University of L'Aquila

Department of Information Engineering, Computer Science and Mathematics

Course of Software Architectures 2019-20

AI Operationalization in the Cloud for healthcare systems

by **Y**et**A**nother**S**oftwareArchitecture**T**eam

Supervisor: Prof. Henry Muccini

Team Name		**YAST**
Name-Surname	ID	Email Address
Emanuele Palombo	260847	emanuele.palombo@univaq.it

January 16, 2020

Contents

Chapter 1

Challenges and Risk Analysis

Table 1.1: Risk Management Table

Risks	**Date of Identification**	**Date of Resolution**	**Method of Resolution**
Huge and complex project	18/11/19	22/11/19	Small parts that interconnected can fulfill the requirements provided
Massive (ad-hoc, batch) training and many small jobs that serve the predictions	18/11/19	22/11/19	Distributed / parallel and Load balancing computing
Allow best compatibility and performance between the system components	18/11/19	24/11/19	Choose the most used, documented and trusted tools: Apache family for the major of technologies utilized in the system
Lack of knowledge on the specific domain	18/11/19	never end	Study (a lot) and rely on pre-existing architecture (eg. Uber, Michelangelo, FBLearner, etc)

Chapter 2

Requirements Refinement

2.1 Functional Requirements

2.1.1 Training modality

- **Ad-hoc training**: this method allows to train and generate a model when required
- **Batch training**: provides a way by which it is possible to have a constantly refreshed version of the model in production. This allows the models to be re-trained at periodic time intervals using the latest data to ensure that the model is up-to-date with the latest patterns in data.
- **Dynamic Training**: this method allows such models to be trained dynamically using real-time data.

2.1.2 Serving models

- **Serving Batch**: the created (trained) model needs to be either provided as a file or in a database which can then be used by the application(s) to generate predictions (eg. It can be put "on the edge", on locally in phone).
- **Serving REST API**: the model will be served using a web-service which can be invoked by the applications using a REST call (eg. Phone can send a REST call to a web-service by passing the input required for the model as a JSON).

2.1.3 AI workflow

- **AI source code to deploy**
  - **Jupyter Notebook**: Data scientists build AI models using Jupyter notebooks with Python or R.
  - **Model deployment handling**: The Jupyter notebooks could send Model to the production engineers who are responsible for handling the deployment.
  - **Model to production**: All the three training models must allow to put the resulting trained model in production.
- **Versioning**: Models must be versioned and stored in a specific repository and made it available to the serving layer. Versioning includes source code of the model and persisting the structure of the model itself.
- **Monitoring**: System must constantly evaluate the accuracy in autonomous way.
- **Display Analytic**: Allow User to visualize and compare all logged information.

2.2 Non-Functional Requirements

- **Performance**
  - 8 Data Scientists working 200 h a month each.
  - The system will start hosting an initial set up of 10 specific projects each one will have a model that is built continuously during the working hours.
- **Scalability**
  - Each model is consumed by an API with calls from applications that can hit 200 calls/sec (ie. max).
  - For the batch processing, data is fed via files in csv format with maximum size 1Tb.
- **Availability**
  - The system must be available to the end user at all times High avalability 99.999% ("five nines"), 31.56 seconds of downtime per year
- **Reliability**
  - Fault tolerant and without single points of failure

notes: *Availability* and *Reliability* are common senses non-functional requirements added by us.

2.3 Requirements Prioritization

We have used a scale with a reduced resolution (ie. Low, Medium, High) to prioritize the requirements.

Table 2.1: Priorities of Functional Requirements

Requirement	Priority
Ad-hoc Training	High
Batch Training	High
Dynamic Training	Medium
Serving Batch	High
Serving REST API	High
Jupyter Notebook	High
Model deployment handling	Medium
Model to production	Medium
Versioning	High
Monitoring	High
Display Analytic	Medium

2.4 Use-Case Diagram

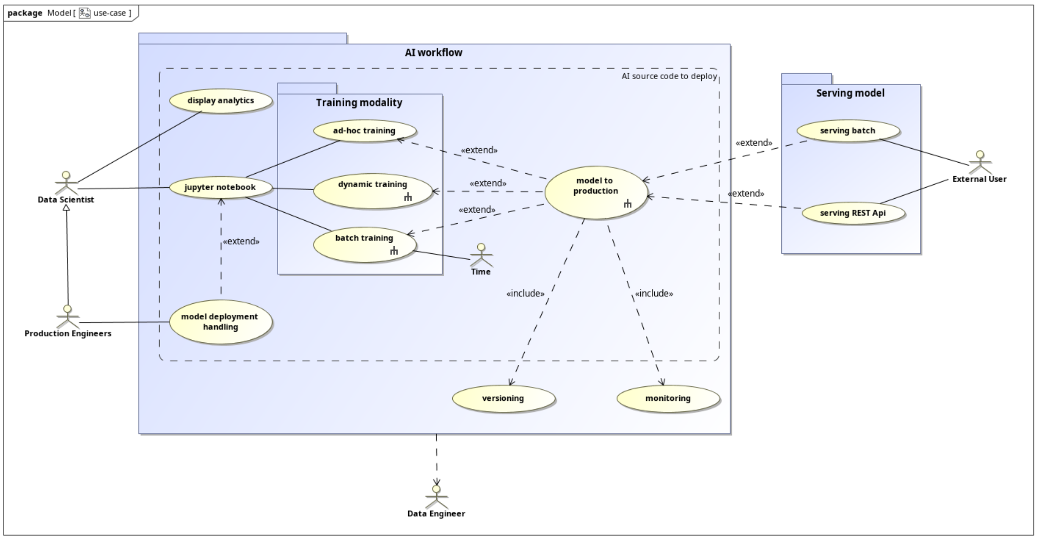


Figure 2.1: Use case diagrams of the system

2.4.1 Actors

- **Data Scientist:** writes AI algorithms and build the AI models.
- **Data Engineer**: defines data workflow and pipelines.
- **Production Engineers**: improves/fixes the source code of Data Scientist.
- **External User**: the "end" user wearing the device. These users will use the apps made by the Developers building AI-powered solution consuming models (eg. bracelet through the REST API). He includes both the developer and the user wearing the bracelet.
- **Time**: invokes jobs at periodic time intervals.

2.4.2 Most important Use-Case description

Name	Ad-hoc training
Actors	Data Scientist, Product Engineers, Data Engineer
Trigger	The Data Scientist writes the Python or R source code through the Jupyter Notebook
Description	If requested could be invoked the help of Production Engineer (model deployment handling) for improving/fixing the source code. The trained model will be produced against a DataSet (provided by the Data Scientist or present in the Data Lake)

Table 2.2: Ad-hoc training table

Name	Batch training
Actors	Time, Product Engineers, Data Engineer
Trigger	Scheduled (ie. at periodic time intervals)
Description	As for the Ad-hoc training, but it will be done with fresh data at periodic time interval decided by the Data Engineer

Table 2.3: Batch training table

Name	Dynamic training
Actors	Data Scientist, Product Engineers, Data Engineer
Trigger	External/Internal events with payload
Description	As for the Ad-hoc training, but it will be executed on new provided data by an internal or external events

Table 2.4: Dynamic training table

Name	Model to production
Actors	Data Scientist, Product Engineers, Data Engineer
Trigger	After ones of different training
Description	The generated trained model will be put in production, following the data pipelines defined by Data Engineer. The data pipelines are different, not always a model will be put in production

Table 2.5: Model to production table

Name	Versioning and Monitoring
Actors	Data Scientist, Product Engineers, Data Engineer
Trigger	Each time a model will be trained
Description	Source code, trained model and metrics will be versioned, saved and monitored

Table 2.6: Versioning and Monitoring table

Name	Serving batch
Actors	Data Scientist, Product Engineers, External user
Trigger	Trained model ready
Description	The trained model will be provided as a artifact (eg. file), which can then used by the application to generate predictions or recommendations (ie. edge cloud) (eg. embed AI in the bracelet)

Table 2.7: Serving batch table

Name	Serving REST API
Actors	External user
Trigger	Trained model deployed in production
Description	The trained model will be served using a web-service which can be invoked by the applications using a REST call (ie. remote prediction)

Table 2.8: Serving REST API table

Chapter 3

Informal Description of the System & its Software Architecture

3.1 Description of the System

Due the fact that the system must be built from scratch (we have only the key technologies), the group have tried to design the system described below.

