly expanding in different domains, companies have seen an increasing value in the productionization of machine learning algorithms. However, the pace at which these algorithms are ready to be served to the users influences the cost of the feature. Studies show that the current time for a model to transition to the production environment is, on average, over a month [6]. Moreover, with the change in the trends and behaviours also machine learning algorithms have to be maintained and updated to guarantee high-quality predictions, so business value. Therefore, MLOps initiatives aim to establish resilient and efficient workflows by creating robust pipelines [3], well-established practices [18, 44] and auxiliary frameworks and tools. This is a non-trivial task as it involves the creation and integration of various components such as large clusters, model archives, inference endpoints. In particular, the thesis addresses the need for a robust end-to-end machine learning life-cycle, thus the transition from the model training to the deployment and maintenance of the trained model for real-time inference. Specifically, the research focuses on providing reliable model versioning and automated model updates and logging.

1.1 Contribution

The thesis introduces a methodology (named Model-as-a-Package) for handing code dependencies as highly specialized libraries to simplify deployments of machine learning models. Consequently, the thesis proposes a machine learning framework addressing particularly model versioning and model de-

ployment problems. The framework has proven itself in practice to support Function-as-a-service (FaaS) deployment channels and offers additional solutions for the automation of model update tracking and tracing compared to the state-of-the-art frameworks.

The core features the proposed framework provides are:

1. Highly available server-less model archive solution
2. Automatic tracing and tracking for analysis and monitoring
3. Automatic model update capabilities
4. End-to-end flow, from training to inference
5. Support for FaaS deployments and edge devices

Two central use cases have been selected for the evaluation: a supervised classification problem of textual news articles using the `20newsgroups` publicly available dataset [31] and an internal unsupervised learning problem focused on malicious events retrieval based on event similarity scores.

The evaluation shows that the proposed framework satisfies all the requirements for the aforementioned features. In spite of this, the framework still requires additional solutions for parameter and metric visualization. Therefore, the thesis proves the compatibility of the framework with external visualization solutions and the success with both the use cases. Complete architectural and procedural documentation is provided to replicate and adapt the proposed solution to other machine learning platforms. Moreover, the framework has proven to outperform the current state-of-the-art in terms of functionalities for the end-to-end model and lifecycle management. In particular, the proposed solution is the only solution supporting the end-to-end process with real-time inference deployments as Function-as-a-Service channels, in this case, the AWS Lambda serverless service.

1.2 Structure of the Thesis

Firstly, the thesis addresses in Chapter 2 the characteristics and requirements of machine learning operations. Open problems are discussed here. Section 2.5 studies the state-of-the-art solutions in the deployment and maintenance of robust and maintainable end-to-end machine learning pipelines. Consequently, Chapter 3 defines the requirements for the thesis and Section 3.1 compares the state-of-the-art tools against the aforementioned requirements.

Consecutively, the thesis describes the Model as a Package paradigm in Section 5.2. Moreover, the architecture design and principles of the model archive are discussed in Section 5.3.1, the training pipeline in Section 5.3.2 and serving solution in Section 5.3.3. Ultimately, the end-to-end implementation in Python is discussed and the various architectural component analyzed. Pipelines and interactions among those components are addressed as well.

The framework is evaluated in Section 6. The evaluation proceeds with an absolute comparison against the requirements defined in Chapter 3. For each of them, the thesis discusses how it has been solved in contrast with the other state-of-the-art solutions. Additionally, a feasibility demo is proposed and proven workload conditions are reported.

Ultimately, Chapter 7 reports the finding and future work for improved model management in the machine learning operation domain.

Chapter 2

Background

As of today, companies of different sizes have started to investigate methods to exploit the benefits a data-driven approach introduces. However, it is well-known that there are fundamental differences in how the code is managed in research and industry due to the different needs of the two worlds. Indeed, industrial production-ready code requires validation and features, such as unit and integration test, linting and security vulnerability checks, which are superfluous for the large majority of academic projects. Therefore, to support rigorous industrial requirements, the long history of software engineering has provided a large set of tools and best practices which aim to optimize costs, time and resources in the development and maintenance phases of extensive software projects. Nonetheless, not all of them are suitable to solve problems and challenges in data science and machine learning domains. These two fields have substantially different ways of operating in contrast with the aforementioned traditional software engineering processes. Therefore, the so-called machine learning operations have become part of the industry-standard to glue well-established development and maintenance processes with data science and data mining initiatives.

2.1 Machine Learning Operations

MLOps (a compound of "machine learning" and "operations") [4] is a set of methodologies and practices aiming for collaboration and coordination between data scientists and development and operations (DevOps) professionals. The ultimate goal of MLOps initiatives is to increase the speed and reliability of end-to-end data and machine learning model lifecycles. This ranges from managing training data and resources to maintaining ML mod-

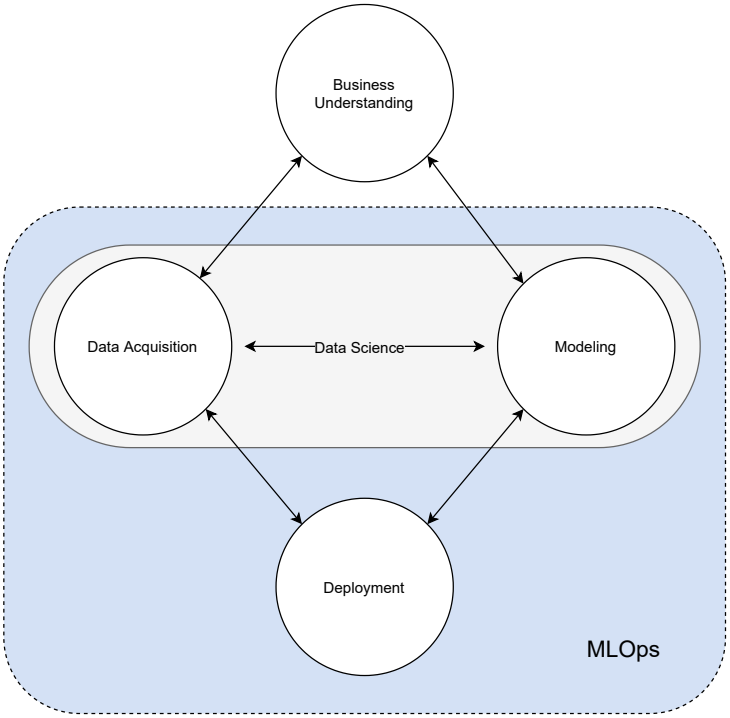

Figure 2.1: MLOps domain area of interest [22, 23, 33]

els and pipelines in production environments. MLOps practices introduce tools and frameworks to foster automation and improve the quality of service of ML models under business and regulatory requirements. Although the field is relatively recent, it is rapidly evolving under multiple aspects including model management (software development lifecycle, continuous integration/continuous delivery (CI/CD)), data management, governance, orchestration and deployment.

2.1.1 Areas of interest for MLOps

Figure 2.1 shows the four main macro-areas of action of most common machine learning projects in the industry [22, 23, 33]. Besides *Business Understanding*, which drives the process of matching results to solve business and client problems, the other three belong to the specific areas of interest of data scientists and data engineers. Each of these areas exhibits several own challenges which are going to be discussed shortly. Additionally, it is strongly dependent on resources and methods, such as dependency libraries or specific CI/CD tools, used in the other connected to it. Each sub-area is characterized by specific peculiar aspects described below and highlights specific challenges

to be solved successfully to guarantee a maintainable ecosystem.

Data acquisition and understanding

- Architecture specifications: It refers to where the data is being processed. It can be on-premises, on private/public cloud resources. Each environment has its specific characteristics. In particular, it is worth noticing that cloud providers share many of the core capabilities, however, the service APIs may differ.

- Variety of data sources: Data may come from a large and diverse variety of sources including data lakes [13], data warehouses [13], online transactional processing (OLTP) databases and real-time generated sources such as message brokers. Each of them, among all the specific characteristics, have different access methodologies, availability guarantees and data retrieval performances.

- Data velocity and extraction types: Data may be fetched/extracted in a batch or streaming approach. This aspect is tightly connected to the category of the data source.

- Variety of data: These refer to the format data has. Examples are, non extensively: JSON-formatted strings, Comma/tab-separated value entries, images, audio, raw text and queries to various database engines. Each one of these requires different wrangling, exploration and cleaning procedures.

Modeling

- Feature engineering: It refers to all the processes of generation, transformation and cleaning of features depending on each specific use case.

- Model training: It includes all the steps connected to the training phase of ML models. These steps range from the algorithm definition according to the type of problem to solve to the design of the training methodologies. As a consequence, the management of the generated artefacts and the process of parameter's fine-tuning are included in this category.

- Model evaluation: It refers to all the required steps to compute, compare and validate the ML model performance against arbitrary metrics.

Deployment

- Model versioning: It is crucial to ensure all the relevant model artefacts generated during the various training runs are recorded and traced in order to enable the use for inference purposes.
- Model retraining/update: A crucial capability is to enable planned re-training of already defined configurations in order to keep models up-to-date with the most recent data.
- Testing and scoring: It is essential to guarantee that given consistent data, the training and inference phases have expected behaviours and in accordance with each other. In general, the reliability of ML model predictions has to be ensured across all the environments in which a specific model is adopted.
- Benchmark and Performance monitoring: Assessing and tracking the endpoint's performance ensures to detect issues related to the availability and performance of the service. To achieve so, best practice is to generate and monitor engineering metrics for training jobs and inference endpoints executions and from those set up alarms for errors and produce benchmark reports.
- Prediction quality monitoring: Besides inference performances, the model quality through time requires monitoring solutions as well. The challenge is to track the evolution of model quality based on new data obtained at the inference phase. With these results, it is possible to diagnose when a model has degraded or outdated prediction capabilities.

These practices translate into software tools which express the system logic and behaviour.

2.2 Challenges

The development of production-ready machine learning applications is a challenging process. Several reasons determine most of the failures of ML initiatives. The major ones are the lack of standardized practices and tools to efficiently track experiments and training runs, ensure reproducible results, deploy trained models at scale and guarantee the maintenance of these models to ensure the delivery of value through time [15].

To understand the challenges, first, it is important to understand how ML models are currently being developed in the industry. Figure 2.2 shows

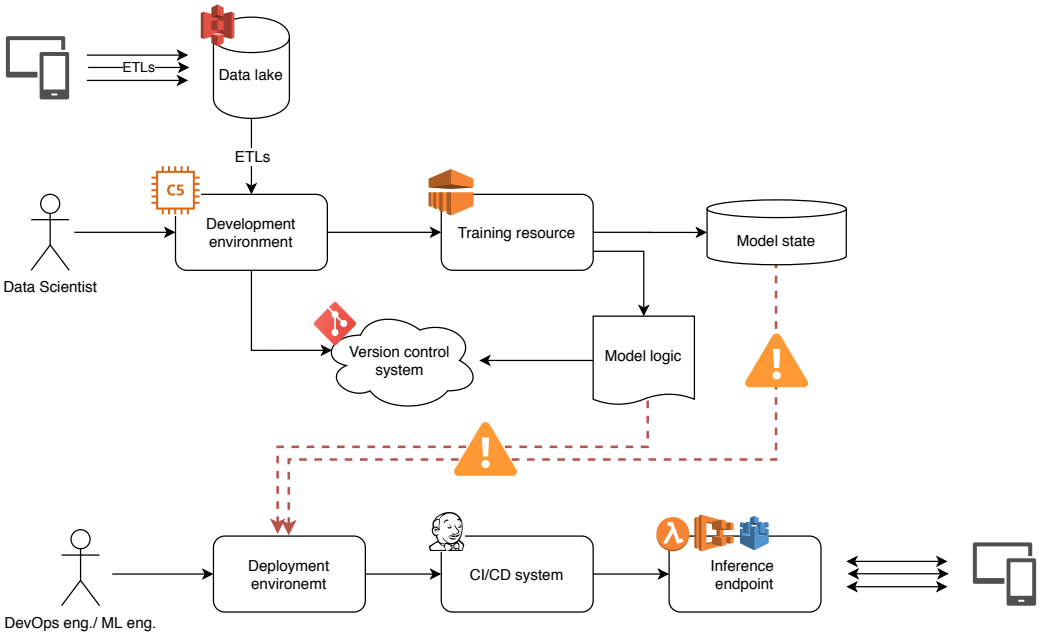

Figure 2.2: Common issues of machine learning pipelines

the way in which the core steps are defined in most traditional big data and machine learning pipelines. This pattern is already adopted in F-Secure and is extremely common among data-driven companies [2, 8, 36]. The workflow starts from the experimentations of data scientists in a development environment connected to the available data sources. From there the scientist sets up a training job which produces a new model. From there, the model has to transition to the DevOps engineer who is in charge of the creation of a real-time serving endpoint for inference purposes.

However, this workflow is far from being perfect. In fact, there are limitations and barriers which lead to significant stalls in the deployment process and in the worst-case scenario failures of machine learning initiatives. In particular, Figure 2.2 highlights three core weaknesses of the aforementioned workflow:

- Problems with the storage of artefacts. Once the training pipeline finishes its execution, artefacts, including the ML model, are generated and serialized for future use. These artefacts' size in memory can range between kilobytes to gigabytes depending on several parameters including the configuration, the size of training data and the algorithm in use. This means that it is likely that traditional code versioning solutions are not suitable to keep track of these objects and alternative solutions have to be designed.

- Problems with artefact de-serialization at inference phase. Serialization methods do not serialize complex classes in the same way they do with the native types such as strings, lists or dictionaries. Indeed, advanced instances are composed of a state and a logic of steps of execution. Hence, only the state can be tracked through serialization. Numerous failures at inference time occur when a pipeline attempts to de-serialize an artefact without its corresponding training code logic.

- Problems with correct dependency management and integration. As the ML domain introduces a separation of tasks which is new to the software engineering and DevOps domains, new challenges on the reproducibility and consistency arise. In particular, oftentimes there are major differences between training and inference environments. This is especially critical when the inference environment corresponds to real-time systems distributing predictions at scale. In particular, the main differences between the two environments are the compute architecture and the dependency requirements needed to perform efficient predictions in terms of performance and costs.

2.3 Function-as-a-Service as a serving solution

In November 2014 AWS released Lambda, the first serverless compute service in the market. The design of AWS Lambda defines an event-driven managed service. The application is implemented as a function connected to event sources and the latter trigger its invocation. With this, numerous advantages are introduced [45]. In particular, in the machine learning operation domain, reasons for which Function-as-a-Service (FaaS) technologies would provide as serving solution are:

- Pay-per-execution which derives that no cost is charged for idle time. This is particularly convenient when the traffic is highly influenced by business hours.

- Faster auto-scaling compared to container solutions and virtual machines. For the same reasons of the first, being able to scale on demand is essential to cope with variable rates of simultaneous requests.

Therefore, in the AI domain, when the limitations set by FaaS services do not interfere with project requirements, ML models can be deployed in a scalable and cost-effectively fashion as serverless functions.

2.4 MLOps in the cyber-security domain

Machine learning operations play a business-critical role in companies acting or in close contact with the security and, in particular, in cyber-security domains. With the increasing use of AI-based solutions serving predictions and recommendations for an extensive set of use-cases [20, 29, 46], the attack surface of malevolent actors has widened. As a consequence, threats and attacks directed towards machine learning models have started to become exceptionally common [5, 19]. On the other hand, with an increasing number of models deployed, internal disruptions may occur and lead to similar undesired behaviours when these are not detected and immediate action is taken to patch them [21]. These two types of events are not tolerated in companies of this domain, where trust is the primary quality to build and protect [25]. Therefore, this determines peculiar characteristics which have to be encapsulated in all the internal processes to mitigate the risk of current and future threats.

Furthermore, anomalous behaviours and threats can manifest in a highly diverse set of patterns especially in the machine learning domain. This derives that being able to recollect all the pieces of information needed to reconstruct actions and causes from the past is crucial to foster trust under every condition. Consequently, tracking and tracing of behaviours of a system are critical especially for those which run in a partially autonomous fashion and machine learning operations initiatives have to ensure that the highest quality standards on these capabilities are continuously delivered by the services. Moreover, based on these, active monitoring enables rapid response time which is essential to prevent dangerous situations to happen.

2.5 Related work

The machine learning operation tooling landscape has been increasingly growing in the number of open-sourced projects and their level of maturity. They try to tackle various aspects in the process of productionization of ML models by expanding already existing libraries or with new tools to enhance the quality and performance of processes or make them more insightful. In the next sections, the major open-source frameworks are described.