The designed system tries to provide these main features: *Data Ingestion, Data Analysis and Validation, Data Transformation, Model Training, Model Validation, Model Serving, Tracking and Logging* all *supervisioned* and *orchestrated* by our system.

3.2 Sub-systems

The system is divided in 4 main logical groups:

3.2.1 Trigger/Starter

Here we can find the components that could start a computation.

- **Message Broker:** manages streaming data. A publish/subscribe broker that can receive messages from inside or outside world. Thanks to it, for example we can catch events that request the Dynamic Training (with training set in the payload). Another example is the storing of the features (without the target) that come from the served (online) models. Obviously any kind of message can be received by this (eg. deploy, training, etc...)

- **Task scheduler:** it executes tasks while following the specified dependencies of each ones. it's a task scheduling that allows you to create, orchestrate and monitor data workflows. eg. It allow the periodic execution of the bath training, follow step by step the AI model workflow (ie. source code to trained model). It can "control" all component of the system.
- **AI Model Source:** The Ide where the Data scientists write the source code of the models.

3.2.2 Processing (distributed system)

The resource located in a distributed pattern. The job executed here will be onerous and long, so they need to be spread on more processor (to respect the non-functional 2.2).

- **Processing Workers:** the "brain" run all the system code, especially for the training space of the model. It does also data transformation. Due the heavy process executed here, all jobs will have a shared access to a speed In-Memory DB. The jobs can also use a *Structured Data DB* (eg. new features that come from Served models).
- **DataLake:** the persistence memory of the "brain", hosted on a Distributed File System. It will maintain unstructured data (eg. trained model, CSV DataSet).

3.2.3 Versioning/Monitoring

To respect the functional requirements (see 2.1.3) this part preserves the chronological history of the trained models.

- **Tracking system:** collects and releases, when requested (ie. model rollback), the trained model information. It links all the information together (trained model, source code, training parameters, metrics of test, etc...)
- **Repository:** keeps track of the source code of the different models
- **Log History:** keeps tracks of the parameters and metrics of a single training

- **Visualizer and Analyzer:** dashboard to visualize and compare the logged information. This allow users to understand the better models, parameters and datasets.

3.2.4 Serving (external)

Outside of these three groups (in the Online part, see 3.4) we have:

- **Served models**: here is deployed the trained model that will give prediction/recommendation (scoring) to external users. Opposite to the Distributed System the jobs run here will be quite short with a lot of calls (hit 200 calls/sec, see 2.2). We have many wrapped trained model with a load balancer in front.
- **Online features**: a database that host information/features can be useful to the inference process of the served model (eg. user profile information non provided in the message request).

3.3 Architectural Patterns

The used architectural patterns are:

- **Publish/Subscribe:** used by the *Message Broker* component
- **Distributed system cluster:**
  - Master/Manager with multiple Workers/Slaves nodes that run application workloads
  - Replicated Load-Balanced Services to provide online prediction
- **Client-Server/Request-Response:** for the *External User* predictions requests (ie. REST API)
- **Pipes and Filters/Middleware:** used to save new features in the dataset (putting in front of the served models), but also during the data workflow (ie. data transformation)

3.4 System boundaries

We have divided our system in two main parts (also mentioned in the Subsystems, see 3.2):

- **Online:** this will provide service to the external world (eg. bracelet). It has a limited knowledge of the information and a limited provided service.
- **Offline:** here there are all the AI workflow/pipelines described in the requirements. These services are private but is also connected with Online part (eg. sync the data, deploy the model, etc...)

3.5 Overall architecture

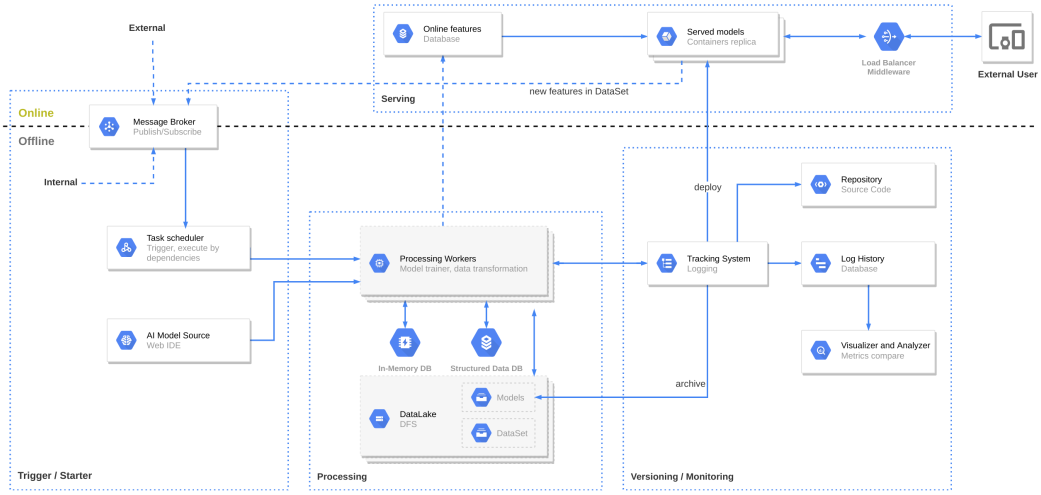


Figure 3.1: Overall architecture of the system.

3.6 Requirements Fulfillment

A list of functional and non-functional requirements and how we have tried to fulfill them:

3.6.1 Functional Requirements

3.6.1.1 Training modality

- **Ad-hoc training:** the training/test data a will be get from *DataLake > DataSet* and all *Data Scientist* can access to it, but also others sources are supported (eg. s3, http, etc). AI Model Source allows to write the AI algorithm and the trained model will be built through the distributed *Processing Workers*.

- **Batch training:** thanks to the *Task Scheduler* we can set the all tasks flow execution at periodic time intervals.
- **Dynamic Training:** through the broker can be published messages with new features in the payload that feed the training model (retrained in the *Processing Workers* following the right pipeline defined in the *Task scheduler*).

3.6.1.2 Serving models

- **Serving Batch:** the trained models (files) will be stored in different format (depend our needs) in the *DataLake > Models* and used from anyone.
- **Serving REST API:** this requirement is aimed by *Served models* that wrap the trained model in a container with a web server that expose the REST API.

3.6.1.3 AI workflow

- **AI source code to deploy:**
    - **Jupyter Notebook**: by the *AI Model Source.*
    - **Model deployment handling:** The *Production Engineers* can access to the AI Algorithm (also by *Tracking system*).
    - **Model to production:** This can be done by the *Task schedule*r or (worst case) with a simple hook on the tracking system (eg. on-commit).
- **Versioning & Monitoring:** refer to Versioning/Monitoring 3.2.3.