2.5.1 MLflow

MLflow is an open-source project by Databricks. The creators define it as "an open-source platform to manage the ML lifecycle" [50]. It consists of multi-

ple modules which address experimentation, reproducibility and deployment of ML models.

The major model in MLflow is **MLflow Tracking**. It represents the core of the MLflow platform and its main focus is to provide tracking capabilities to projects by recording parameters, metrics and artefact such as models, images, documents, generated at runtime. This information is stored in a centralized service where the scientist can analyze the results of his experiments over multiple code executions.

This component enables enhanced and simplified supervision and comparison of machine learning models. It reduces the time to set up standard logging and visualization capabilities and support team collaboration natively if running in a shared centralized server. The code overhead introduced by MLflow is limited since it does not need any advanced MLflow logic to be defined.

MLflow's tracking capabilities can be deployed on both on-premises and cloud environments and the current state of the project leverages cloud relational database systems and object storage services including Amazon Relational Database Service (RDS) and Amazon Simple Storage Service (S3). Figure 2.3 shows a possible configuration of an MLflow server running on an Elastic Compute Cloud (EC2) instance sharing the same virtual private cloud with the training runtimes in Elastic Map-Reduce (EMR) clusters. The EMR clusters run MLflow clients and report the desired metrics and artefacts to the MLflow server. In the proposed case the server is leveraging the managed RDS and S3 solutions by Amazon.

The MLflow platform offers additional modules which expand the tracking capabilities with deployment and project management. These are MLflow Models, MLflow Project and MLflow Registry.

MLflow Models supports the serialization of models created with the use of several external libraries for machine learning and data mining including Scikit-learn [40], Tensorflow [1], Pytorch [39] and Spark [51]. Additionally, custom models can be serialized using the MLflow's `pyfunc` serialization solution. In this use case dependencies are tracked in a YAML file with a well-defined structure.

MLflow Projects defines a configuration structure as a file from which it is possible to reproduce consistently the runtime environment either in a physical/virtual machine or containers. Also, the Projects component includes APIs and command-line interfaces for executing projects. With the support of external tools it becomes possible to convert projects into workflows.

MLflow Model Registry builds a model management system on top

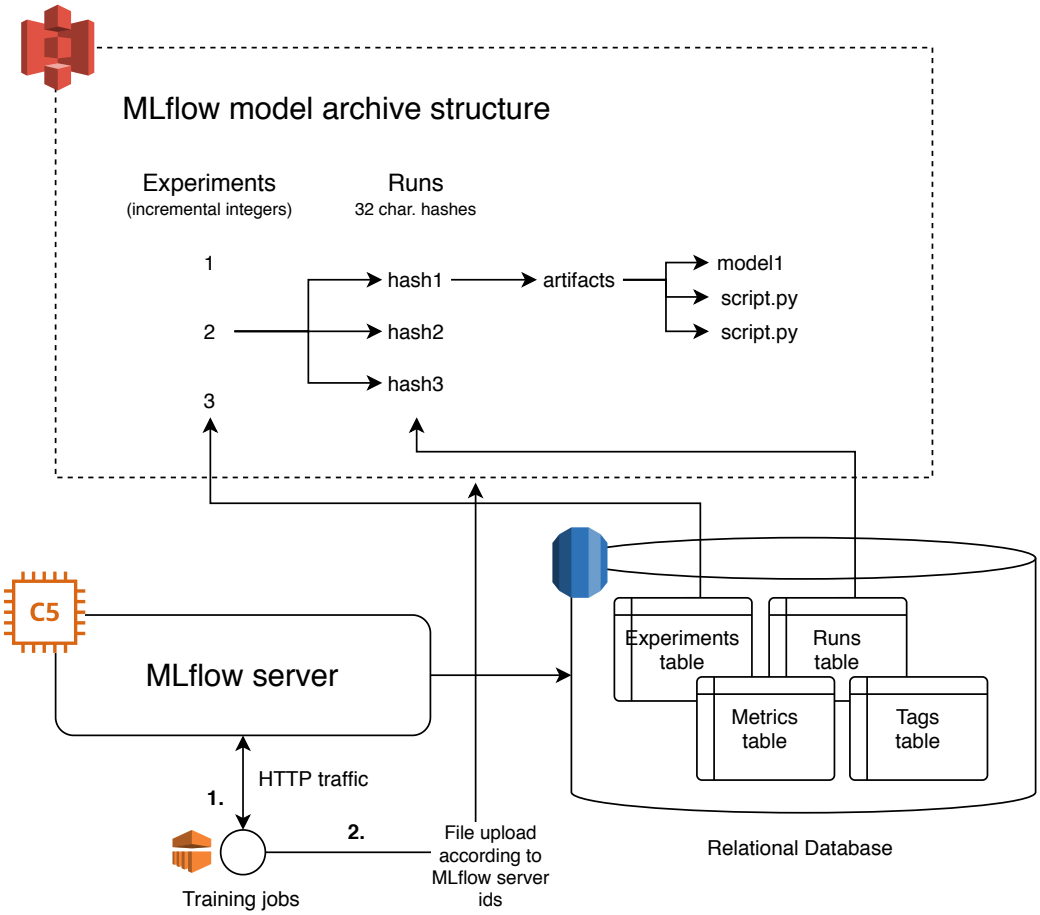

Figure 2.3: MLflow infrastructure schema

of the Tracking and Models modules. The Registry expands the tagging system by introducing a model versioning solution based on three stages of maturity linked to specific executions. The architecture of the model registry solution includes a centralized server (MLflow server) which communicates with the client and defines the path under which the artefacts of a given project are stored. In particular, the server assigns a unique identifier and all the information of the given run is stored under that specific path.

MLflow's solution for model serving offers the capability of deploying models available in the Model Registry component. This solution is relying on multiple dependencies including the `conda` package manager [14], the MLflow library itself and all of their dependencies.

2.5.2 Data Version Control

Data Version Control (DVC), as the name suggests, is the major solution for versioning of large files including datasets and machine learning models of any kind [24]. The DVC functionalities are built on top of the Git version control system and expand its native capabilities by solving the main limitation Git has, which is the versioning of voluminous files.

DVC integrates directly with various object storage solutions provided by the largest cloud vendors, including AWS S3. Data versioning is enabled by replacing those large files which one wants to version with small human-readable pointer files which can be easily handled by with Git. These placeholders point to the original version of the data stored in the cloud. This enables the decoupling of large files from source code management.

Besides, this approach opens new applications of the concept. In particular, data can be branched out together with the source code and thus experiments carried out in separate Git branches can be reproduced by applying the desired commit state which generated the specific experiment. Ultimately, DVC extends the concept of versioning metadata of large files to version metadata of experiments as well. Hence, the framework offers also basic experiment tracking capabilities without the need for a centralized server to manage the tracking logic.

2.5.3 BentoML

BentoML is an open-source framework focused on the productisation of machine learning models at scale [17]. A pipeline developed in line with the BentoML framework can be trained and deployed at ease. The solution supports both model serving behind a REST inference endpoint for real-time predictions or batch prediction inference. In particular, one of the strengths

of the framework is the compatibility with multiple real-time serving channels including Function-as-a-service solutions from the main vendors, Kubernetes and other open-source platforms and directly through managed container orchestration services.

One additional feature BentoML offers is automated micro-batching for increased performance for API utilization. This addition implements a middleman server with the task of collecting incoming requests before these reach the endpoint. The server stacks multiple nearly simultaneous requests to perform micro-batch inference against the model to reduce the computational overhead created when predicting each request separately.

2.5.4 Metaflow

Metaflow is an open-source project by Netflix. It is a framework for creating and executing data science workflows in local environments and scaling them to the cloud at ease [37]. For this reason, Metaflow is fully integrated with the AWS environment and the tight integration with it leverages the automatic management of computing resources and native parallelization of execution steps of services such as AWS Batch. The library implements a directed acyclic graph approach to schedule the execution of each of the steps of a pipeline. Additionally, the framework tracks and logs to a database instance all the output variables for each step. This enables more insightful debugging of pipelines in case of failures.

The main use case Metaflow aims to support is the batch processing one for any data transformation tasks, including batch model prediction. In particular, the graph structure not only enables better parallelization among tasks but also allows the reference and interconnection with tasks belonging to external pipelines.

2.5.5 Sacred

From the official Sacred documentation, this tool aims to help the user "to configure, organize, log and reproduce experiments" [30]. It has been created in the academic environment, in particular by Dalle Molle Institute for Artificial Intelligence, and aims to support researchers with all the overhead effort spent on experiment management which stacks on top of the pure research tasks. In particular, the framework aims to keep track of parameters and metrics of experiments similarly as MLflow does, by leveraging a MongoDB database instance to centrally store the different values and metadata about experiments. In addition, it offers capabilities focused on simplifying

the job for academic such as a solution for straightforward executions under different hyper-parameters and automatic seeding capabilities to support reproducibility of the results.

2.5.6 ClearML

ClearML is a service on top of existing projects to provide semi-automated tracking and tracing of runs [7]. In particular, the solution aims to cover all the most widespread tools in the Data Science domain with auditing functionalities to capture all forms of metadata, parameters and metrics generated during the various experiments. In details, the framework already support automated capturing of version control system metadata, Python environment information, standard output and error streams, resource utilization and sensor information, artefacts and both custom and internal hyper-parameters and metrics from the major machine learning and data mining frameworks.

The coordination of the service is carried out by a separate central web server leveraging a MongoDB database, an ElasticSearch deployment and object storage solutions offered by the cloud vendor for artefact archiving.

The model archive logic introduced by the ClearML framework is resembling MLflow's architecture of Figure 2.3. The only difference is introduced by the fact that ClearML stores the model artifacts under the following schema: `root/<project name>/<run name>/models/`. Hence the run key is the run's name instead of a 32 characters unique identifier used by MLflow.

2.6 Feature analysis of state-of-the-art tools

As the schema in Figure 2.4 shows, the state-of-the-art open-source tools scenario is clustered. MLflow is the first and most extensive project which aims to combine all the different aspects into a single end-to-end platform for machine learning. The other tools described instead address specific problems and aim to provide a comprehensive solution to those quite often with opinionated design decisions. The three major types of solutions are hereby analyzed:

From the Model Tracking perspective, MLflow, ClearML and Sacred require a central authority to govern the ingestion and management of the various metadata and metrics. Table 2.1 shows that the chosen solution for each of the frameworks is represented by a web server and additional services such as relational and not-only SQL databases such as MongoDB and op-

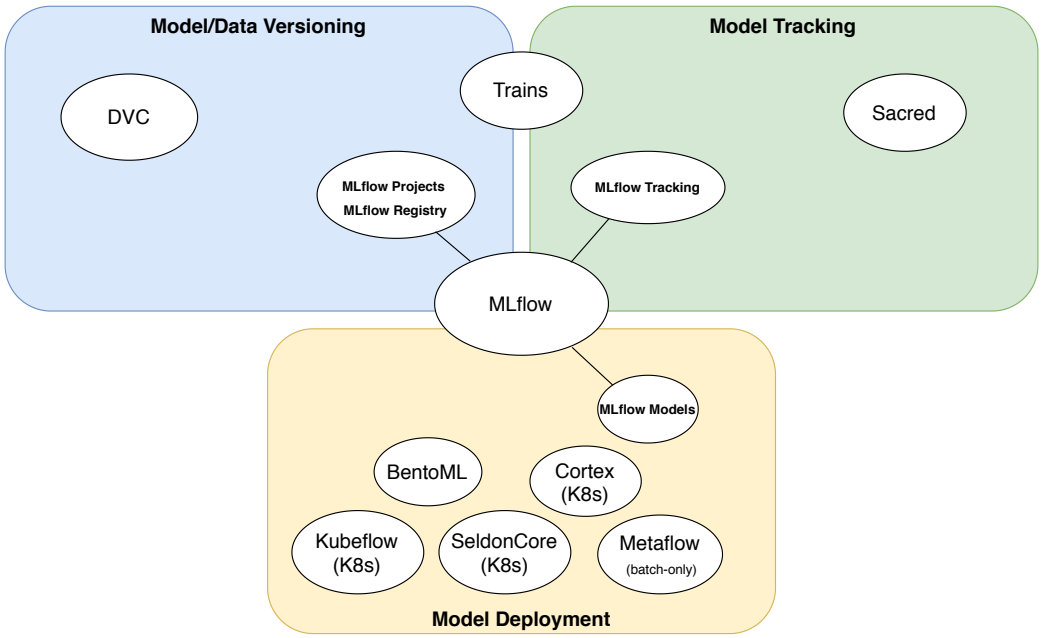

Figure 2.4: Domains of the state-of-the-art tools

tionally search engines. This approach becomes meaningful since it enables the serving of user-friendly web applications for intuitive displaying, filtering and analysis of experiments' results. Most configurations can be deployed both on-premise for single-user use or in cloud environments for enterprises. On the other hand, DVC integrates parameters and metrics tracking in its metadata stored with the Git version control system. Ultimately, Metaflow addresses the problem of tracking and logging by providing a solution which enables the inspection of the training runs after even after the termination of the run itself by tracking the steps with a checkpoint system.

From the Model/Data Versioning perspective, the research highlighted that managing versions of data and models is the aspect which is the hardest to generalize between different use cases. Because of this very few and different solutions are addressing the challenge. The only complete solution for versioning of large files is DVC, which extends git commands to enable tracking of large data stored in the cloud and consequently models of any scale. As Table 2.1 shows, the solution does not introduce any additional external server, however, highly relies on the integration with the version control system. This enhances reproducibility and sharing capabilities but opens challenges for the use in channels which do not naively support development tools interactions such as inference endpoints. The alternative

Framework	Model versioning	Model tracking	Model Deployments
MLflow	Web server, RDBMS, obj. storage	Web server, RDBMS	Local, Containers
DVC	Git integration, obj storage	Local, Git	—
BentoML	Local	—	Local, Containers, Serverless
Metaflow	Web server, RDBMS, obj. storage, ...	—	AWS Batch
Sacred	—	Web server, noSQL DB	—
ClearML	Web server, noSQL DB, obj. storage	Web server, noSQL DB, ElasticSearch	—

Table 2.1: Open-source framework technologies assessment

solutions for model versioning only are the integration offered by MLflow Registry and ClearML. Both implement the concept of model archive by defining a structure for the object storage service in which models are stored and model's paths (keys) are recorded in the database module connected to the web server. Both approaches link very tightly models to the execution which has generated them. This introduces major limitations in the accessibility of the web server becomes unavailable.

From the Model Deployment perspective, there are many tools to help the distribution of models at scale. Despite the ultimate goal being the same, each tool uses different deployment methodologies, therefore all have specific requirements and limitations concerning the use cases they are suitable for. In details, Metaflow's step-based pipelines are suitable for offline batch pipelines. Indeed, the overhead in storage and computations is not affordable by low latency task as real-time inference. As Table 2.1 shows, MLflow Models enables the serving of models in local mode or through container im-

ages. The first approach requires a full MLflow suite available in the hosting environment whereas the latter is more flexible and can be deployed to various services, such as AWS Sagemaker and Elastic Container Service (ECS), able to support containerized microservices. BentoML has the same offering as MLflow and in addition, supports serverless Function-as-a-Service deployments. Moreover, it adds additional performance improvements especially helpful for compute-heavy models such as deep neural networks.

Chapter 3

Requirements for the thesis

From the challenges highlighted in the previous sections, it is possible to define clear sub-problems which address specific aspects of the end-to-end lifecycle. These are set as requirements to build on top of and are used in the next sections to evaluate the capabilities of the state-of-the-art frameworks and the solution proposed in the thesis.

First, the need for a reliable solution for the storage of the state of trained models is a top priority to ensure robust pipelines. In particular, a robust model archive solution should facilitate the transition from training to arbitrary inference environments. Simultaneously, it should also deliver guaranteed access, hence stability and high availability are factors to consider when designing the solution. Moreover, proper governance of the models has to ensure that the history of model versions is preserved at some extent according to company's policies. This is particularly relevant for those companies like F-Secure, acting in the domain of cyber-security for the reasons explored in Section 2.4.

Secondly, an essential component of for an MLOps solution is the capability of driving the flow from the training phase to the deployment and serving phase consistently and reliably. In fact, tools which are not well integrated among each other and with the environment they act upon lead to roadblocks in the processes, thus often introduced delays and instabilities in the delivered functionalities. Therefore there is the clear requirement for a solution which addresses the end-to-end flow of deployment of machine learning models. In particular, the solution should be able to provide all the functionalities needed for the training pipelines, such as experiment tracking and model versioning, and coordinate these with the inference environment to guarantee a smooth and effortless transition of the trained model from one

to the other environment.

Once a reliable end-to-end flow has been established, it is considered that the organization has already achieved a strong level of automation in its operational processes. However, this does not fully address all the problems which arise when machine learning algorithms are in production. As the dynamics of the world change, so the underlying data does; as a consequence, models slowly lose their initial predictive quality which results in poor performance in the best case or to mistakes in the worst scenarios. Therefore the need for supervision is critical to prevent model drift to happen. To perform this at scale it is required to have automated tracing and tracking capabilities for both training pipelines and inference endpoints. The outputs of automated logging systems can then be used to monitor and detect problems before these can become harmful to the business value the models produce.

Subsequently, actions have to be taken to ensure the quality of machine learning algorithms to be delivered. In particular, trained models require updates to new versions. These may include new components in the logic or, in most cases, a newly updated model state which updates parameters and weights of the algorithm. In the case of models trained in an offline fashion, it means that the inference endpoint has to include functionalities to load and update to the latest version reliably and possibly without human intervention. This suggests that a non-essential yet extremely helpful requirement for successful machine learning operation is automated model update capabilities.

One additional challenge is the support of Function-as-a-Service and IoT devices for model inference. The peculiarity of these solutions is the limited storage resources or low speed on input/output operations from and to the disk volumes. Both these factors create bottlenecks when packaging the model in the endpoint. In contrast, these serving channels offer numerous benefits which make them the best solution for scalable and cost-efficient deployments. On one hand, FaaS deployments for machine learning models offer all the advantages described in Section 2.3. IoT devices instead move the inference capabilities at the edge. When the model permits it, it enables high cost savings and increased throughput.

In particular, the thesis focuses on the AWS own solution for FaaS, AWS Lambda. It has very strong constraints on the total size of the distributable archive to deploy. The up-to-date quotas are listed in Table 3.1. Therefore, given these limitations, an additional requirement would discuss the support for limited storage resources or low speed IO such as FaaS and edge compute

solutions.

Resource	Maximum quota
Compressed deployment package size	50 MB
Uncompressed deployment package size incl. layers	250 MB
Uncompressed package for console editor	3 MB
/tmp directory storage	512 MB
Volatile memory allocation	10 GB

Table 3.1: AWS Lambda quota restrictions [10, 11]

After the thesis design and implementation phases have concluded, AWS has introduced the support of Elastic File System (EFS) for lambda functions [42]. This implies that it is possible to include additional files and dependencies in external volumes attached to the Lambda function environment. Few months later, the support of Lambda deployments using container images up to 10GB has been introduced [41]. These two new features solve issue concerning the packaging of large files in FaaS deployments but introduce additional component to manage, secure and maintain. The thesis does not address these two options as they were introduced in a subsequent time to the design and implementation phases.

3.1 Requirements analysis

After having performed a thorough description of the features and value propositions of the different state-of-the-art open source frameworks currently available the analysis focuses on the evaluation of the compliance with the challenges and requirements specified in Section 3. The comparison has been split into two separate tables to achieve better clarity.

The comparison of the different solutions highlights that a vast majority of the requirements remain unsolved by most of the frameworks. In addition, there is no framework capable to stand out over the others in terms of features and capabilities. Only MLflow proposes real end-to-end capabilities and support for additional functionalities which try to cover all of the advanced requirements set. Addressing all the requirements defined in Section 3 and covering all the additional aspects which emerge in MLOps is non-trivial. Given the relatively short period in which MLOps initiatives have begun to grow, it is justifiable that established and mature solutions are not yet available in the market. On the other hand, this opens new research initiatives aiming to solve the yet not solved problems. Therefore, the thesis focuses

Framework	Model archive	End-to-end capabilities	Serverless/IoT support
MLflow	Yes	Yes	No
DVC	Yes	Training	No
BentoML	No	Serving	Yes
Metaflow	No	Train/Batch inference	No
Sacred	No	No	No
ClearML	Yes	No	No

Table 3.2: Requirements evaluation - 1

Framework	Automated model update capabilities	Automated model tracking and tracing
MLflow	No	Basic
DVC	No	No
BentoML	No	Basic
Metaflow	No	Batch only
Sacred	No	No
ClearML	No	Train only

Table 3.3: Requirements evaluation - 2

on solving the open challenges in model versioning, model deployment and maintenance. In particular, the main objective is to support the end-to-end machine learning lifecycle with automation of deployments with a server-free model archive and automation for model update and monitoring capabilities. Figure 3.1 shows the value proposition and positioning of the proposed framework. It can be compared with the open source state-of-the-art tools currently available (Figure 2.4).

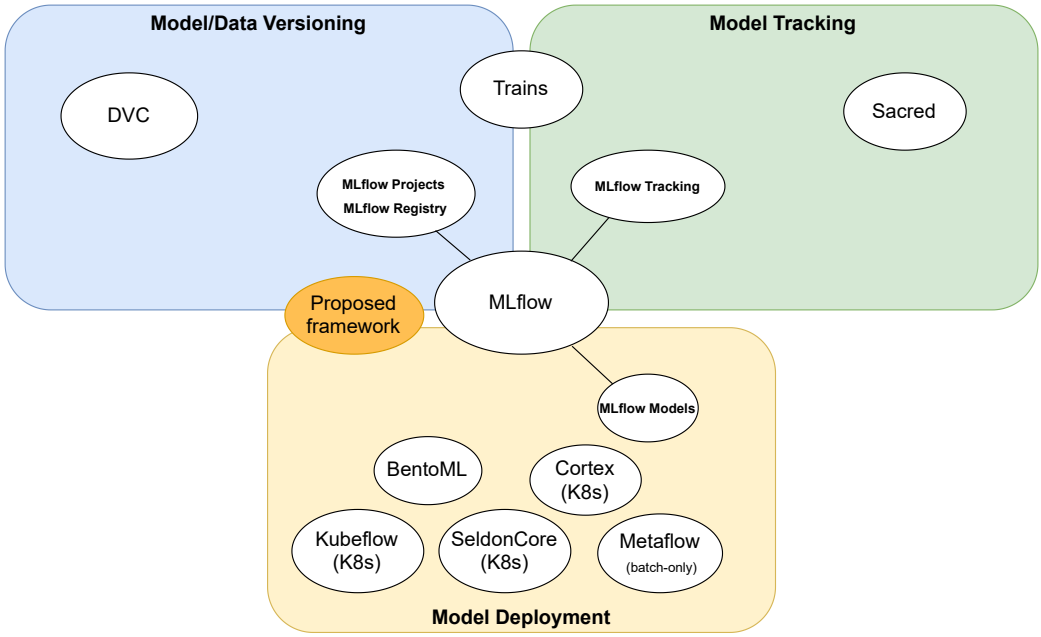

Figure 3.1: Positioning of the proposed framework compared to the state-of-the-art tools

Chapter 4

Architecture and Environment

4.1 Overall Architecture

The environment under which the thesis research has been carried out is hereby described. F-Secure operates in the cyber-security domain hence the architecture currently in place for machine learning pipelines is meant to serve the company's core protection engine. The firm provides access to its cloud resources to leverage the latest technologies in the Amazon Web Services environment, including S3, Lambdas and EMR clusters.

As part of the building blocks of the internal architecture it is possible to distinguish two main ML components, namely the training and inference environments. These compose the platform for end-to-end machine learning tasks. Moreover, both of them interact with a third environment, namely the aforementioned Cloud Protection solution, which is the F-Secure's in-house cloud malware rule-based analyzer.

Figure 4.1 shows a high-level overview of the two ML-focused components and how they interact with one another and with the Cloud Protection. As the description goes more in-depth with the description of the different services, the novel contributions of the thesis are highlighted.

4.2 Cloud Protection

As it has been already mentioned, the Cloud Protection engine is the core service for malware analysis. Moreover, it represents the main source of data for the various machine learning initiatives developed internally in the AI unit. The engine consists of different components among which there is:

- The metadata store: It is the data storage system in the Cloud Pro-

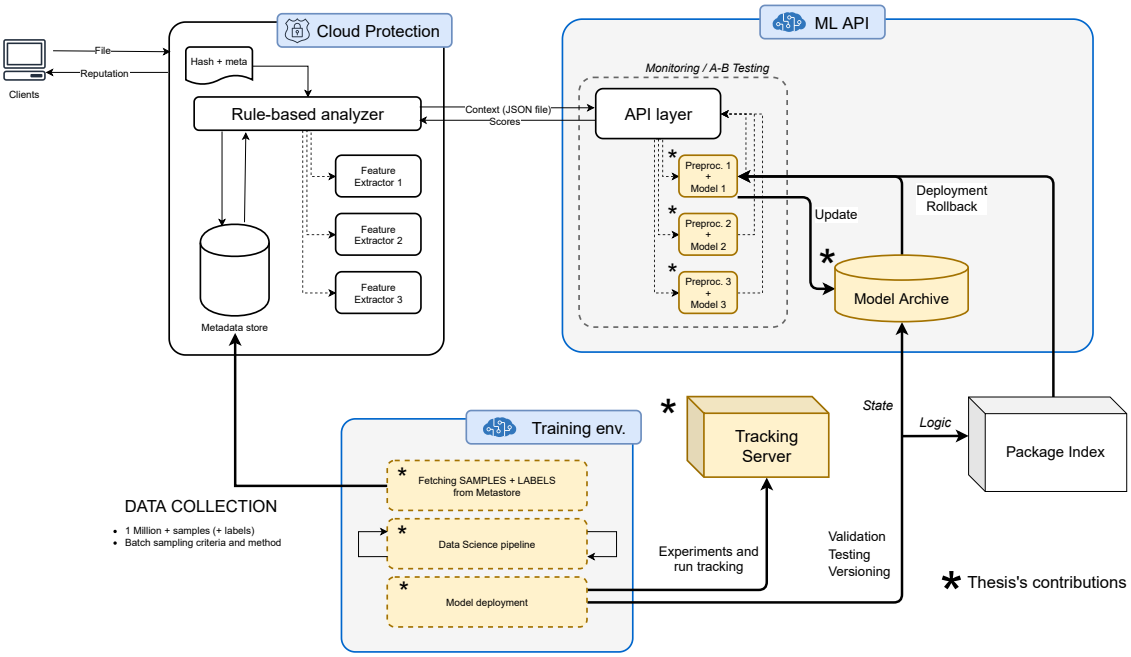

Figure 4.1: F-Secure architecture and thesis contributions to it

tection. It stores metadata about known threats and malware. It represents the data source from which data for the Mining of malicious events problem is extracted.

- The rule-based analyzer: It is the logic that decides for each sample fed to the Cloud Protection system. In particular, it queries in real-time the inference endpoints to request predictions from the various models.

The thesis does not address any of the components of this engine. However, it is useful to have a general understanding of the big picture of the data and where it comes from.

4.3 Training environment

The training environment consists of a pipeline orchestrator which schedules and monitors the execution of training and extract-transform-load (ETL) tasks. This component, therefore, is in charge to request the allocation of computing resources such as Elastic Map-Reduce clusters able to run machine learning jobs with the support of Apache Spark [51]. Permissions are granted so that the cluster is able to access the various data sources and data sinks with reading, write or both privileges.

Consequently, each machine learning pipeline is directly linked to a cluster during each execution of it. The cluster is indeed dynamically instantiated on-demand automatically by the orchestrator. The latter is a centralized service which has the role of managing the cluster instances and their execution. It abstracts the steps of allocation of resources and consecutively proceeds with the execution of the jobs. The cluster configuration and requirements are defined according to the infrastructure-as-code (IaC) paradigm as part of the code of each project whereas executions are launched by event triggers or according to pre-defined schedules.

The thesis contribution marked with an asterisk in Figure 4.1 refers to the software abstraction which manages the logic in the steps run during the execution of model training processes. Therefore, the contribution in the training environment is leveraging the hardware components and does not influence those.

4.4 Inference environment (ML API)

With reference to Figure 4.1, a second environment is focused on the serving of machine learning models for real-time inference through HTTP protocol. Given an increasingly growing set of machine learning endpoints, the environment implements a single point of access as an application-level Elastic Load Balancer (ELB) from which external users' traffic is directed towards the desired endpoint. This defines a common logic for the first layer of authentication and high-level routing to the different services available. In addition to the ML API environment, a private package index is accessible company-wide to serve and distribute libraries and packages of different programming languages in a centralized manner. This component is accessible through the HTTP protocol.

Given this configuration, the thesis implements endpoints with AWS FaaS solution, Lambda, for serving machine learning models behind the application level ELB service. In addition, two additional services, namely an experiment tracking server and a model versioning archive solution, are implemented as part of the thesis's architectural contribution.

Chapter 5

Methods

Security-related policies aim to ensure that governance principles are observed and access control, tracking and tracing are enforced correctly under the least privileges principle. This derives that models have to introduce additional complexity to enable more verbose logging and detailed error handling. Secondly, inference endpoints have to be carefully designed to enable granular authorization to the access to the resources and auditing the behaviour of the system and the actions taken and responses given by it.

Therefore, the proposed framework aims to provide automation, tracking, governance and maintainability to complex end-to-end machine learning pipelines at scale. The chosen working environment is Python since it supports a large set of libraries optimized for data manipulation, such as Spark [51], Numpy [49] and Pandas [32], and machine learning applications such as Scikit-learn [40] and Pytorch [39]. The framework will be evaluated against two different use-cases, presented in Section 5.1, which focus on different crucial aspects of complex ML pipelines.

5.1 Use cases definition

The thesis focuses on two use case scenarios. They represent real problems which are common under different flavours and conditions in various fields including sentiment analysis, spam detection and recommendation systems. The solutions are implementing different machine learning approaches namely supervised and unsupervised learning. The main goal of the two problems is to generalize the broad range of machine learning approaches by extensively addressing all the aspects which characterize basic to advanced

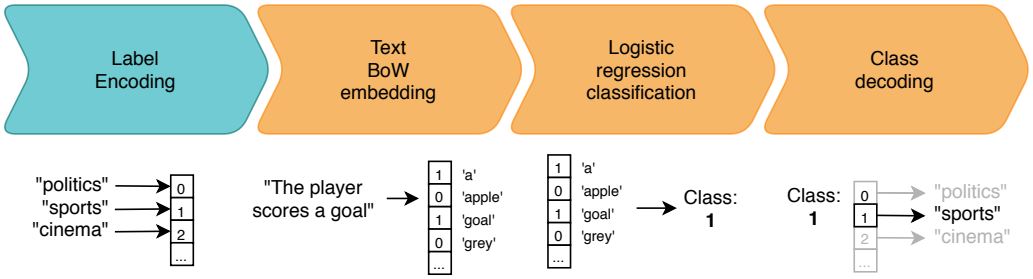

Figure 5.1: News articles classification steps. Blue steps are performed during training only, orange are applied both train and inference

machine learning pipelines in the industry. By doing so, these solutions become the test subjects to determine the degree of success for the proposed solution and consequently help to drive the design of the framework itself.

5.1.1 Classification of news articles

The classification of news articles is a well known supervised learning problem. Being a supervised problem, the dataset [31] consists of a collection of news articles with the corresponding news type label assigned to each of the records. Consequently, the task aims to assign to a given piece of text the appropriate type of news based on the content of the article.

The pipeline

The plain text form of each article is not suitable for the classification task as it is, thus a pre-processing is required. Labels are encoded into a numerical space through a bijective function. State of the art applications in the natural language processing (NLP) domain adopt smart tokenization and embedding techniques which exceed the requirements of this toy example. Therefore, the solution proposes a bag-of-words embedding with word-based tokenization.

The resulting vectors have a size equal to the cardinality of distinct words in the corpora. Each vector is classified by a logistic regression model to the predicted numerical class. The last step applies the inverse encoding function to obtain the nominal class from the predicted numerical value.

Figure 5.1 shows a schema of the different steps in the pipeline. The blue colouring of the *Label Encoding* step highlights that the step is executed at training time only and shared since the mapping has to be consistent between training and inference. The orange steps instead build the model.

Training process

The training process for the news articles classification problem consists of a train and validation approach. The full labelled dataset is split into two independent sets namely train set and validation set. Records are sampled randomly and assigned to the two sets with a distribution of 70% of samples belonging to the train set and the remaining 30% belonging to the validation set. Other supervised learning solutions implement an additional test used as final evaluation set. In the current scenario, it is not implemented since the engineering logic would not change and the goals of the problem does not include the optimization of the quantitative metrics of the solution.

The methodology of use of the two sets defines that the train set is used to fit the BoW embedding and the logistic regression algorithm whereas the validation set is fed to the trained model to perform the output predictions over unseen records. At this point metrics for the ground truth labels are calculated for both train and validation sets. These are useful for the data scientist to understand the objective quality of the model and empower him or automated software to make educated decisions when comparing multiple models' performances against the same sets.