3.6.2 Non-Functional Requirements

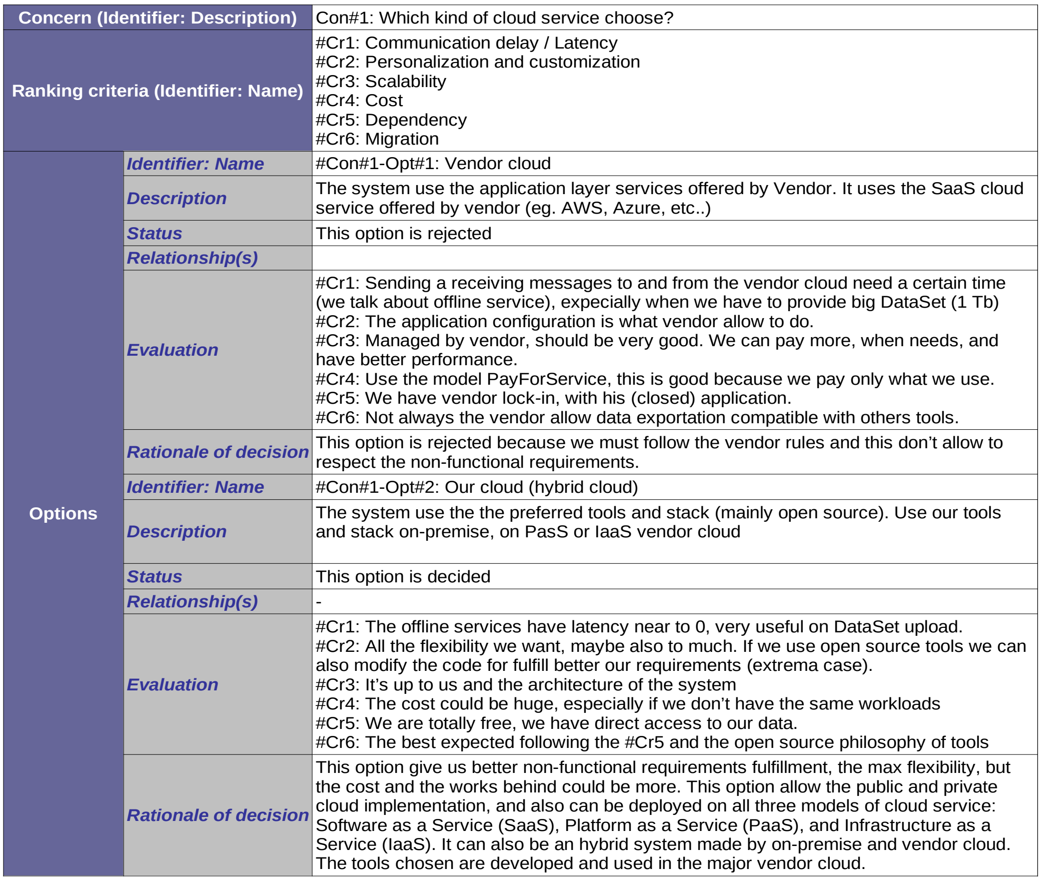
- **Performance:** thanks to the *Processing Work* (see 3.2.2 with the Master-Workers architectural pattern, see 3.3) we can execute onerous and long jobs, spreading them on more processor.
- **Scalability:** thanks to the *DataLake* (see 3.2.2) we can manage big files (1Tb) without any issues.
  Instead with the *Replicated Load-Balanced Services* on *Served Model* (see 3.3), we can achieve 200 call/sec balancing the works over the containers and spawn new one if needed.

- **Availability and Reliability:** all the redundancy and distribution should allow us to accomplish to this requirement

Chapter 4

Design decisions

4.1 Type of Cloud Service



Concern (Identifier: Description)		Con#1: Which kind of cloud service choose?
Ranking criteria (Identifier: Name)		#Cr1: Communication delay / Latency #Cr2: Personalization and customization #Cr3: Scalability #Cr4: Cost #Cr5: Dependency #Cr6: Migration
Options	***Identifier: Name***	#Con#1-Opt#1: Vendor cloud
	Description	The system use the application layer services offered by Vendor. It uses the SaaS cloud service offered by vendor (eg. AWS, Azure, etc..)
	Status	This option is rejected
	Relationship(s)	
	Evaluation	#Cr1: Sending a receiving messages to and from the vendor cloud need a certain time (we talk about offline service), expecially when we have to provide big DataSet (1 Tb) #Cr2: The application configuration is what vendor allow to do. #Cr3: Managed by vendor, should be very good. We can pay more, when needs, and have better performance. #Cr4: Use the model PayForService, this is good because we pay only what we use. #Cr5: We have vendor lock-in, with his (closed) application. #Cr6: Not always the vendor allow data exportation compatible with others tools.
	Rationale of decision	This option is rejected because we must follow the vendor rules and this don't allow to respect the non-functional requirements.
	Identifier: Name	#Con#1-Opt#2: Our cloud (hybrid cloud)
	Description	The system use the the preferred tools and stack (mainly open source). Use our tools and stack on-premise, on PasS or IaaS vendor cloud
	Status	This option is decided
	Relationship(s)	-
	Evaluation	#Cr1: The offline services have latency near to 0, very useful on DataSet upload. #Cr2: All the flexibility we want, maybe also to much. If we use open source tools we can also modify the code for fulfill better our requirements (extrema case). #Cr3: It's up to us and the architecture of the system #Cr4: The cost could be huge, especially if we don't have the same workloads #Cr5: We are totally free, we have direct access to our data. #Cr6: The best expected following the #Cr5 and the open source philosophy of tools
	Rationale of decision	This option give us better non-functional requirements fulfillment, the max flexibility, but the cost and the works behind could be more. This option allow the public and private cloud implementation, and also can be deployed on all three models of cloud service: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). It can also be an hybrid system made by on-premise and vendor cloud. The tools chosen are developed and used in the major vendor cloud.

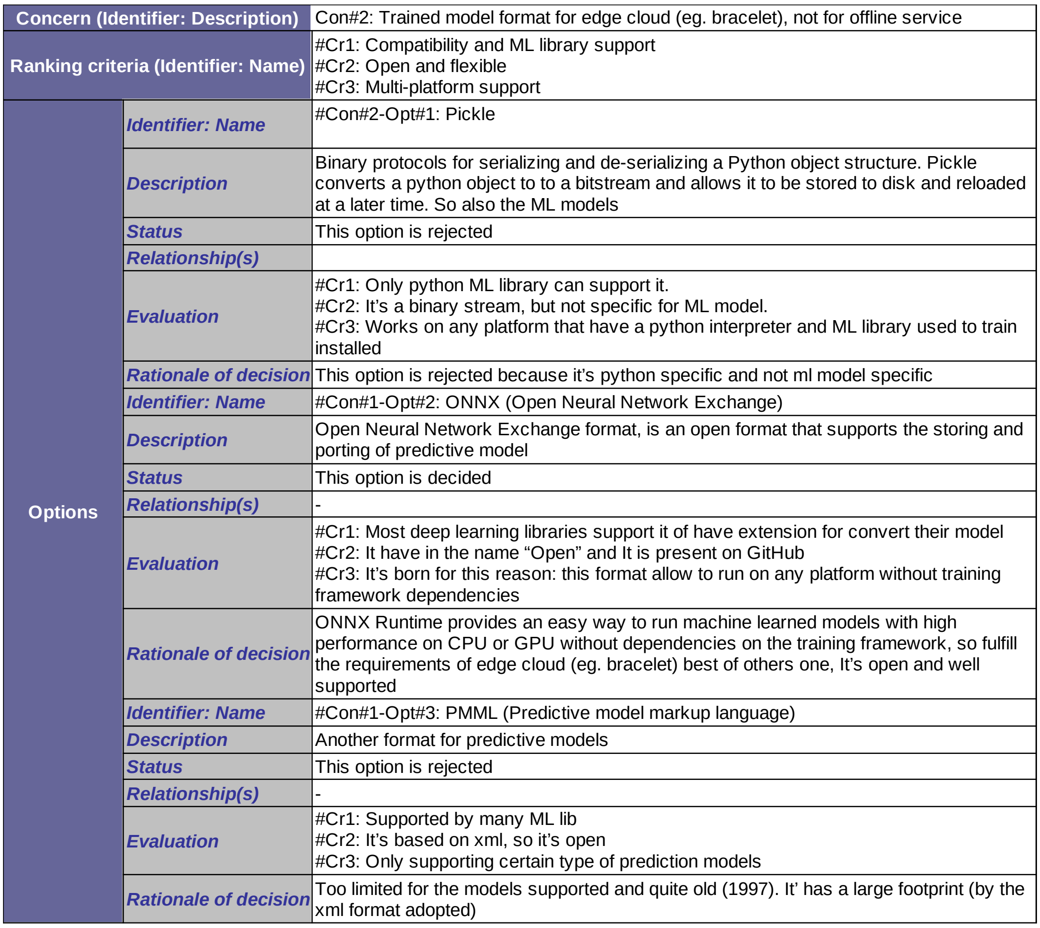
Figure 4.1: First design decision

The important thing to take in mind about this Design Decision, it's the possibility of use our approach on private cloud (on-premise), but also on public (external cloud) or an hybrid choice.

For example deploy the *Online* part on external cloud, there we don't have latency problem (the request is made by REST API by External Users in any case), ad the *Offline* part will be deployed in our private cloud, where we have the issue related to the 1 Tb of data (see non-functional requirements 2.2, the possible DataSets to upload on system.

Other important point is the privacy, our company treat health data and this are real sensible data and not always we can store this data in a external cloud.

4.2 Model format for edge cloud



Concern (Identifier: Description)		Con#2: Trained model format for edge cloud (eg. bracelet), not for offline service
Ranking criteria (Identifier: Name)		#Cr1: Compatibility and ML library support #Cr2: Open and flexible #Cr3: Multi-platform support
Options	*Identifier: Name*	#Con#2-Opt#1: Pickle
	Description	Binary protocols for serializing and de-serializing a Python object structure. Pickle converts a python object to to a bitstream and allows it to be stored to disk and reloaded at a later time. So also the ML models
	Status	This option is rejected
	Relationship(s)	
	Evaluation	#Cr1: Only python ML library can support it. #Cr2: It's a binary stream, but not specific for ML model. #Cr3: Works on any platform that have a python interpreter and ML library used to train installed
	Rationale of decision	This option is rejected because it's python specific and not ml model specific
	Identifier: Name	#Con#1-Opt#2: ONNX (Open Neural Network Exchange)
	Description	Open Neural Network Exchange format, is an open format that supports the storing and porting of predictive model
	Status	This option is decided
	Relationship(s)	-
	Evaluation	#Cr1: Most deep learning libraries support it of have extension for convert their model #Cr2: It have in the name "Open" and It is present on GitHub #Cr3: It's born for this reason: this format allow to run on any platform without training framework dependencies
	Rationale of decision	ONNX Runtime provides an easy way to run machine learned models with high performance on CPU or GPU without dependencies on the training framework, so fulfill the requirements of edge cloud (eg. bracelet) best of others one, It's open and well supported
	Identifier: Name	#Con#1-Opt#3: PMML (Predictive model markup language)
	Description	Another format for predictive models
	Status	This option is rejected
	Relationship(s)	-
	Evaluation	#Cr1: Supported by many ML lib #Cr2: It's based on xml, so it's open #Cr3: Only supporting certain type of prediction models
	Rationale of decision	Too limited for the models supported and quite old (1997). It' has a large footprint (by the xml format adopted)

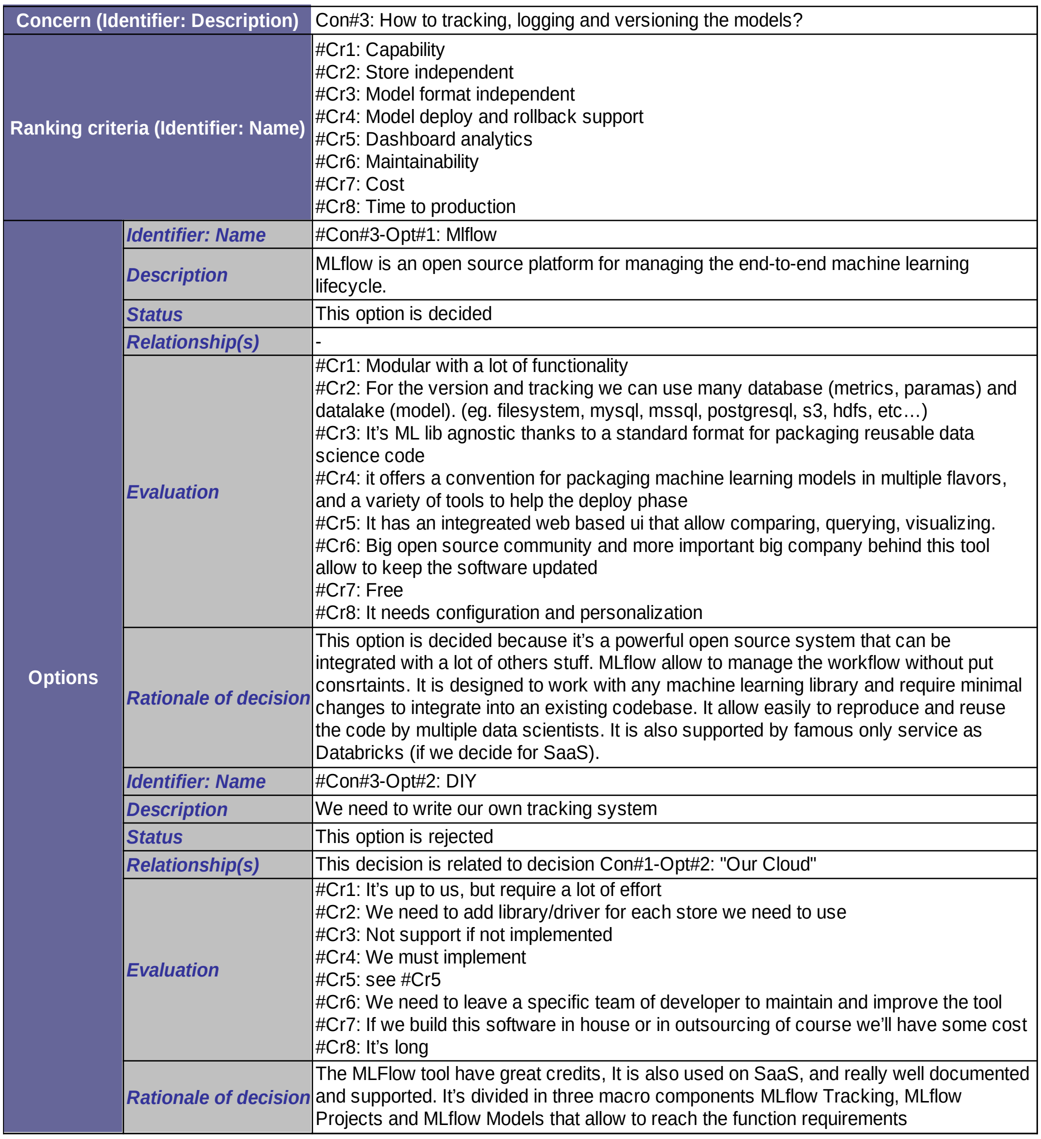
Figure 4.2: Second design decision

4.2.1 ONNX

The ONNX seems was the best, most adopted, open format that allow multi-platform (agnostic) support independently the ML lib used. This is perfect for the edge cloud (the bracelet scenario). Of course we can use others type of formats (libraries specific) on the *Offline* part of our services.

There are also others formats not reported, like POJO and MOJO. They can be used only into java application. They are however very specific to using the H2O's platform.