5.1.2 Mining of malicious events

The mining of malicious events is both a machine learning and data mining problem. It belongs to the group of unsupervised learning problems. The main goal is to identify the most similar records in the data given a query element. Pairwise similarity scores are computed directly or indirectly between two sets of features each of which belongs to a record.

Therefore, in this problem, there is no mandatory constraint to have labels for data nor there is a label or value to predict. This aspect permits to have a more relaxed training pipeline since the current use case does not expect a validation set. Indeed, in this and other non-trivial unsupervised and supervised learning problems, the inspection of experts determines the quantitative evaluation of the approach.

The main goal of the problem is to uncover similarities between records of malicious events. Therefore, the solution has to deal with heavily customized training and inference pipelines from the data loading to the predictions. In particular, the feature engineering step introduces complexity in the model by using custom code not provided by traditional machine learning libraries.

Further, the pipeline presents non-trivial dependency management challenges which imply that smart handling of dependencies is required by the

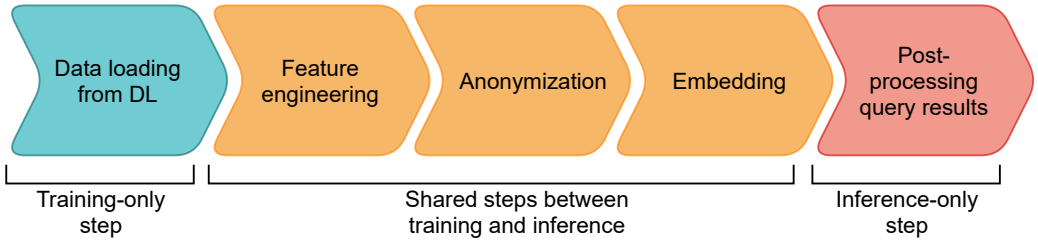

Figure 5.2: Mining of malicious events pipeline. Blue steps are training-only, orange are applied both train and inference, red are applied only at inference

framework to avoid extra costs and performance loss.

The pipeline

The full pipeline for the mining of malicious events is far more complex than the news classification solution. This is because the project is an F-Secure internal initiative with a broader vision than being only a framework's test example.

Figure 5.2 shows the major steps in the pipeline. The model consists of three main steps which are performed both at train and inference time. There is an additional step which is performed during the prediction phase only.

The *Feature Engineering* step is meant to clean, re-arrange and enhance the features of different nature and type present in each of the records. The *Anonymization* step ensures that all the data going to be part of the model itself in the embedding is anonymized therefore protected from attacks aiming to steal sensitive information of the customers. The *Embedding* step ultimately stores the data information in the embedding space to enable querying operations to fetch similarities between events. The last step is meant to query the embedding and perform post-processing data manipulations to adjust the output format according to the expected one on the inference endpoint.

Training process

The pipeline loads the compressed raw data from the data lake and batch reads the records. Therefore, the full set of events is stored in memory only after the *Feature Engineering* phase in which each entry is drastically reduced in size to contain only the relevant information. Consequently, the full dataset follows the *Anonymization* and *Embedding* steps and from this moment the model is suitable to be queried.

5.2 Model-as-a-Package paradigm

The thesis proposes the Model-as-a-Package (MaaP) method for distributing code dependencies. Consequently, a machine learning framework is built based on MaaP principles with the ultimate goal of performing scalable and reliable ML operations. The MaaP paradigm aims to address two of the most challenging aspects emerging from the process of productionization of models in the industry. These are dependency management and model versioning. As it has been highlighted in the requirements of Chapter 3, the two combined drive the degree of success on the transition of the models from the training environment to the inference endpoint. This is especially critical in the case of real-time predictions. Indeed, it is widely known that training and inference environment have different compute and management requirements which affect the transition process from the first environment to the latter.

The research first investigates which are the main building blocks which compose or impact any kind of machine learning model in the real world. In particular, the spectrum of research focuses of *trained* models since those are the ones embedding the knowledge required to perform predictions thus the ones which are possibly transitioning to the serving/inference environment.

Therefore, the core characteristics are, without any prevalence over the other:

The **code**: It represents the logic and the steps which are executed during both training and inference. The code can be defined by external libraries for the most simple use cases or tailored created by the data scientist or machine learning engineer for more advanced or customized applications. Source code can be packaged or compiled into a distributable release which is meant to be shared through repositories to provide high availability. Therefore, there are two main repository types widely used in the research and industry to guarantee backup and availability. The first and most common is the library repository. It hosts usually hundreds of libraries released by first-party or third-party sources and ensures availability of these libraries in the system. The service relies on a web server as the endpoint through which libraries are requested and served to the users. At the client-side, the traditional approach to fetch libraries uses a library or package management system utility which communicates directly with the distributor. On the other hand, library repositories do not provide efficient tracking of changes and versions in the code and enable team collaboration. Therefore, the industry adopts version control systems to address these two problems. However, as a compromise, these tools do not provide the same availability and ease of use of

library repository systems.

Code packaged and released in repositories requires a standard method of collecting all the necessary files and to define metadata and external dependencies in order to guarantee a successful execution of the code once it has been released and adopted by other users. This setup process introduces various features including versioning of the releases and dependency management which are not available by default for normal packages.

The **state**: It consists of the outcome configuration of weights and parameters after the finalization of the training process. The state of a machine learning model is considered static for the majority of the use cases, however, there are specific designs which open the possibility to inference and update models real-time, thus leading to a dynamic state through time. The thesis focuses on models with static state since the transition from a static to dynamic state, when possible, requires advanced setup strongly bounded to the characteristics of the model itself. Model states can be made of several different types, ranging from a simple unstructured set of values or large key-value dictionaries to complex objects generated by advanced serialization protocols. The most simple ones are also very flexible but less descriptive about requirements whereas the most complex ones provide better support for tracking non-conventional data structures or objects. In the machine learning domain, these configurations are often stored as a file and treated as data rather than code due to two main reasons: the exceptionally large size these objects can reach and the low demand. Therefore, the data lake is the major archive for model states and these can be easily distributed from it to endpoints, applications, edge devices.

The **data**: It corresponds with all the data fed to the model during the training process. Various aspects of data (volume, veracity, value) affect in multiple ways the quality and performance of the model. Therefore, although data per se is not a dependency which has to be shared from the training to the inference environment, it still largely determines the quantitative outcomes of any machine learning model.

Given these three main components, it is relevant to understand how they are traditionally stored and define suitable locations where to record the artefacts.

From these considerations, the goal for MaaP is to enable smooth integration of the state of a model with the respective code component, including custom code and external dependency libraries. Therefore, the idea is to leverage the distribution capabilities of the library repository. To achieve so, the Model as a Package paradigm proposes to treat a model as a highly

specialized library dependency connected to a specific state in a point in time. Consequently, this approach allows packing a specific version of the code in the same manner of standard software engineering principles. Testing and linting can be performed as part of standard CI/CD pipelines and the code can still be versioned in standard version control systems. Besides, one additional advantage which comes with the adoption of library repositories for the distribution of model code is the possibility to leverage package management systems to ensure consistent dependencies between the training environment and inference endpoints. Hence, MaaP defines clear practices to ensure correct external dependency management.

5.2.1 Dependency management in model's code

As it has been mentioned in Section 2.1.1, often major differences exist between the training environment and the inference endpoint. This in particular for real-time predictions where, for instance, a model can be trained on a cluster of machines running batch transformation jobs and deployed as a serverless application or on edge devices. Therefore, in these scenarios, correct dependency management is crucial to optimize performance and costs.

Dependencies may vary between training and inference environments. In particular, it is common that several dependencies necessary for the training pipeline are redundant in the inference environment. Hence, these are occupying vital space which impacts on deployments in serverless applications or edge device and additionally is bad software engineering practice. As an example, a training pipeline includes PySpark as a dependency to pre-process on the fly the raw training data and converts it into a Pandas data frame. In this case, PySpark becomes redundant if the inference endpoint consumes single events without any need to batch extract them from the data source. Consequently, external dependencies can be classified into three main independent categories depending on the context of use. With references to the steps in Figure 5.2, the following dependency types can be defined:

- Shared dependencies: these libraries are included both during training and inference. They are imported by the source code only in one or many of the modeling steps in the figure.
- Training-only dependencies: these libraries are included both during training only. They are imported by the source code only in the training steps in the figure.

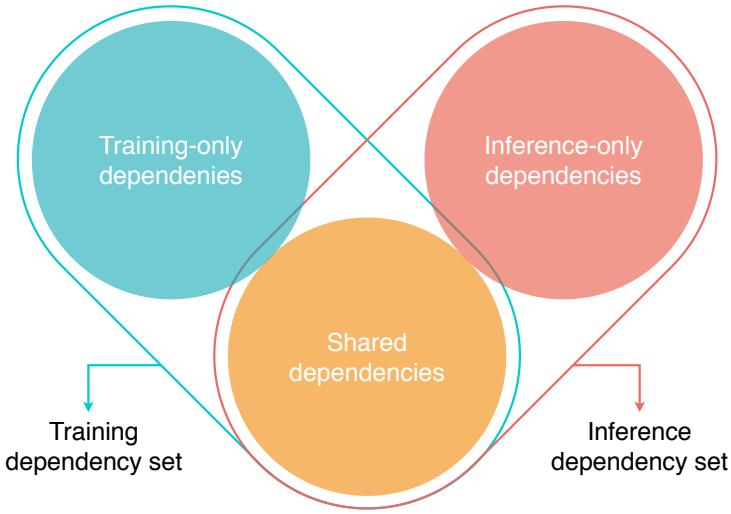

Figure 5.3: Dependency sets logic

- Inference only dependencies: these libraries are included both during training only. They are imported by the source code only in the inference steps in the figure. These are rarer since it is frequent to run predictions and validation of the output during the training phase.

Therefore, the set of shared dependencies is the bare minimum requirement which has to be installed in any environment to ensure the model can predict. This set has to be joined with either the training-only or the inference-only set to guarantee a correct execution in the training or inference environment respectively as shown in Figure 5.3.

To address this, MaaP proposes a selective dependency approach. The approach defines that training-only and inference-only dependency sets are assigned to different tags. Shared dependencies instead have to be tag agnostic, thus installed by default. This approach results to be intuitive in Python thanks to the `setuptools` library which supports tagged sets of dependencies.

However, the aforementioned dependency management approach is not sufficient to guarantee a successful execution of the training or inference pipeline. Indeed, it is necessary to perform clever isolation at the module level when importing dependencies in the pipeline. The goal is to prevent to import libraries which are not supposed to be loaded -and installed- in a specific environment. This aspect is addressed by the design of the framework.

5.3 Framework design

The outcome of the thesis is a machine learning framework for scalable ML operations in F-Secure. The framework incorporates the principles of the Model-as-a-Package paradigm and expands tooling and functionalities to support additional aspects in the lifecycle of machine learning pipelines as a software abstraction to glue together the training and inference environments. In particular, the proposed design addresses problems of model versioning, model update and deployment automation of advanced machine learning models. Moreover, it introduces solutions for monitoring and analyzing experiments.

Therefore, in this context, the framework implements the concept of a pipeline in a pipeline. The idea behind it is to define a nested and streamlined pipeline inside the training step to better govern and monitor the entire training step by defining a set of soft constraints and requirements. These aim to help the engineer to define and plan the code and the functionalities in a better organized and structured manner without limiting the freedom of manipulation of the pipeline itself.

Components

The framework is composed of a model archive and two main software components namely the Trainer and Predictor. The latter two leverage the model archive and the library repository to work in coordination in end-to-end machine learning pipelines.

5.3.1 Server-free model archive

The model archive is the architectural component built to provide persistent storage capabilities to the framework. The latter leverages it to achieve persistent model versioning and artefact tracking for multiple projects. Unlike other alternative solutions, such as the MLflow's model registry and ClearML model repository, the proposed solution for the model archive is not dependant on additional services such as web services and database runtimes.

Being the model archive the crucial component for archiving and versioning all the artefact linked to ML models, this service has to provide the following features:

- Long term storage, to guarantee historical versioning and rollback capabilities.

- Generic object storage capabilities, to allow to store any type of artefact without any schema validation constraints on the input itself.
- Support for large objects, to guarantee that models of any size can be versioned with no constraints.
- Limited cost.

	Long term storage	Generic objects	Large objects (>100MB)	Cost (main factor)
Relational	Yes	Yes	Yes (<4GB)	High (instance cost)
Key-value	Yes	Yes	No	High (instance cost)
In memory	No	Yes	No	High (instance cost)
Document	Yes	Yes	Yes (GridFS)	High (instance cost)
Column wide	Yes	Yes	No	High (instance cost)
Graph	Yes	-	-	High (instance cost)
Object storage	Yes	Yes	Yes	Low (storage cost)

Table 5.1: Comparison of data storage solutions for model versioning

The main drivers for the decision for the appropriate storage solution are the cost and the capability of storing single binary files. From this, Table 5.1 compares different categories of data storage solutions ranging from SQL to No-SQL and object storage services. Therefore the chosen solution for the model archive is the object storage as part of the data lake; F-Secure has chosen to adopt AWS, thus Amazon Simple Storage Service (S3) is the corresponding service. Being the object storage service structured as a key-value store, it is necessary to ensure each model is uniquely identifiable and referenced in a predictable manner according to a schema.

Model versioning schema

The design of the schema for the model archive defines the versioning capabilities of the entire framework. Therefore the proposed schema aims to comprehensively address the following framework requirements:

R1 Support for multiple projects: The schema has to be capable to distinguish and separate the storage of artefacts belonging to distinct projects, thus to ensure correct model versioning for multiple projects without one interfering with others.

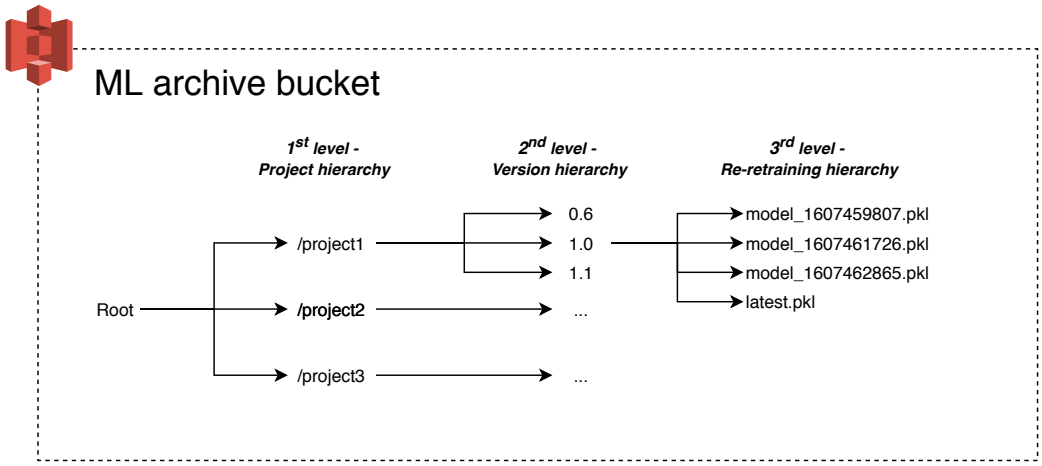

Figure 5.4: Model archive key schema

R2 Multiple project versions: With the improvement of a project code and requirements may diverge between different versions. Therefore, the schema has to separate model artefacts whose project/code versions are not compatible with each other.

R3 Scheduled re-training versions: The schema has to ensure that training sessions of the same version in different moments in time store separate models for rollback while guaranteeing always a model served through the inference endpoint.

Furthermore, F-Secure has chosen Semantic Versioning [43] as the formal convention to standardize the process of definition of project versions, thus the following discussion elaborates on this approach without excluding possible alternative conventions.

The proposed structure for the versioning system implements three levels of hierarchy from the root of the model archive. The first level defines the discrimination between projects. The second level of nesting discriminates among different versions. Ultimately, in the third level, each unique model is uniquely identifiable through an id. Moreover, the most recently trained model is duplicated to a static key (`latest.pkl`" in Figure 5.4) to make it reliably available to the inference endpoint.

Hence, the key for each model is generated according to the following rules:

- A unique first-level prefix is assigned to each distinct project. An example can be the project name. This solves requirement R1.

- A per-project unique version second-level prefix is assigned to different version releases of each particular project. Additionally, with the adoption of Semantic Versioning, the proposed schema truncates the version to the sole `major.minor` pattern. This reduces the granularity of distinct version updates with the assumption that the `patch` tag is irrelevant to the model structure since it is only meant to patch bugs, thus not change the overall structure of the model itself. This solves requirement R2.

- Each model is saved under the format:

  `[first-level prefix]/[second-level prefix]/[filename]`

  The filename has to be unique in the context of a given project and version. Additionally, a static filename is introduced to enable a reliable reference to the latest model for each valid project-version combination. Therefore the filename format is defined as follow: `latest.pkl` for the static naming solution which references the latest update of the model. `[timestamp()].pkl`, where `timestamp()` is an integer representing the timestamp collected during a training execution, for each unique training run generating a new version of the model for a valid project-version combination. This solves requirement R3.

5.3.2 Training solution

The proposed solution for the training phase introduces a novel component named Trainer. The Trainer represents the interface through which the various machine learning pipelines are going to interact with external resources including the model versioning system and the tracking solution.

In the first place, the Trainer aims to define an increased level of governance over the training lifecycle by separating both from the high-level perspective and from the code one the components related to the project itself from the ones providing MLOps services and functionalities. This is achieved by introducing a common interface to abstract engineering practices from the data science project domain to the Trainer itself. This is done by introducing a set of values the component introduces:

- Standardize and automate repetitive and redundant tasks:

  One of the major deficiencies in machine learning workflows when the number of initiatives increases is the presence of repeated code for both

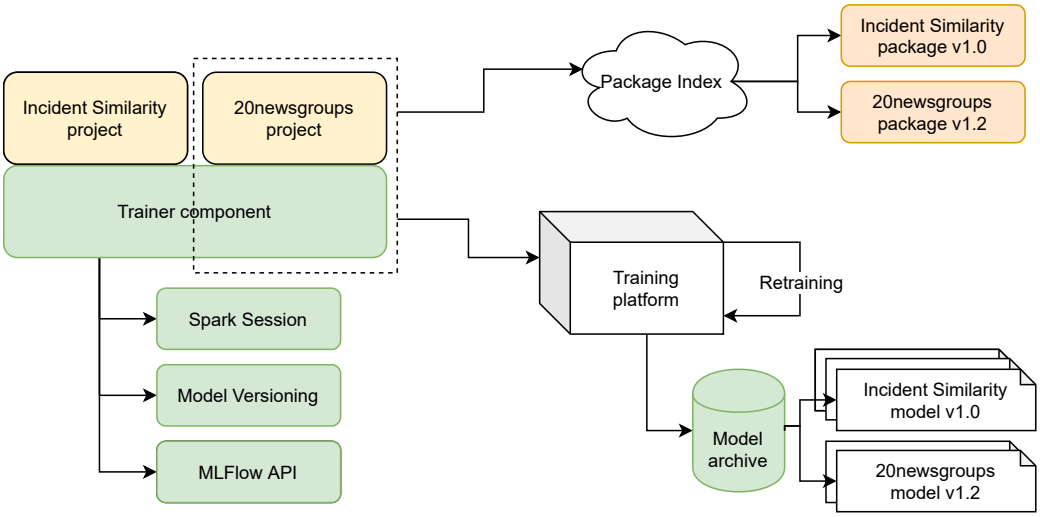

Figure 5.5: Trainer component schema

initialization and core functionalities of the pipelines such as the model versioning solution. This is a bad software engineering practice due to the intrinsically high cost of maintenance and the increased risk of failure. Additionally, in some particular cases such as model versioning, it is essential to guarantee a standardized interface to ensure the correct use of the service.

- Better separation of code components and functionalities in projects, modules and tests:

  As highlighted above, reducing the redundancy among projects defines better separation between major engineering functionalities and the specific features and methodologies specific for a certain machine learning pipeline. This implies that each data science initiative can focus on increase the code quality of the different specific model components and improve reliability and maintainability by creating more thorough tests for each of them.

- Enable improved model reproducibility and versioning with no efforts:

  The introduction of external logic for the management of the model's code and state through the concepts of MaaP and model versioning implies that the logic defined in the specific model training pipeline is agnostic to the management activities running under the hood.

- Increase the level of insight during the different phases (in particular during automated re-trainings):

In addition to the externalization of core engineering tasks from the single projects, the Trainer enables the incremental introduction of tools and features which are going to be automatically ready to use from the different projects based on their use case. In particular, the integration with tracking solutions is essential to simplify the offline monitoring process alongside with the increment the level of insights obtained from the single training session from the comparison of multiple runs across time.

- Increase prototyping iterations:

  The Trainer aims to reduce the time overhead from the idea validation and experimentation phases to the productionization of the solution. The definition of a common interface to comply with facilitates the transition from the semi-structured code present in notebooks and prototyping environments to robust code ready for deployment. This is possible thank to the a priori knowledge on the high-level expectations of the production environment.

- Full code flexibility:

  Multiple data sources and data types require different loading methodologies. Moreover, different models may have different pre-processing pipelines, set different expectations on the results and track different relevant metrics and parameters. These statements are especially valid in the cyber-security domain where volume variety and veracity of data may vary a lot from one problem to the other and as a consequence the resulting model as well. Therefore, the flexibility on the definition of the training procedures and resources each project is essential to guarantee the success of a machine learning initiative.

Being the Trainer the core of engineering of the framework, it aims to abstract features and functionalities from the projects to itself. As a consequence, there is a centralization of tasks which may lead to a cluttered framework design. To address this issue, the design proposes a modular architecture capable of introducing new functionalities as plugins. The foremost and native plugins are:

- *Spark support* which introduces the direct spark support to any project leveraging the Trainer architecture. The projects inherit a Spark session ready to use for data manipulation.

- *Model versioning* which introduces the capability of using the model archive to store and version the trained models in a reliable, consistent and robust manner.

- *MLflow tracking* which is in charge to initialize the necessary components, such as authentication and experiment name, to enable the tracking of parameters, metrics and artefacts with the external MLflow tracking service.

Each plugin is focused on either solving a specific aspect in the machine learning operation domain or expanding the range of features or use-cases suitable for the Trainer.

After having defined the design and behaviour of the Trainer with respect to plugins and external tools, the thesis focuses on describing the integration of a project with the framework in Section 5.4.2.

5.3.3 Serving solution

The Predictor component handles the model serving phase of the end-to-end pipeline. With references to Figure 2.2, after the training phase, two basic building blocks of the model, namely the *logic* and the *state* have to be made available and combined in the inference environment to be able to perform real-time predictions in the endpoint.

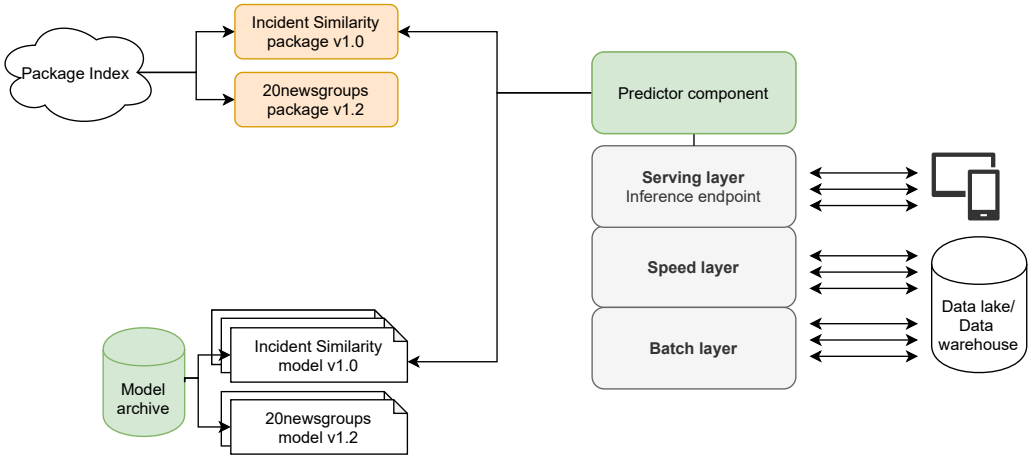

Figure 5.6: Predictor module schema

In this scenario, the Predictor component aims to combine the aforementioned model components in an automatic fashion such that the solution agnostic to the serving channel. Therefore, thanks to this capability a wide set of architectures, such as serverless applications, container deployments or edge devices, can leverage the Predictor component to serve the model without any computational overhead.

As Figure 5.6 shows, the Predictor component represents the software interface which connects the model package distribution and its corresponding model stored in the model archive. Therefore, this design enables the Predictor component to expand the standard model inference process with additional capabilities to enhance the level of automation and insight of the inference endpoints. In details, the Predictor aims to provide additional features to address requirements defined in Section 3. In particular, the solutions the Predictor provides focus on solving the automatic model update and automatic tracking and tracing capabilities. These two challenges are addressed by including preparation and termination steps during the process of prediction triggered by an input query from the users of the service.

The automatic model update capability design is designed to deploy an a priori tagged version of the model. Figure 5.7 shows the two main steps of the underlying logic. The initialization of the model endpoint invokes the first model load operation, it occurs only once and no queries are handled. The other scenario is when a query is received by the endpoint. The model expiration is checked, then optionally the model is updated if the check returns a positive result and finally the main sequence of actions can proceed with the prediction phase. Worth noticing that when the model update is performed, the variable is locked to avoid concurrency issues. The detailed sequence of operations is defined in Figure 5.11 of Section 5.4.3.

The automated logging capability includes a preliminary phase of logging for tracking purposes metadata about the user input when a new query operation is received from the endpoint. Moreover, a post-prediction phase is executed if the prediction concludes with successful results. At that point, model metadata included by the Trainer component are logged for tracing and additional prediction metadata are recorded for tracking purposes.

5.4 Framework implementation

5.4.1 Package distribution

In the specific case of the Python language, the packaging process is carried out by the `setuptools` package and requires the `setup.py` file in the root path of the package.

In the MaaP context, the packaging pipeline collects all the metadata information defined in the `setup.py` and the source code (modules and scripts). These are combined in a well-defined project structure which is consequently consolidated into an archive. This packaging system excludes from

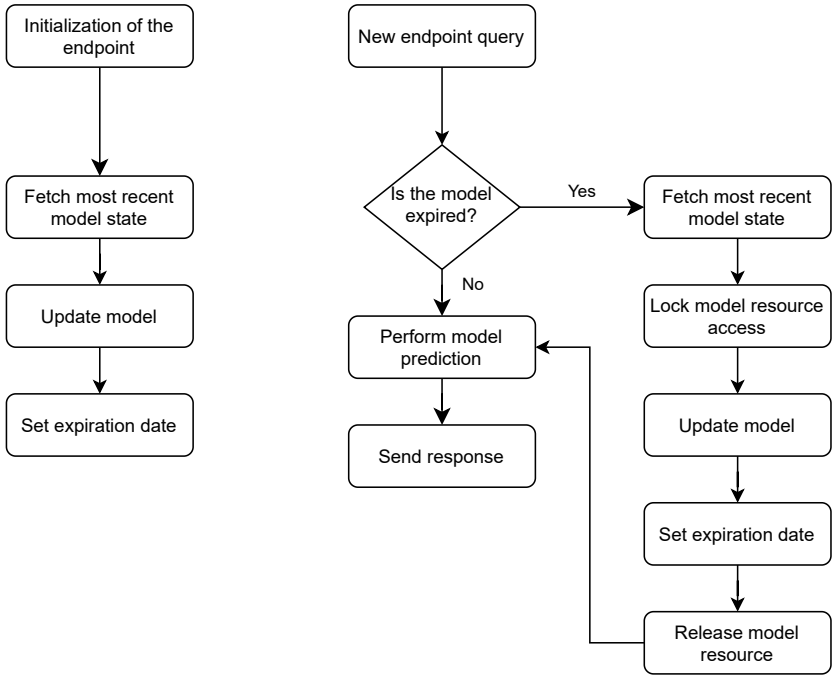

Figure 5.7: Automated model update logic

the archive all of the third-party dependencies required to run the project. These are instead included as references among the other metadata. From there the installation process will read them, fetch their source code from available sources and store it into the local environment. This action is performed recursively over all the different packages until all the necessary modules are available in the execution environment. Therefore, as Figure 5.8 shows, dependencies of a project can be grouped into n-levels which correspond to the minimum distance from the initial node of the graph (the project itself). Note that the graph is acyclic by definition since circular dependencies are not allowed.

5.4.2 Trainer implementation

The implementation of the Trainer component defines two main building blocks namely the architecture definition configuration files and the Trainer logic. They both contribute to the framework, however, while the logic is considered extensible to any environment running Python, the architecture configuration is strongly connected to the F-Secure's internal solution but can be replaced by other custom definitions. Given the scope of the thesis, the architecture configuration represents the set of components and steps

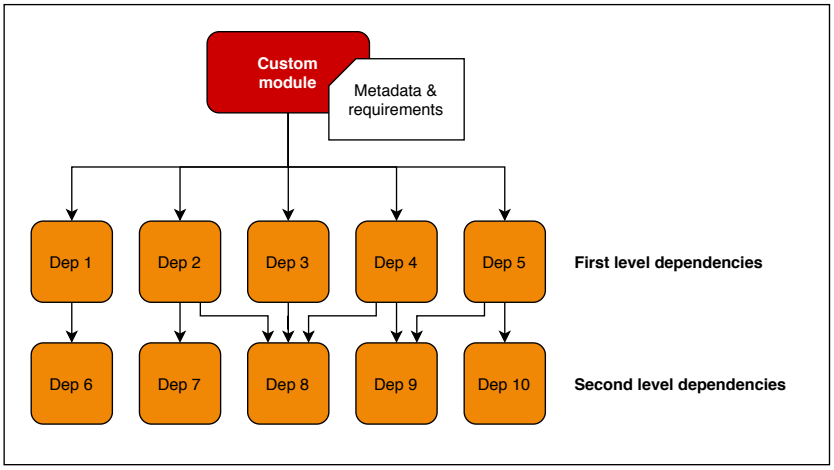

Figure 5.8: Code component members

necessary for the orchestrator to allocate and set up a cluster for a single training run.

In addition to these, a common CI/CD pipeline defines the deployment process. In details, standard software engineering assessments such as testing and linting, are performed at every new release. In addition to these, the automation server packages the code and the architecture configuration to the orchestrator and publishes the model-specific package to the package index in accordance with the Model-as-a-package principle (Figure 5.5). Once the project is deployed to the orchestrator there is the option to set a training schedule plan in line with the project-specific requirements.

The core logic of the framework for the training phase instead consists of a library providing a command-line interface (CLI) tool necessary to execute project pipelines running on top of the EMR cluster. In this way, the Trainer can initialize and configure all the necessary components necessary to train a model. Figure 5.5 shows the architecture serving the Trainer functionalities. Among all, there are:

- the Model archive as an S3 bucket. Its specifications are discussed in Section 5.3.1.
- the tracking server as an MLflow deployment in AWS Fargate, Amazon's serverless compute engine. The server is used only for tracking purposes of the training executions. The contribution of the MLflow tracking server to the thesis has to be considered only for the goal of reaching a complete end-to-end pipeline. This means that any other

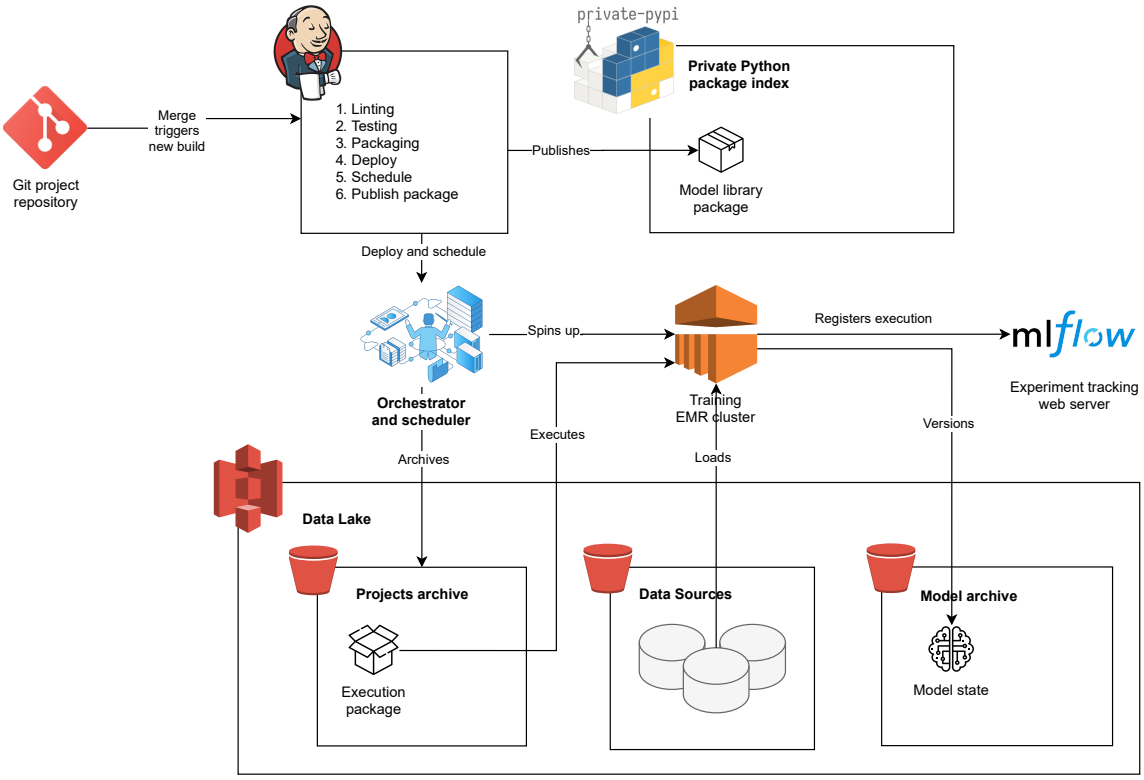

Figure 5.9: Trainer architecture implementation

solution can replace it for metric visualization. It is not the goal of the thesis to provide new data visualization solutions.