4.3 Tracking, logging and versioning the models



Concern (Identifier: Description)		Con#3: How to tracking, logging and versioning the models?
Ranking criteria (Identifier: Name)		#Cr1: Capability #Cr2: Store independent #Cr3: Model format independent #Cr4: Model deploy and rollback support #Cr5: Dashboard analytics #Cr6: Maintainability #Cr7: Cost #Cr8: Time to production
Options	*Identifier: Name*	#Con#3-Opt#1: Mlflow
	Description	MLflow is an open source platform for managing the end-to-end machine learning lifecycle.
	Status	This option is decided
	Relationship(s)	-
	Evaluation	#Cr1: Modular with a lot of functionality #Cr2: For the version and tracking we can use many database (metrics, paramas) and datalake (model). (eg. filesystem, mysql, mssql, postgresql, s3, hdfs, etc…) #Cr3: It's ML lib agnostic thanks to a standard format for packaging reusable data science code #Cr4: it offers a convention for packaging machine learning models in multiple flavors, and a variety of tools to help the deploy phase #Cr5: It has an integreated web based ui that allow comparing, querying, visualizing. #Cr6: Big open source community and more important big company behind this tool allow to keep the software updated #Cr7: Free #Cr8: It needs configuration and personalization
	Rationale of decision	This option is decided because it's a powerful open source system that can be integrated with a lot of others stuff. MLflow allow to manage the workflow without put consrtaints. It is designed to work with any machine learning library and require minimal changes to integrate into an existing codebase. It allow easily to reproduce and reuse the code by multiple data scientists. It is also supported by famous only service as Databricks (if we decide for SaaS).
	Identifier: Name	#Con#3-Opt#2: DIY
	Description	We need to write our own tracking system
	Status	This option is rejected
	Relationship(s)	This decision is related to decision Con#1-Opt#2: "Our Cloud"
	Evaluation	#Cr1: It's up to us, but require a lot of effort #Cr2: We need to add library/driver for each store we need to use #Cr3: Not support if not implemented #Cr4: We must implement #Cr5: see #Cr5 #Cr6: We need to leave a specific team of developer to maintain and improve the tool #Cr7: If we build this software in house or in outsourcing of course we'll have some cost #Cr8: It's long
	Rationale of decision	The MLFlow tool have great credits, It is also used on SaaS, and really well documented and supported. It's divided in three macro components MLflow Tracking, MLflow Projects and MLflow Models that allow to reach the function requirements

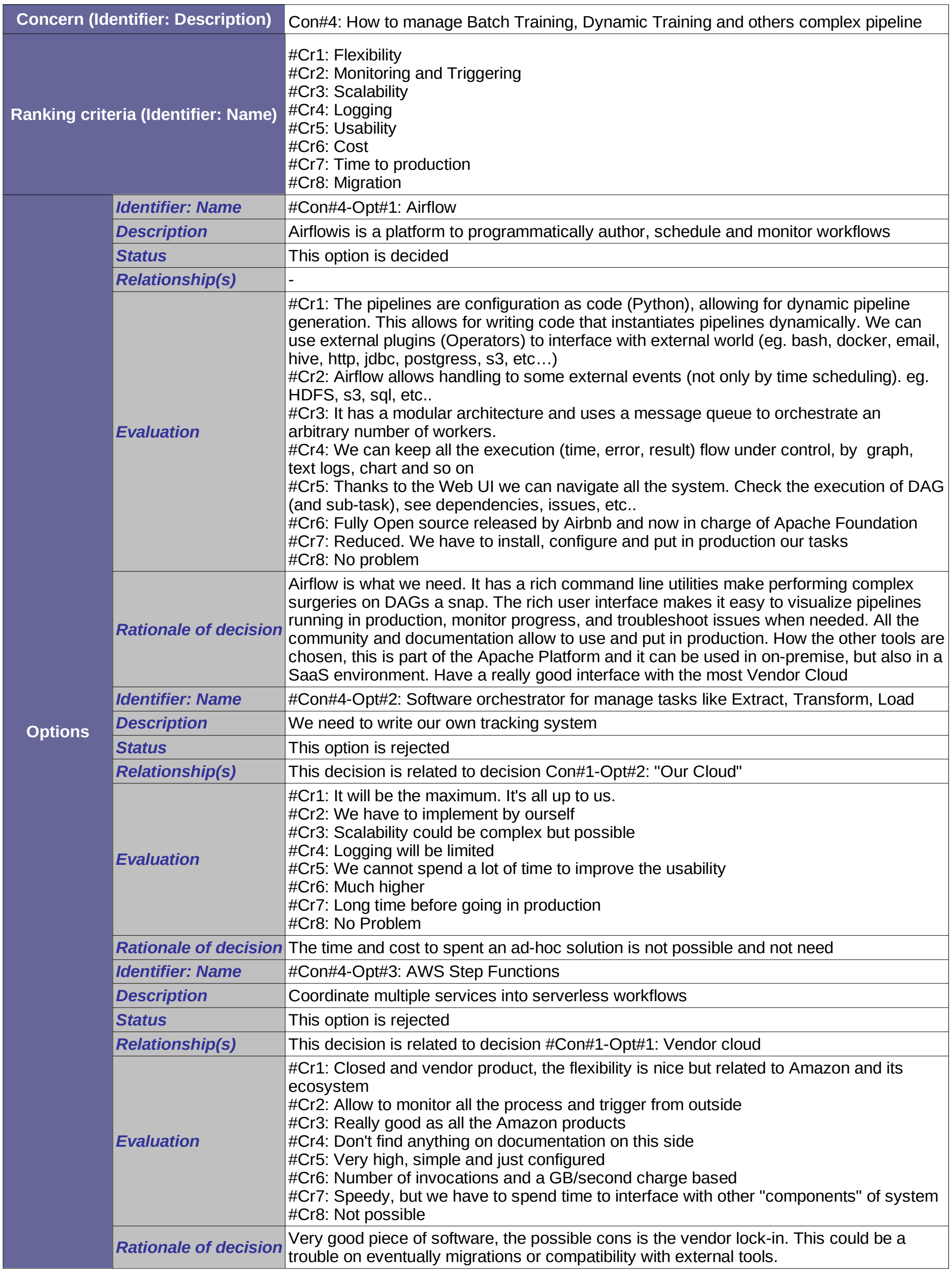
Figure 4.3: Third design decision

4.3.1 MLFlow

MLflow has an API interface and UI for logging parameters, code versions, metrics, and output files when running your machine learning code and for later visualizing the results. It has a standard format for packaging reusable data science code. Each project is simply a directory with code or a Git repository, and uses a descriptor file or simply convention to specify its dependencies and how to run the code.

MLflow provides tools to deploy many common model types to diverse platforms: for example, any model supporting the "Python function" flavor can be deployed to a Docker-based REST server (see functional requirements 2.1), to cloud platforms such as Azure ML and AWS SageMaker (see *#Con#1-Opt#2* hybrid alternative), and as a user-defined function in Apache Spark for batch and streaming inference (the our choice for the *Processing (distributed system).*