Moreover, Figure 5.9 highlights the process through which, once a new update to a machine learning project is applied, the CI/CD system publishes the model logic package to the package index in line with the Model-as-a-Package paradigm and triggers the execution of a new training job. Optionally, the CI/CD system schedules re-training job executions to update the model state.

As it has been mentioned before, the Trainer component implements a command-line tool which enables the execution of a given project's pipeline in EMR clusters or other technologies such as single virtual machines and local environments. The Trainer relies on the use of a per-project external configuration file as the main solution for package discovery and external parameter definitions. The file defines relevant core metadata about the project, such as the project name, and the entrypoint class in a YAML format. In details, the entrypoint class is the reference the Trainer uses to perform the training procedure. Hence, this class and its dependencies are loaded dynamically at

runtime. Moreover, based on this, the framework abstracts all the necessary configurations to be able to interact with external resources such as the EMR Spark environment, the tracking server and model archive. These extensions are built on top of the framework as a set of plugins.

The *Spark support* is provided by the Trainer component to projects as part of the core suite of tools for big data manipulation. EMR clusters indeed offer the Spark runtime as part of the default set of functionalities. As a consequence, the Trainer leverages it through the Python implementation `pyspark`. Therefore, during the initialization step, the framework automatically creates a new Spark session object which is shared with the project as an argument passed to the entrypoint class.

The *tracking server* plugin focuses on setting up a ready-to-use session with the external MLflow tracking server. In particular, the plugin is executed at the Trainer's initialization time so that all the essential configuration steps are performed and the session is ready to use by the project's logic. In details, the major steps are, in order: performing authentication to obtain valid credentials to access the tracking server, creating a new experiment associated with the specific project and starting a persistent run which lasts until the training session ends. In particular, the last step allows the project's entrypoint and sub-modules of it to log custom artefacts and parameters without the need to handle the verbosity of the configuration of the integration with the external service.

The *model versioning* extension is focused on defining standardized and shared logic to interact with the custom model archive solution. This plugin is hidden from the logic of the project since it can be considered as a support for the Trainer to provide correct model versioning. The capabilities offered by the model versioning plugin are: to generate a unique key for models complying with the standard specified by the model archive schema pattern, serializing the model as binary object and storing the serialized object into the model archive with the aforementioned key as identifier.

As Figure 5.10, the Trainer implements the aforementioned concept of pipeline in a pipeline. This implies that the framework enforces a soft schema for the entrypoint class to comply with. The schema defines a set of consecutive steps which are executed in sequential order and mirror the logic of the training pipeline. Each of them defines a specific task:

1. *Initialization* which imports and instantiates all the steps correspond-

ing to the different plugins, such as the *tracking server* and the *Spark support*, and project's pipeline.

2. *Training* which runs the training of the pipeline, from data import to the actual training of the model itself.

3. *Scoring* which executes the evaluation of the model, Moreover it expects metrics to be collected from the pipeline to log them through both the logging and tracking service.

4. *Collection* which expects to receive the trained model. This step is essential to share the trained model from the project to the Trainer component to perform the correct versioning of it in the model archive. Consequently the Trainer versions it through the *model versioning* extension.

At the end of the *Collection* step the model is submitted to a validation step to ensure it satisfies the minimum interface requirements required by the inference stage. In the case the validation ends with a positive result, the trained model is augmented by the Trainer with additional metadata related to the training execution. These are going to be essential in the inference phase to perform in-depth logging and monitoring of the prediction endpoint.

5.4.3 Predictor implementation

The inference architecture is dramatically different compared to the training one. As Figure 5.11 shows, the components shared between the training and inference environments are the model archive and the package index. Moreover, the role of the CI/CD system in this scenario is to package the project package with all the necessary dependencies (including the Predictor component) and, with this, deploy the inference endpoint, in this case as a serverless Lambda function.

The endpoint is consequently discovered by the Elastic application-level Load Balancer which forwards incoming connections to the endpoint after having authenticated the client. This component corresponds to the API layer of Figure 4.1.

Additionally, logs from the different endpoints are stored in a separate bucket. From there all the data is collected by a big-data analysis engine such as Splunk [34].

The Predictor component is a library dependency which can be used by any deployment type to perform predictions using models produced leveraging the training counterpart of the proposed machine learning framework.

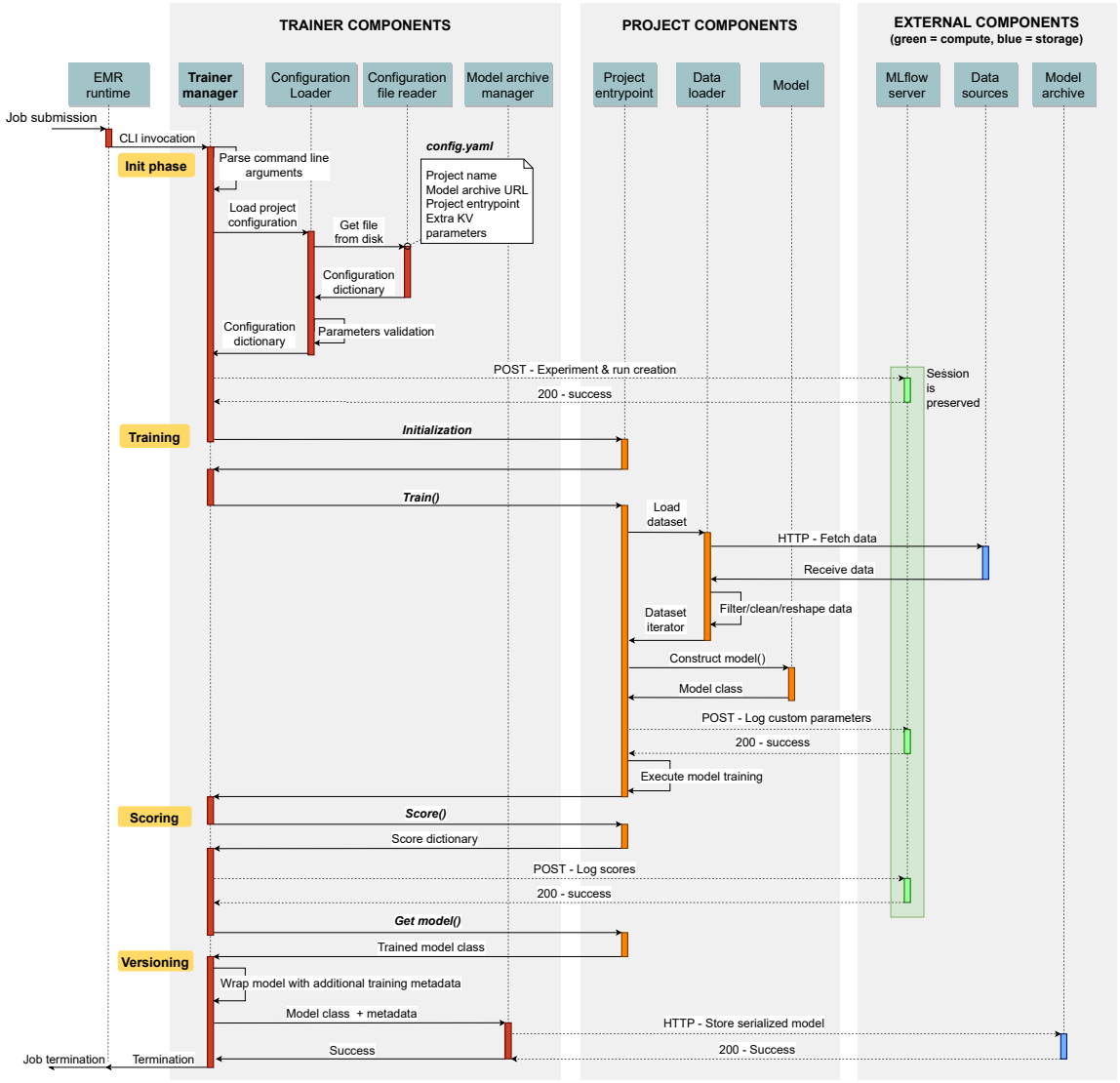

Figure 5.10: Trainer component sequence diagram

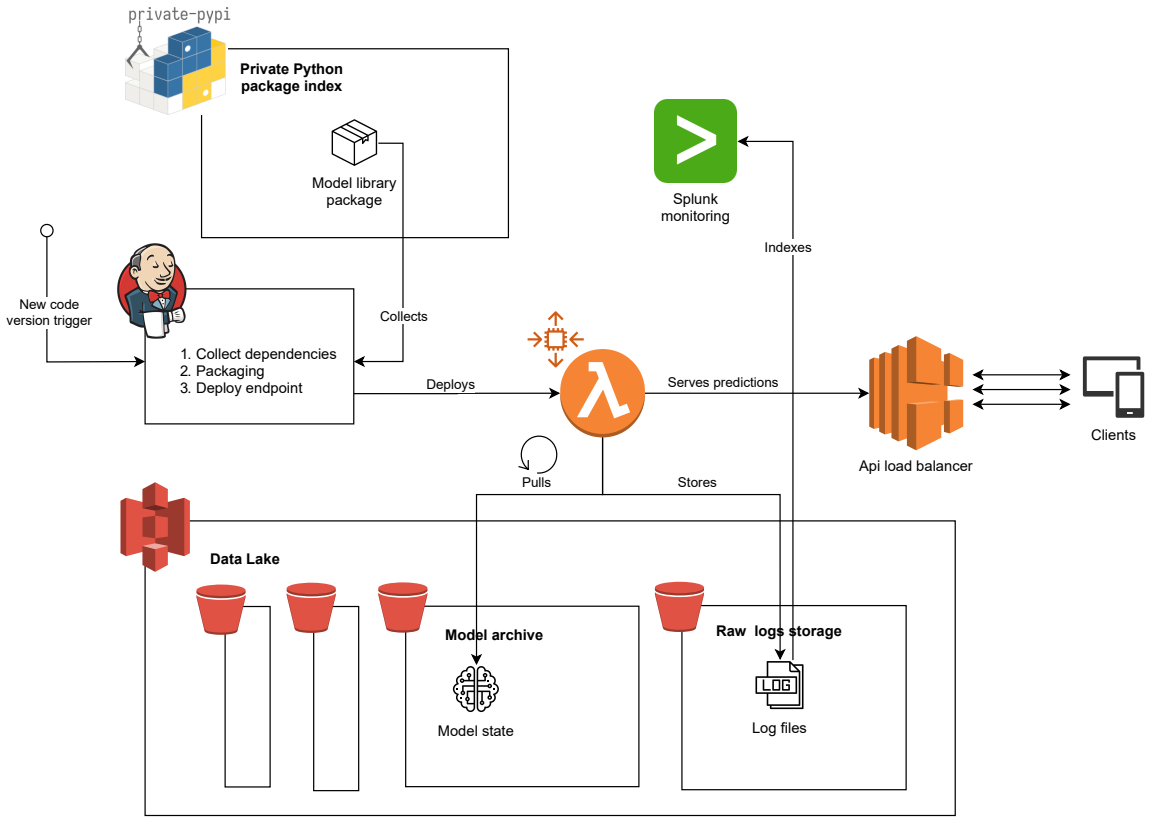

Figure 5.11: Predictor architecture implementation

The component focuses on providing an abstraction to the interaction with the model archive and to automatically handle the import of the necessary dependencies for the model. By doing so the endpoint's architecture definition has to define, among the other dependencies, only the Predictor component and the desired model to use with the corresponding version.

Figure 5.12 shows all of the steps mentioned above in a sequence diagram. In particular, there are two main stages are the Cold Start and the Warm state. They match directly to the Lambda's internal cold start and warm state. During these, the Predictor library executes two major operations: establishing a communication with the model archive and a dynamic de-serialization to load the necessary model logic. The component indeed fetches the latest model state delivered in the archive for a given version of the desired project. To achieve so, it leverages the static reference defined in the model archive schema to uniquely identify the latest deployed model. Consecutively, the Predictor imports the necessary libraries for the model to be de-serialized and proceeds with the de-serialization into a Python object. Therefore, at

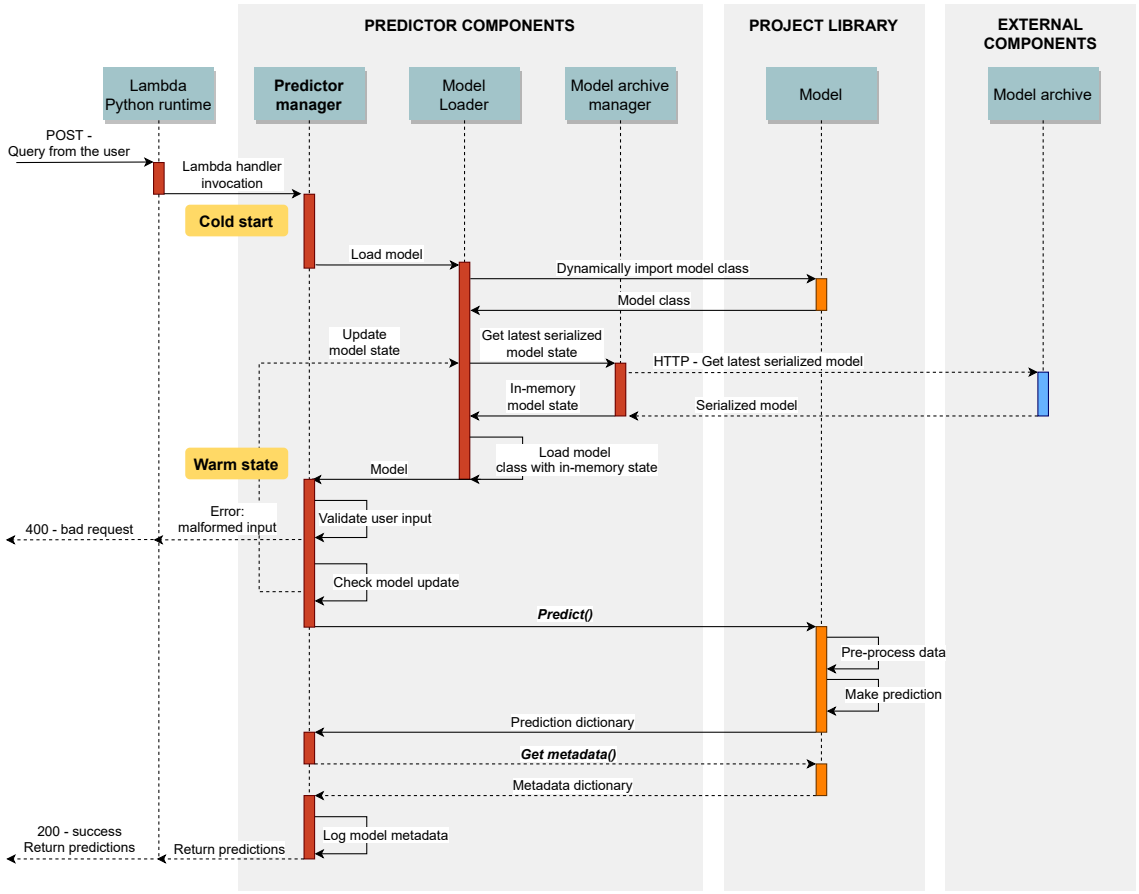

Figure 5.12: Predictor component sequence diagram

this stage, the framework delivers the object to the endpoint. Consequently, the latter can leverage the model to perform real-time predictions.

Once the latest state of the model is fetched and in a later time a new training run is executed, an inconsistency between the inference and model archive occurs. In fact, the model deployed to the endpoint serves a less recent version than the one available in the model archive. The goal is to guarantee at some degree that the state of the model for a given version always matches with the latest available. Moreover, during the update process, the goal is to have zero downtime of the endpoint.

This problem is comparable to the data consistency problem in distributed systems. In particular, the model archive handles the write operations whereas the endpoint the read operations. This scenario offers two consistency models which determine different complexity and availability:

- *Strong consistency* guarantees that every new model is immediately

deployed to the inference endpoint. To achieve this it is necessary to perform an atomic update operation in both model archive and inference endpoint which causes a temporary downtime for the latter one.

- *Eventual consistency* guarantees that if no new updates are made to the model state, eventually the endpoint will run the last updated state. This is achievable with no downtime for the inference endpoint.

Given this situation, the design elegies availability as opposed to immediate update. Therefore the Predictor implements an adjustable expiration timer for the loaded model state. This ensures that the endpoint runs a recently outdated state at most for an elapsed time equals to the expiration time range.

As a conclusive stage of the prediction step, there is the logging phase which emits all the logging information collected during the invocation in line with what mentioned in Section 5.3.3.

Chapter 6

Evaluation

The previous section described the principles behind the proposed machine learning framework and consecutively framework's design choices and implementation for each component have been described. This section proceeds to analyze the degree of effectiveness of the framework in comparison with the other state-of-the-art solutions discussed in Section 2.5. In particular, limitations of the frameworks will be addressed for two selected use-cases of supervised and unsupervised learning.

The evaluation of the framework itself follows a bottom-up approach by first analyzing the strengths and weaknesses of the different components and from there the perspective enlarges to address the analysis of the behaviour of the entire framework.

The main criteria the evaluation takes into account are the feature requirements specified in the problem statement of Section 3.

Ultimately, a feasibility demonstration is discussed to present operational conditions of the solution.

6.1 Logic and state dependency management

The state and dependency management workflow is evaluated according to a set of common scenarios the thesis identified. These scenarios have occurred in the two use cases and can be generalized, not extensively, to other use cases based on the same pipeline of Section 2.2. These are:

S1 A data scientist wants to reproduce the latest training configuration for debugging a certain error which occurred during the last training execution.

S2 A DevOps engineer wants to reproduce the serving configuration for debugging a certain error which occurred in the inference endpoint.

S3 The CI/CD system is in charge to package the project for Lambda deployment. It has to package the source code and external dependencies together.

Although S1 and S2 seem to be the same scenario, they are not. In fact, S1 aims to reproduce the conditions under which the training has occurred, hence training dependencies, such as `pyspark`, have to be included. On the other hand, in S2 the set of dependencies differ from S1 due to differences in libraries used only in the serving channel, such as web server frameworks.

From these scenarios we derive a set of evaluation criteria:

C1 Does a developer have the possibility to automatically reproduce the desired environment?

C2 What and how many required tools have to be made available to set up the project?

C3 How much space these required tools take?

The Model-as-a-Package paradigm has its implementation in production as the Trainer component and the CI/CD pipeline which enables the publication of Python modules from the project's source code. The solution does not rely on external libraries additional to `pip` to install the model logic in a new environment. The size of the single `pip` library and its dependency is 10MB.

On the other hand, MLflow and ClearML offer alternative solutions for code dependency management.

MLflow's Models component offers the capability to record model code and model state and store them in the model registry. Additional dependencies are recorded manually and are decoupled from the model logic itself. Consequently, MLflow offers additional tooling to enable the integration of the versioned model logic and state. However, dependency issues arise when the working environment is not mirroring the training conditions. To address this, MLflow provides an integration with the `conda` environment manager, however, this approach relies on the `conda` toolkit to be available in the system. In total, all the requirements needed to get started with MLflow reserve at least 500MB of space in the disk: 400MB for the `conda` runtime and 210MB for the additional MLflow library. In this context, all scenarios S1, S2, and S3 should necessarily set up the `conda` and `mlflow` dependencies pre-installed in the working environment. Moreover, S1 and S2 will not have

the capability to separately install train-only and inference-only dependencies since MLflow does not provide this capability.

ClearML instead performs automated code and dependency discovery. The information is consequently stored with the model state in the object storage solution. This solution mimics the MLflow design. However, no solution for the reproduction of the environment is provided, hence all S1, S2, S3 should have manual logic to fetch the necessary components and install dependencies.

Both the solutions address the problem of storing the logic and the state of a model consistently through time.

Framework	Model logic versioning	Model state versioning
MaaP framework	Package index	Object storage
MLflow	Object storage	Object storage
ClearML	Object storage	Object storage

Table 6.1: Logic and state dependency management support - 1

Framework	Dependency management	Environment setup requirements
MaaP framework	Setuptools discovery + manual	`pip`
MLflow	Manual	MLflow + `conda`
ClearML	Automatic	—

Table 6.2: Logic and state dependency management support - 2

Tables 6.1 and 6.2 recap the functionalities and limitations provided by the framework. Therefore, after having analyzed the various aspects of each dependency management system Table 6.3 evaluates the discussed frameworks against the requirement criteria.

Framework	C1	C2	C3
MaaP framework	Yes	`pip` (1)	10MB
MLflow	Yes	`conda` + `mlflow` (2)	more than 500MB
ClearML	No	—	—

Table 6.3: Evaluation against the specified criteria

6.2 Server-free model archive

The server-free model archive, the MLflow Model Registry and the ClearML logging solutions define three alternative approaches to the long term model storage logic and architecture. These have been discussed in their related sections: Section 2.5 for MLflow and ClearML solutions, Section 5.3.1 for MaaP framework for the model archive.

These solutions aim to solve the challenge of providing versioning of ML model or any other artefact generated by the various training sessions. In particular, both the *Mining of malicious events* and the *Classification of news articles* would benefit from the model archive in these scenarios:

S1 The *Mining of malicious events* model requires an update to serve similarities among the most recent events

S2 The ETL tasks have fetched more data and *Classification of news articles* model requires additional training iterations to feed the new data available to increase in accuracy performance.

S3 The tracking web server is not available due to temporary updates or disruptions. The inference endpoints have to be able to still fetch reliably the appropriate model state.

S4 The latest model version for the *Mining of malicious events* is providing very dissimilar events. The model has to be temporarily rolled back to the previous version to guarantee the quality of the service.

From these scenarios, the crucial factors which define a solid and fault-proof model registry are defined:

C1 Support for versioning multiple model versions.

C2 Support for versioning multiple scheduled re-trainings for the same version.

C3 Availability: The capability of fetching a certain model when desired.

C4 Version tagging: The capability to apply tags to model versions.

C5 Rollback: The capability to hand-pick models if required.

The main limitation of the model versioning architectures of MLflow and ClearML is the full dependence on the web server component. This single

Framework	C1	C2	C3	C4	C5
MaaP framework	Yes	Yes	High	Yes	Yes
MLflow	Yes	Yes	Depends on web server availability	Yes	Yes
ClearML	Yes	Yes	Depends on web server availability	Yes	Yes

Table 6.4: Model versioning solutions comparison

point of failure approach makes the service to be tightly bound to the availability of the server. This is even more accentuated by the fact that the object storage structure follows an internal logic which is not just standalone but instead relies on additional pieces of information, including experiment or run ids and tags, which are not stored in the same location as the rest of the model objects.

6.3 Automated logging capabilities

Logging is a crucial functionality to guarantee the proper monitoring and alarming during both training and inference stages. Moreover, it is crucial to have a separation of interests between the operational metrics and the data science metrics.

Complete monitoring and alarming capabilities at training time have to support the following use cases:

C-t 1 Monitoring of system/architecture level characteristics such as usage of computational resources through time.

C-t 2 Alarming based on metrics generated by 1.

C-t 3 Monitoring of meaningful data science scores through time. These include model's hyper-parameters, metrics and additional artefacts.

C-t 4 Alarming based on scores generated by 3.

At training time, Metaflow offers the best tracking and logging thanks to a rigorous checkpoint system. Indeed, it enables the capability of inspecting batch jobs even after the execution of those. This is unique compared to the other state-of-the-art solutions. The alternatives in this context offer tracking and tracing capabilities through external web services. The proposed framework offers the same capabilities since the proposed architecture

Framework	C-t 1	C-t 2	C-t 3	C-t 4
MaaP framework	No	No	Yes	No
MLflow	No	No	Yes	No
DVC	No	No	With limitations	No
Metaflow	Yes	No	Yes	No
Sacred	No	No	Yes	No
ClearML	Yes	No	Yes	No

Table 6.5: Training monitoring and alarming capabilities

leverages the deployment of MLflow server instance for training tracking and tracing purposes.

On the other hand, there are major differences between the monitoring capabilities at the inference level between the two solutions.

The evaluation of the logging capabilities at inference side requires to comply with the following set of capabilities:

C-i 1 Monitoring availability and performance of the endpoint.

C-i 2 Alarming based on metrics generated by 1.

C-i 3 Perform data validation and detect data drift.

C-i 4 Trace model version for each prediction.

C-i 5 Trace re-training session for each model version.

C-i 6 Track relevant data science metrics related to the prediction for the problem.

Table 6.6 describes the differences between the proposed framework, MLflow and BentoML which are the only solutions offering model serving. MLflow, however, has limited support for inference metrics. On the other hand, the proposed framework and BentoML solutions have wide support for both tracking engineering and data science metrics tracking capabilities to support the monitoring the machine learning models. Being BentoML a solution for inference-only it does not provide automated support for tracking model version and metadata information. Yet, with little configuration, the support is possible.

To recap, the main end-to-end solution MLflow serving solution is not yet mature enough to support advanced monitoring requirements. Despite this, MLflow permits to implement internal logic in the model to perform

Framework	C-i 1	C-i 2	C-i 3	C-i 4	C-i 5	C-i 6
MaaP framework	Yes	Only logging	Yes	Yes	Yes	Supp.
MLflow	No	Only logging	Yes	No	No	No
BentoML	Yes	Only logging	Yes	Supp.	Supp.	Supp.

Table 6.6: Inference monitoring and alarming capabilities ("Supp." means that the capability is supported but not natively implemented)

monitoring in the desired way. However, this type of solution can be seen as a workaround since the prediction and monitoring implementations are in the same context. The proposed framework instead solves all the above data science monitoring challenges. This is made possible by a more comprehensive process of automatic metadata collection at training time which supports the various endpoints in tracing model specifications at a single query level. Yet, both frameworks are only able to produce metrics, thus both solutions require extra tooling for metric ingestion, visualization and monitoring.

6.4 Automated model update

None of the state-of-the-art solutions provides model update capabilities. This implies that whenever a new update for a model is published, the endpoint has to be deployed from scratch. In contrast, the Predictor component is capable of performing updates of model internal state in real time described in Section 5.3.3 and Section 5.4.3.

6.5 Deployment of ML models

As it has been mentioned in Section 2.1.1, there are three main state-of-the-art real-time serving solutions for machine learning models: containers, serverless applications, served through third-party services as AWS Sagemaker or ECS (through container deployments) and embedded as a dependency in external applications in IoT devices or mobiles phones. Each of the different scenarios has different advantages and limitations which have to be considered when selecting the appropriate deployment service. In this section, the goal is to evaluate how flexible is the deployment system of the proposed MLOps framework in contrast with the MLflow and BentoML. In particular, the main requirement is the support for the FaaS solutions and IoT devices. Moreover, in contrast with container deployments, FaaS and

IoT devices may face physical limitations concerning the hardware resources available as defined in Section 3. Hence, the limitations of the approach are gathered:

1. Limited storage space: Serverless applications and IoT/mobile devices have strong storage space limitations.
2. Low I/O performance: Depending on the volume requirements, devices may have limited I/O performance with storage volumes.
3. Multiple environments: Inference endpoints may not share the same runtime as the Training process.

Despite MLflow natively supports container deployments, no native support is provided for FaaS and IoT solutions. Adaptation of the workflow based on this framework also fails due to the large size reached by the basic set of dependencies of the framework itself. This alone exceeds the maximum limit for lambda packages. For the same reason, it should be avoided to adapt MLflow to IoT solutions due to the overhead introduced by the same basic dependencies.

Moreover, MLflow's implementation is not able to support in-memory model deployment. Therefore this forces the endpoint to store the model on disk first. As a consequence, the model deployment is slower or in the worst-case scenario not possible, in particular on those service with limited local storage compared to RAM such as serverless applications.

On the other hand, the MaaP framework and BentoML framework share many similarities in the project packaging for FaaS deployments, specifically AWS Lambdas.

One more advantage of the MaaP framework is its capability to install the project and all Predictor with the consequent dependencies as a standard Python package. This simplifies the integration of a model in external services yet without dropping the automated model update and logging features introduced for the prediction phase.

Therefore, the results in Table 6.7 the use cases for which the different solutions are suitable for.

One additional limitation with the MLflow serving solution is the difficulty to perform project's integration tests in CI/CD pipelines. The current implementation of the latter does not allow to test with ease the integration between the model logic and the service which exposes the model to predictions. Workarounds can be set up but those still require to pull all the real model state from the Model Registry. This is not feasible when the model becomes large in size because the cost in terms of resources and time is prohibitive.

Framework	Container	FaaS	IoT devices
MaaP framework	Not developed yet	Yes	Yes
MLflow	Yes	No	No
BentoML	Yes	Yes	Limited

Table 6.7: Serving channels support

6.6 Framework evaluation

Framework	Model archive	End-to-end capabilities	Serverless/IoT support
MaaP framework	Yes	Yes	Yes
MLflow	Yes	Yes	No
DVC	Yes	Training	No
BentoML	No	Serving	Yes
Metaflow	No	Training/Batch	No
Sacred	No	No	No
ClearML	Yes	No	No

Table 6.8: Requirements evaluation - 1

The different comparisons between the proposed MaaP-based framework and the state-of-the-art solutions reveal substantial differences in the feature set offered by each of them. On one hand, there are solutions like Metaflow only focused on training and ETL pipelines whereas other like BentoML which focus on the serving phase.

The only end-to-end solution is MLflow. However, due to its beginnings focused on the tracking server, the model serving capabilities are not yet mature enough to support highly customized pipelines. On the other hand, instead, the MaaP has been conceived and built to address specifically advanced scenarios, and from those generalize to more simple use-cases with ease.

Another crucial difference between the two end-to-end solutions, namely MLflow and the MaaP framework, is the set of components which constitute the two solutions. The first is heavily relying on the server component which has the role of synchronizing the various services and serve the various use-cases. On the other hand, the MaaP is a distributed pair of components (Trainer and Predictor) which do not rely on a centralized orchestration but

Framework	Automated model update capabilities	Automated model tracking and tracing
MaaP framework	Yes	Yes
MLflow	No	Basic
DVC	No	No
BentoML	No	Basic
Metaflow	No	Batch only
Sacred	No	No
ClearML	No	No

Table 6.9: Requirements evaluation - 2

instead on a set of well-defined design decisions.

Yet, the MaaP framework still suffers from a set of limitations. The first is the lack of support for training time alarming capabilities. Although the solution provides a set of measures to prevent various deployment threats and problems, there is no component in charge of directly catching the anomalous behaviours and report those to the engineers to escalate the issues and act on these. The second is the lack of alarming at the inference side. However, this is a well known and already-solved problem. In fact, to address this use case the Predictor component's automated logging is handled by indexing services such as Splunk[34] or Grafana, from which the engineer can compute data aggregations, define dashboards for data visualization and define meaningful alarming.

The evaluation of the model against the aforementioned use-cases shows that the first productionization of a baseline algorithm for both supervised and unsupervised problems can be performed in less than 8 man-hours of work (in a day) with all the features addressed. This confirms the maturity of the framework and proves the success in the scalability of the solution without sacrificing in simplicity. Moreover, the challenge of the deployment of large models in limited resources has been successfully tackled. The MaaP paradigm with a correct dependency definition enabled the use of specific dependencies in the train-only phase, thus the limitations in space of AWS Lambdas have not interfered with the deployment process. Moreover, large model states, such as the pairwise similarity matrix of the "Mining of malicious events" use-case, have been successfully imported into RAM and deserialized. The same setup, evaluated with the MLflow serving solution, was not possible to achieve due to the excessive size of the MLflow's minimum set of dependencies alone. Indeed, the compressed archive was 63.2 MB where the AWS limit is 50 MB.