4.4 Scheduler and pipeline manager

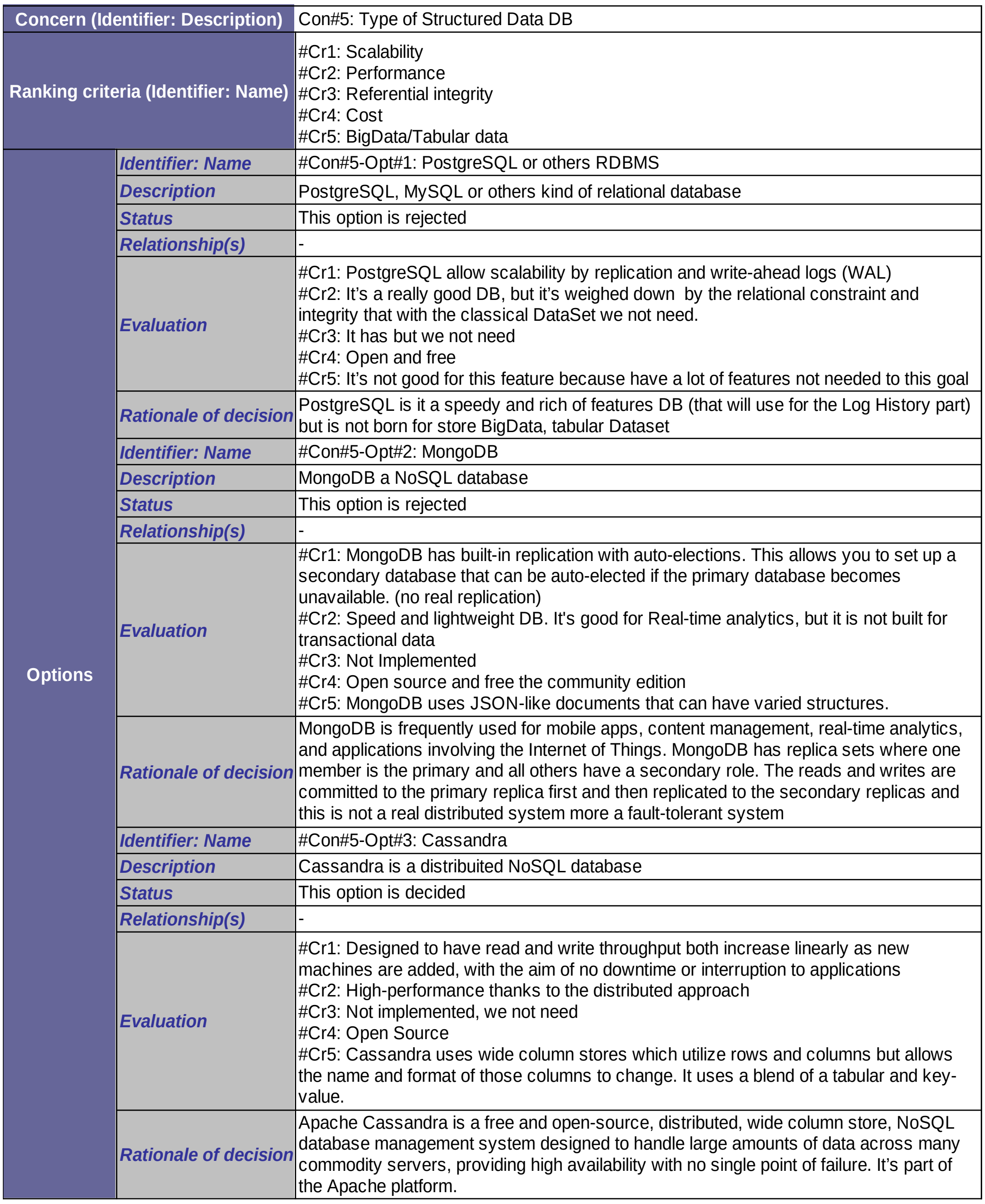


Concern (Identifier: Description)		Con#4: How to manage Batch Training, Dynamic Training and others complex pipeline
Ranking criteria (Identifier: Name)		#Cr1: Flexibility #Cr2: Monitoring and Triggering #Cr3: Scalability #Cr4: Logging #Cr5: Usability #Cr6: Cost #Cr7: Time to production #Cr8: Migration
Options	***Identifier: Name***	#Con#4-Opt#1: Airflow
	Description	Airflowis is a platform to programmatically author, schedule and monitor workflows
	Status	This option is decided
	Relationship(s)	-
	Evaluation	#Cr1: The pipelines are configuration as code (Python), allowing for dynamic pipeline generation. This allows for writing code that instantiates pipelines dynamically. We can use external plugins (Operators) to interface with external world (eg. bash, docker, email, hive, http, jdbc, postgress, s3, etc…) #Cr2: Airflow allows handling to some external events (not only by time scheduling). eg. HDFS, s3, sql, etc.. #Cr3: It has a modular architecture and uses a message queue to orchestrate an arbitrary number of workers. #Cr4: We can keep all the execution (time, error, result) flow under control, by graph, text logs, chart and so on #Cr5: Thanks to the Web UI we can navigate all the system. Check the execution of DAG (and sub-task), see dependencies, issues, etc.. #Cr6: Fully Open source released by Airbnb and now in charge of Apache Foundation #Cr7: Reduced. We have to install, configure and put in production our tasks #Cr8: No problem
	Rationale of decision	Airflow is what we need. It has a rich command line utilities make performing complex surgeries on DAGs a snap. The rich user interface makes it easy to visualize pipelines running in production, monitor progress, and troubleshoot issues when needed. All the community and documentation allow to use and put in production. How the other tools are chosen, this is part of the Apache Platform and it can be used in on-premise, but also in a SaaS environment. Have a really good interface with the most Vendor Cloud
	Identifier: Name	#Con#4-Opt#2: Software orchestrator for manage tasks like Extract, Transform, Load
	Description	We need to write our own tracking system
	Status	This option is rejected
	Relationship(s)	This decision is related to decision Con#1-Opt#2: "Our Cloud"
	Evaluation	#Cr1: It will be the maximum. It's all up to us. #Cr2: We have to implement by ourself #Cr3: Scalability could be complex but possible #Cr4: Logging will be limited #Cr5: We cannot spend a lot of time to improve the usability #Cr6: Much higher #Cr7: Long time before going in production #Cr8: No Problem
	Rationale of decision	The time and cost to spent an ad-hoc solution is not possible and not need
	Identifier: Name	#Con#4-Opt#3: AWS Step Functions
	Description	Coordinate multiple services into serverless workflows
	Status	This option is rejected
	Relationship(s)	This decision is related to decision #Con#1-Opt#1: Vendor cloud
	Evaluation	#Cr1: Closed and vendor product, the flexibility is nice but related to Amazon and its ecosystem #Cr2: Allow to monitor all the process and trigger from outside #Cr3: Really good as all the Amazon products #Cr4: Don't find anything on documentation on this side #Cr5: Very high, simple and just configured #Cr6: Number of invocations and a GB/second charge based #Cr7: Speedy, but we have to spend time to interface with other "components" of system #Cr8: Not possible
	Rationale of decision	Very good piece of software, the possible cons is the vendor lock-in. This could be a trouble on eventually migrations or compatibility with external tools.

Figure 4.4: Fourth design decision

notes: more info on Airflow can be found in *Component Diagram* chapter (see 6.1) and *Airflow* chapter (see 8.5.6).

4.5 Structured Data DB



Concern (Identifier: Description)		Con#5: Type of Structured Data DB
Ranking criteria (Identifier: Name)		#Cr1: Scalability #Cr2: Performance #Cr3: Referential integrity #Cr4: Cost #Cr5: BigData/Tabular data
Options	*Identifier: Name*	#Con#5-Opt#1: PostgreSQL or others RDBMS
	Description	PostgreSQL, MySQL or others kind of relational database
	Status	This option is rejected
	Relationship(s)	-
	Evaluation	#Cr1: PostgreSQL allow scalability by replication and write-ahead logs (WAL) #Cr2: It's a really good DB, but it's weighed down by the relational constraint and integrity that with the classical DataSet we not need. #Cr3: It has but we not need #Cr4: Open and free #Cr5: It's not good for this feature because have a lot of features not needed to this goal
	Rationale of decision	PostgreSQL is it a speedy and rich of features DB (that will use for the Log History part) but is not born for store BigData, tabular Dataset
	Identifier: Name	#Con#5-Opt#2: MongoDB
	Description	MongoDB a NoSQL database
	Status	This option is rejected
	Relationship(s)	-
	Evaluation	#Cr1: MongoDB has built-in replication with auto-elections. This allows you to set up a secondary database that can be auto-elected if the primary database becomes unavailable. (no real replication) #Cr2: Speed and lightweight DB. It's good for Real-time analytics, but it is not built for transactional data #Cr3: Not Implemented #Cr4: Open source and free the community edition #Cr5: MongoDB uses JSON-like documents that can have varied structures.
	Rationale of decision	MongoDB is frequently used for mobile apps, content management, real-time analytics, and applications involving the Internet of Things. MongoDB has replica sets where one member is the primary and all others have a secondary role. The reads and writes are committed to the primary replica first and then replicated to the secondary replicas and this is not a real distributed system more a fault-tolerant system
	Identifier: Name	#Con#5-Opt#3: Cassandra
	Description	Cassandra is a distribuited NoSQL database
	Status	This option is decided
	Relationship(s)	-
	Evaluation	#Cr1: Designed to have read and write throughput both increase linearly as new machines are added, with the aim of no downtime or interruption to applications #Cr2: High-performance thanks to the distributed approach #Cr3: Not implemented, we not need #Cr4: Open Source #Cr5: Cassandra uses wide column stores which utilize rows and columns but allows the name and format of those columns to change. It uses a blend of a tabular and key-value.
	Rationale of decision	Apache Cassandra is a free and open-source, distributed, wide column store, NoSQL database management system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure. It's part of the Apache platform.