6.7 Use case demonstration

The feasibility of the end-to-end pipeline is proposed. The first and most mature use case for the MaaP framework is the Mining of malicious events referred internally as Incident Similarity. The project has a weekly re-training job scheduled to guarantee updated similarity scores and to avoid missing the most recent incidents detected by the Cloud Protection engine. Therefore, Figure 6.1 has the goal to show all the effectiveness of the MaaP framework for the training phase. In particular, one can notice the various metadata collected by the framework such as the model version and whether the trained model has passed the validation criteria before being versioned. In addition, the custom parameters and metrics are visible.

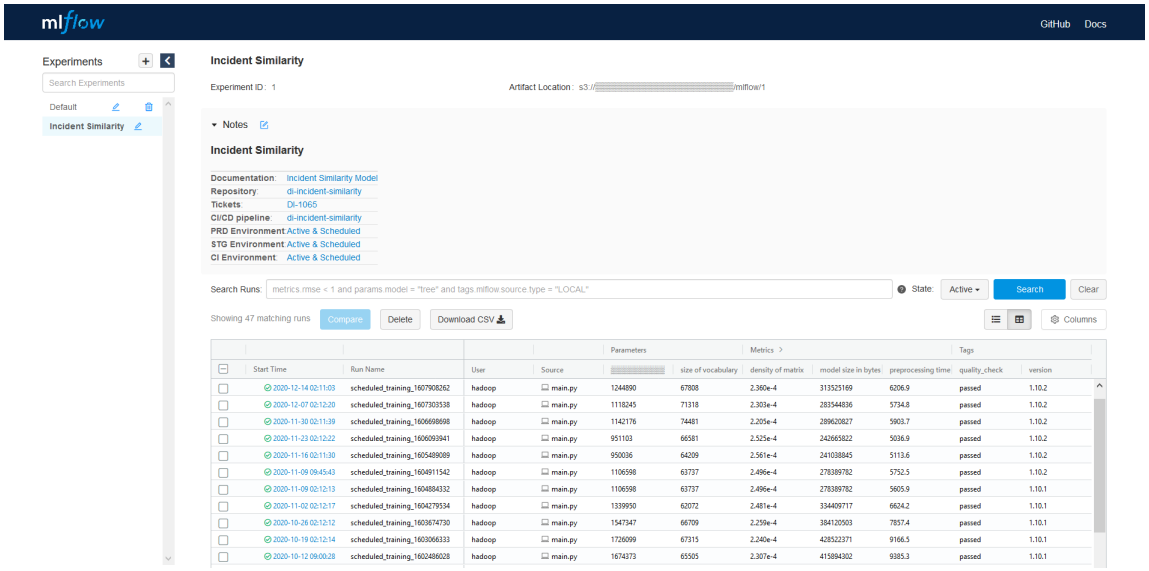

Figure 6.1: Snapshot of the training environment metric tracking server

As mentioned in the architecture definition, every time a re-training session takes place a new model version is stored in the model archive ready to be served in the endpoint. To demonstrate this use case Figure 6.2 shows the automated re-training deployment taking place in the bottom time chart whereas the top chart shows the improvement from an older to a newer version of the model architecture.

Figure 6.2 also proves the capabilities of automated logging of the endpoint, in particular, the metadata included in the model during the training phase are logged by the inference Lambda function endpoint for tracking and tracing purposes. Finally, the two timecharts demonstrate the scalability of

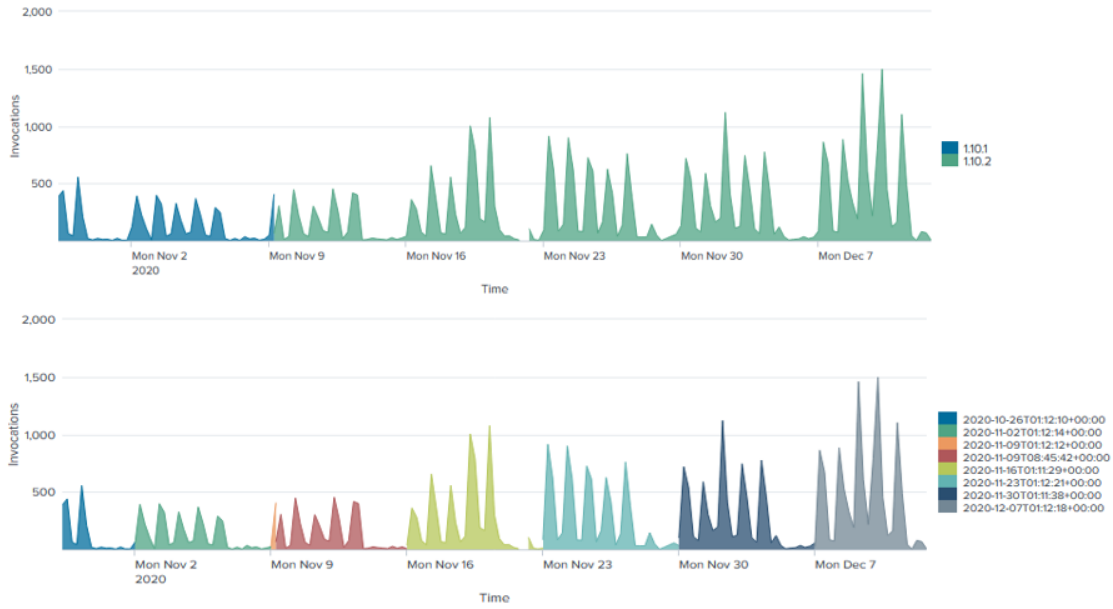

Figure 6.2: Snapshot metrics from the inference endpoint collected in Splunk

the solution which in this case proves to be scalable at the level of at least 1500 peak requests per day which is projected to grow in the next months.

Chapter 7

Conclusions

In past years, the most popular cloud providers have been investing time and resources to offer managed solutions for standard prediction problems varying from machine translation to general image classification. This is a considerably large improvement which has enabled the use of machine learning algorithms for those companies which do not invest time and effort into a fully formed data science unit. This approach is reliable yet has several constraints which limit the use cases of any of the solutions. Consequently, middle size companies and early stages startups focused on technology solutions do not find any of the two approaches suitable for their needs since they are either too expensive or poorly customizable. However, recently, new tools such as MLFlow and Metaflow have been disclosed to the public as open-source software by solid technology firms as Databricks and Netflix. These are the initial attempts towards an increasingly standardized yet flexible and reliable approach for robust machine learning lifecycles. These software tools introduce new logic and practices which have an impact on the methodologies and approaches of their users, both engineers and scientists.

Each of the aforementioned tools and other already available bring a precise view of the problems which may benefit some pipelines and affect negatively others. In fact, none of the tools aims to provide full end-to-end support because the effort to accommodate all the different MLOps aspects would be extremely huge and yet there are already DevOps tools which may be adapted to accomplish MLOps tasks. Only MLflow has made a first attempt to cover the end-to-end pipeline able to generalize among different use-cases. However, specific design decisions have introduced simplifications which preclude the engineer to have full flexibly in the serving channel.

Therefore, the MaaP concept is introduced in this thesis. With the idea of managing the code of machine learning models as if it is a distributable code package, several model versioning and dependency issues are addressed

in a DevOps manner. To leverage and implement this concept, the thesis has proposed the MaaP framework. The latter introduces novel practices, methodologies and features which support the job of the machine learning engineer. The solution implements novel automation capabilities for model versioning, dependency management and in the life-cycle of the inference endpoint. Results have shown that these enable optimized services and a greater set of supported serving solutions, including serverless applications and low resources IoT devices. Moreover, the framework has proved to be suitable for use-cases covered by the MLflow suite and more advanced and customized problems such as ad-hoc unsupervised recommendation systems. The evaluation phase has also shown that the MaaP framework still lacks of more advanced alarming capabilities. Despite this limitation, the solution does not prevent the user to add additional logic to the environment to enable alarming through external tools or services.

This is the starting point for an improved version of the end-to-end solution including features such as alarming and online learning support. Moreover, when looking at the big picture, the novel ideas introduced by the MaaP framework and the successful use-case stories prove that the proposed solution is a step forward towards an increasingly effective and efficient MLOps culture.

Bibliography

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)* (2016), pp. 265–283.

[2] Agile Stacks. Machine learning workflow. https://docs.agilestacks.com/article/gkyq26pzmr-creating-an-ml-pipeline, 2020. Accessed: 2020-12-28.

[3] Agrawal, P., Arya, R., Bindal, A., Bhatia, S., Gagneja, A., Godlewski, J., Low, Y., Muss, T., Paliwal, M. M., Raman, S., Shah, V., Shen, B., Sugden, L., Zhao, K., and Wu, M.-C. Data platform for machine learning. In *Proceedings of the 2019 International Conference on Management of Data* (New York, NY, USA, 2019), SIGMOD '19, Association for Computing Machinery, p. 1803–1816. https://doi.org/10.1145/3299869.3314050.

[4] AITrends. Why mlops (and not just ml) is your business' new competitive frontier. https://www.aitrends.com/machine-learning/mlops-not-just-ml-business-new-competitive-frontier/, 2018. Accessed: 2020-12-28.

[5] Akhtar, N., and Mian, A. Threat of adversarial attacks on deep learning in computer vision: A survey. *IEEE Access 6* (2018), 14410–14430.

[6] Algorithmia. 2020 state of enterprise machine learning. https://info.algorithmia.com/hubfs/2019/Whitepapers/The-State-of-Enterprise-ML-2020/Algorithmia_2020_State_of_Enterprise_ML.pdf, 2020. Accessed: 2020-12-28.

[7] Allegro.ai. Clearml - auto-magical suite of tools to streamline your ml workflow. https://github.com/allegroai/clearml, 2020. Accessed: 2020-12-28.

[8] Altexsoft. Mlops: Methods and tools of devops for machine learning. https://www.altexsoft.com/blog/mlops-methods-tools/, 2020. Accessed: 2020-12-28.

[9] Amazon. Amazon web services. https://aws.amazon.com/. Accessed: 2020-12-28.

[10] Amazon. Aws lambda layers. https://docs.aws.amazon.com/lambda/latest/dg/configuration-layers.html. Accessed: 2020-12-28.

[11] Amazon. Aws lambda quotas. https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html. Accessed: 2020-12-28.

[12] Amazon. What is cloud computing. https://aws.amazon.com/what-is-cloud-computing/. Accessed: 2020-12-28.

[13] Amazon. What is a data lake? https://aws.amazon.com/big-data/datalakes-and-analytics/what-is-a-data-lake/, 2013.

[14] Anaconda. Anaconda - your data science toolkit. https://www.anaconda.com/products/individual, 2020. Accessed: 2020-12-28.

[15] Andy Konwinski, Cyrielle Simeone, M. Z. Managed mlflow on databricks now in public preview. https://databricks.com/blog/2019/03/06/managed-mlflow-on-databricks-now-in-public-preview.html, 2019. Accessed: 2020-12-28.

[16] Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., et al. A view of the parallel computing landscape. *Communications of the ACM 52*, 10 (2009), 56–67.

[17] BentoML. Bentoml - model serving made easy. https://www.bentoml.ai/, 2020. Accessed: 2020-12-28.

[18] Breck, E., Cai, S., Nielsen, E., Salib, M., and Sculley, D. The ml test score: A rubric for ml production readiness and technical debt reduction. In *Proceedings of IEEE Big Data* (2017).

[19] Brendel, W., Rauber, J., and Bethge, M. Decision-based adversarial attacks: Reliable attacks against black-box machine learning models. *arXiv preprint arXiv:1712.04248* (2017).

[20] Burguera, I., Zurutuza, U., and Nadjm-Tehrani, S. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices* (2011), pp. 15–26.

[21] Burt, A., Leong, B., Shirrell, S., and Wang, X. G. Beyond explainability: A practical guide to managing risk in machine learning models. *The Future of Privacy Foru m White Paper* (2018).

[22] Dawson, R. Why is devops for machine learning so different? https://hackernoon.com/why-is-devops-for-machine-learning-so-different-384z32f1, 2019. Accessed: 2020-12-28.

[23] Dirafzoon, A. A guide to production level deep learning. https://github.com/alirezadir/Production-Level-Deep-Learning/blob/master/README.md, 2020. Accessed: 2020-12-28.

[24] DVC. Dvc - open-source version control system for machine learning projects. https://dvc.org/, 2020. Accessed: 2020-12-28.

[25] F-Secure. F-secure - about us. https://www.f-secure.com/en/about-us, 2020. Accessed: 2020-12-28.

[26] Fisher, C., et al. Cloud versus on-premise computing. *American Journal of Industrial and Business Management 8*, 09 (2018), 1991.

[27] Forbes. Data-driven culture: Everyone now wants one, but what is it? https://www.forbes.com/sites/joemckendrick/2020/03/29/now-everyone-wants-a-data-driven-culture/?sh=311b9cbb761e, 2020. Accessed: 2020-12-28.

[28] Google. Cloud computing services - google cloud. https://cloud.google.com/. Accessed: 2020-12-28.

[29] Griffin, K., Schneider, S., Hu, X., and Chiueh, T.-C. Automatic generation of string signatures for malware detection. In *International workshop on recent advances in intrusion detection* (2009), Springer, pp. 101–120.

[30] IDSIA. Sacred. https://github.com/IDSIA/sacred. Accessed: 2020-12-28.

[31] Lang, K. 20 newsgroups dataset. http://qwone.com/ jason/20Newsgroups/. Accessed: 2020-12-28.

[32] McKinney, W., et al. pandas: a foundational python library for data analysis and statistics. *Python for High Performance and Scientific Computing 14*, 9 (2011).

[33] Merritt, R. What is mlops? https://blogs.nvidia.com/blog/2020/09/03/what-is-mlops/, 2020. Accessed: 2020-12-03.

[34] Michael Baum, Rob Das, E. S. Splunk - the data-to-everything platform built for the cloud. https://www.splunk.com/. Accessed: 2020-12-28.

[35] Microsoft. Cloud computing services - microsoft azure. https://azure.microsoft.com/en-us/. Accessed: 2020-12-28.

[36] Microsoft. What is the team data science process? https://docs.microsoft.com/en-us/azure/machine-learning/team-data-science-process/overview, 2020. Accessed: 2020-12-28.

[37] Netflix. Metaflow - a framework for real-life data science. https://metaflow.org/, 2020. Accessed: 2020-12-28.

[38] Park, J.-K., Kwon, B.-K., Park, J.-H., and Kang, D.-J. Machine learning-based imaging system for surface defect inspection. *International Journal of Precision Engineering and Manufacturing-Green Technology 3*, 3 (2016), 303–310.

[39] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch.

[40] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research 12* (2011), 2825–2830.

[41] Poccia, D. Aws lambda – container image support. https://aws.amazon.com/blogs/aws/new-for-aws-lambda-container-image-support/, 2020. Accessed: 2020-12-28.

[42] Poccia, D. A shared file system for your lambda functions. https://aws.amazon.com/blogs/aws/new-a-shared-file-system-for-your-lambda-functions/, 2020. Accessed: 2020-12-28.

[43] Preston-Werner, T. Semantic versioning 2.0.0. https://semver.org/, 2020. Accessed: 2020-12-28.

[44] Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., and Dennison, D. Hidden technical debt in machine learning systems. In *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds. Curran Associates, Inc., 2015, pp. 2503–2511. http://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems.pdf.

[45] Serverless.com. Serverless (faas) vs. containers - when to pick which? https://www.serverless.com/blog/serverless-faas-vs-containers. Accessed: 2020-12-28.

[46] Tsai, C.-F., Hsu, Y.-F., Lin, C.-Y., and Lin, W.-Y. Intrusion detection by machine learning: A review. *expert systems with applications 36*, 10 (2009), 11994–12000.

[47] Vafeiadis, T., Diamantaras, K. I., Sarigiannidis, G., and Chatzisavvas, K. C. A comparison of machine learning techniques for customer churn prediction. *Simulation Modelling Practice and Theory 55* (2015), 1–9.

[48] van Rijmenam, M. How unlimited computing power, swarms of sensors and algorithms will rock our world. https://datafloq.com/read/unlimited-computing-swarm-sensors-algorithms-world/2138, 2016. Accessed: 2020-12-28.

[49] Walt, S. v. d., Colbert, S. C., and Varoquaux, G. The numpy array: a structure for efficient numerical computation. *Computing in science & engineering 13*, 2 (2011), 22–30.

[50] Zaharia, M., Chen, A., Davidson, A., Ghodsi, A., Hong, S. A., Konwinski, A., Murching, S., Nykodym, T., Ogilvie, P., Parkhe, M., et al. Accelerating the machine learning lifecycle with mlflow. *IEEE Data Eng. Bull. 41*, 4 (2018), 39–45.

[51] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., Stoica, I., et al. Spark: Cluster computing with working sets. *HotCloud 10*, 10-10 (2010), 95.