Figure 4.5: Fifth design decision

4.5.1 Cassandra

One of Cassandra's biggest strengths is being able to handle massive amounts of (un)structured data and rapidly scale with minimal increase of administrative work.
It is used by Instagram to handle roughly 80 million photos uploaded to the database every day.

The Cassandra data model is a schema-optional, column-oriented data model. This means that, unlike a relational database, you do not need to model all of the columns required by your application up front, as each row is not required to have the same set of columns.

A brief comparison on write operation is taken from datastax:

Nodes	Cassandra	MongoDB
1	18,683.43	8,368.44
2	31,144.24	13,462.51
4	53,067.62	18,038.49
8	86,924.94	34,305.30
16	173,001.20	73,335.62

notes: others design decision involved during the design process are concern the *Distributed vs Load Balancer* approach and the *Online and Offline* parts, but they are omitted due to the limit of (5) dd reached.

4.6 Tools

After research and Design Decisions our system will have this list of tools:

- [1]**Mlflow**
- [2]**Apache kafka**
- [3]**Apache Ignite**
- [4]**Apache Cassandra**
- [5]**Apache Spark**
- [6]**Apache Hadoop**
- [7]**Apache Airflow**

- [8]**Docker**
- [9]**Nginx**
- [10]**PostgresSQL**
- [11]**JupyterLab**

notes: the tools are just described partially in this chapter and there will be some information on the remain ones in the *Component Diagram* chapter (see 6.1). At the end, in the *Docker-compose services (tools)* (see 8.5) you will find more detailed information.

Chapter 5

Views and Viewpoints

5.1 Stakeholders

- **Core developer:** develops the system components.
- **System Integrator**: responsible for system integration (how the componets interacts and communicate).
- **Data Scientist:** write AI algorithms and build the AI models.
- **Production Engineers**: improves/fixes the source code of Data Scientist.
- **Data Engineer:** defines data workflow and pipelines.
- **Database Developer**: designs and administrates database (structured data).
- **Developers AI-powered solution**: develops app powered by our models.
- **External User**: the end user wearing the device and use the app made by the Developers building AI-powered solution consuming models (eg. bracelet).

5.2 Concern-Stakeholder Traceability

Table 5.1: Concern-Stakeholder Traceability 1

	Core developer	**System Integrator**	**Data Scientist**	**Production Engineers**
Performance	X	X		X
Security	X	X		
Cost	X	X	X	X
Networking & Communication	X	X		
Data Analysis			X	X
Response Time	X	X		X
Dependability	X	X	X	X
Usability				
Scalability	X	X	X	X
Energy Consumption	X			
Configurability	X	X		

Table 5.2: Concern–Stakeholder Traceability 2

	Data Engineer	**Database Developer**	**Developers AI-powered solution**	**External User**
Performance	X	X		
Security	X	X		
Cost	X	X	X	
Networking & Communication	X			
Data Analysis	X			
Response Time	X	X		
Dependability	X	X	X	
Usability	X		X	X
Scalability		X		
Energy Consumption			X	
Configurability	X			

Chapter 6

UML Static and Dynamic Architecture View

6.1 Component Diagram

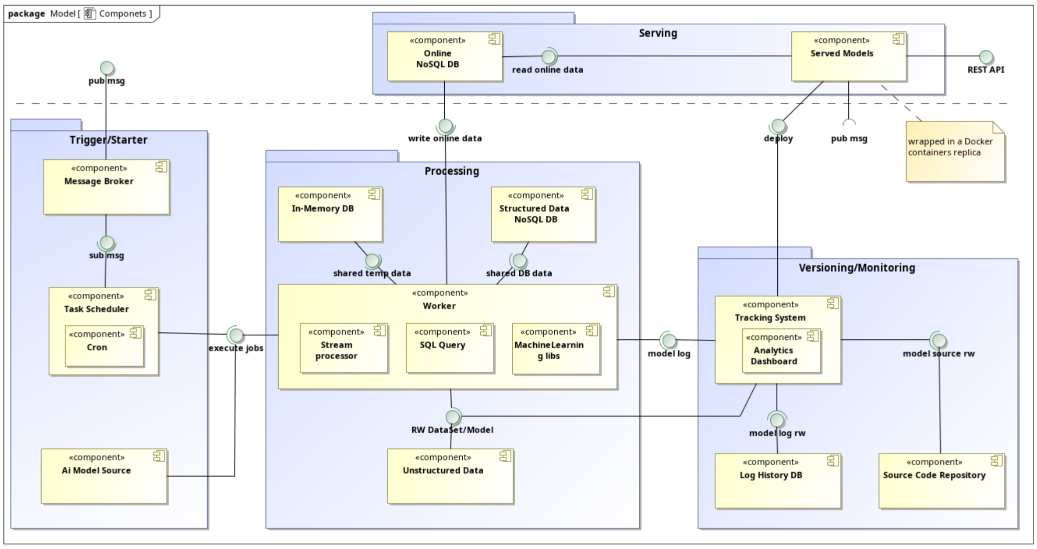

package Model [Componets]
Serving
«component» Online NoSQL DB
read online data
«component» Served Models
REST API
pub msg
Trigger/Starter
write online data
deploy
pub msg
wrapped in a Docker containers replica
Processing
«component» Message Broker
«component» In-Memory DB
«component» Structured Data NoSQL DB
sub msg
shared temp data
shared DB data
Versioning/Monitoring
«component» Worker
«component» Task Scheduler
«component» Cron
«component» Stream processor
«component» SQL Query
«component» MachineLearning libs
execute jobs
model log
«component» Tracking System
«component» Analytics Dashboard
model source rw
RW DataSet/Model
model log rw
«component» Ai Model Source
«component» Unstructured Data
«component» Log History DB
«component» Source Code Repository

Figure 6.1: Component diagram.

6.1.1 Components Description

Part of this components was already described in the *Sub-systems* section, see 3.2 for more information.

In the below section we will see the components by packages.

6.1.2 Trigger/Starter

This package contains the components that can start a AI workflow.

- **Message Broker:** this component will be the Apache Kafka. It's implement the Publish/Subscribe. All components are allowed to publish/subscribe a message. (see 3.2 for more information)

- **Task Scheduler:** this component allow to programmatically (*Cron)* author, schedule, and monitor workflows. When workflows are defined as code, they become more maintainable, versionable, testable, and collaborative. It use Apache Airflow to create workflows as directed acyclic graphs (DAGs) of tasks.

- **AI model source:** this component is a web-based interactive development environment, but allow also simple plain-text upload. It's flexible and extensible. Allow to write the AI algorithms.

6.1.3 Processing (distributed system)

Here will be hosted the distributed resources (compute and space).

- **Worker:** this components allow heavy jobs, if the Data Scientist need. Users can use Apache Spark as distributed general-purpose cluster-computing framework. It Allows data parallelism and fault tolerance. It's not mandatory, the user can decide to use simple Python or R programming. The develops can base his AI algorithms on these sub-compoents:

  - **Stream processor:** allow data stream processing.
  - **SQL Query:** seamlessly mix SQL queries with AI algorithms.
  - **Machine Learning libs:** the Data Scientists can choose the preferred Machine Learning library (Spark MLlib, Sklearn, Tensorflow, Keras) to write them code, but only the Spark Mlib allow to exploits the cluster-computing.

- **In-Memory DB:** a distributed and horizontally scalable in-memory database (IMDB), most probably Apache Ignite. It's a speed (and volatile) Database shared along all workflows.

- **Structured Data NoSQL DB:** a shared structured database that help job execution (eg. Cassandra, MongoDB).

- **Unstructured Data:** shared persistence space for DataSets or Artifacts (eg. trained model). It's like Amazon S3 or Google Cloud storage.

6.1.4 Versioning/Monitoring

This package contains the components for maintain the the chronological history of the trained models.

- **Tracking System:** Tracking AI experiments to record and compare parameters and results (ie. link source code, training set, result metrics, parameters used to running the code). Allow easily model rollback. Packaging the code in a reusable, reproducible form in order to share with other Data Scientists and deploy. Provide a web-based ui to do the Analytic part. This will use the MLflow open source platform.

- **Log History DB:** where the models analytic will be saved (RDBMS).

- **Source code Repository:** one word, Git (or other version-control system).

6.1.5 Serving (external)

This package contains the components exposed to the public (Online) services.

- **Served Models:** the trained model will be deployed here, wrapped inside a container (ie. Docker) and it exposes a public REST API. It have an utilization interface (*pub msg*) where it publish the arrived features collected from the REST API. This will be stored for future use.

- **Online NoSQL DB:** some ML inference of our model can need additional information not present in the REST API payload, here will be selected and stored this information (eg. user profile information).

6.2 Sequence Diagram

6.2.1 Model to Production

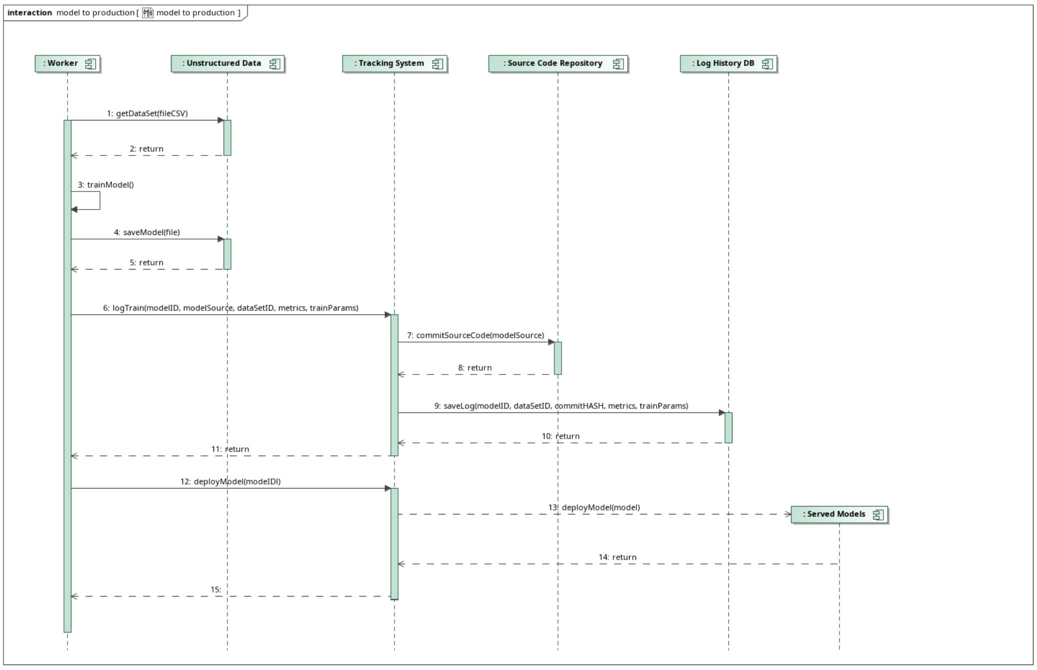


Figure 6.2: Sequence diagram of *Model to Production* use-case

In this sequence diagrams we see the flow that build a trained model, versioning and deploy by Served Models. First of all the *Workers* retrieve the DataSet from *Unstructured Data*, the model is "fitted" (ie. trained against the DataSet) and is saved in the desired format, always in the *Unstructured Data*.
At this point start the versioning phase by a call to the component *Tracking System*. The source code is committed in the *Source Code Repository* and after this it saves all the information related to the training process (*modelID, dataSetID, commitHASH, metrics, trainParams*) in the *Log History DB*, for tracking and analytic finality.
Last step it's the model deploy (if requested), where the *Served Models* (ie. the docker container) is created, with a features mapped REST API interface implemented. It returns to worker if all is done, and the next task of our workflow (if exist) can be executed.

6.2.2 Batch Training

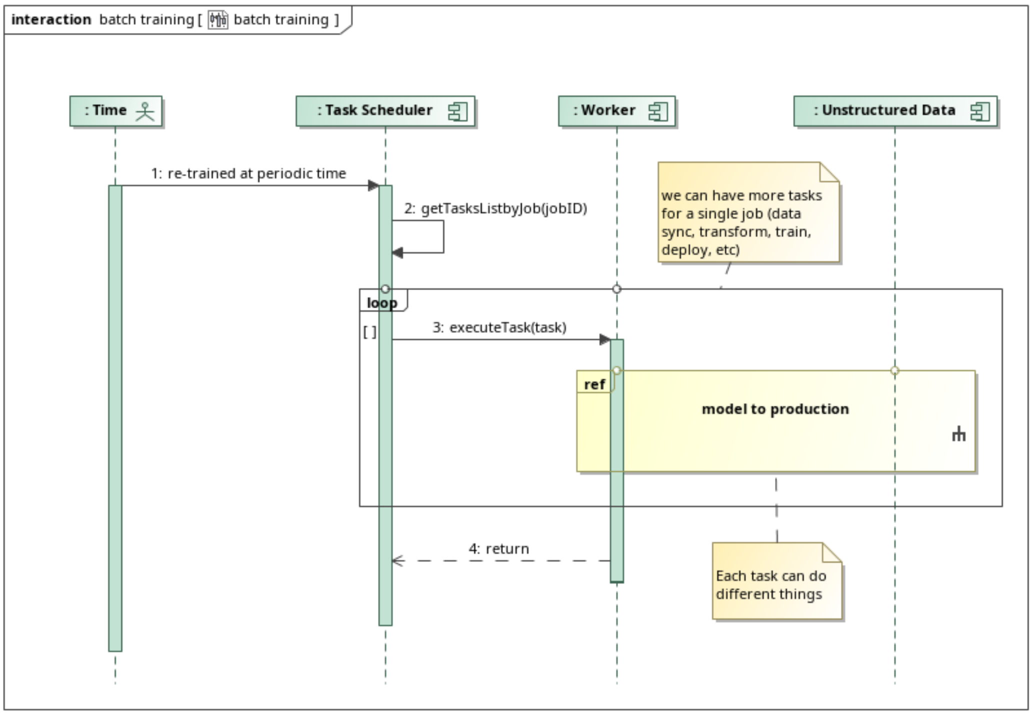

interaction batch training [batch training]
: Time
: Task Scheduler
: Worker
: Unstructured Data
1: re-trained at periodic time
2: getTasksListbyJob(jobID)
we can have more tasks for a single job (data sync, transform, train, deploy, etc)
loop
[ ]
3: executeTask(task)
ref
model to production
4: return
Each task can do different things

Figure 6.3: Sequence diagram of *Batch Training* use-case

The diagrams is triggered by *Timer* actor, this because the *Batch Training* must be executed at periodic time intervals. The *Task Scheduler* invoked know what and how execute the related tasks (thanks to internal DAG associated to the workflow). The Loop represent the list of tasks belong to the workflow, executed trough the Worker (ref statement in the sequence).

6.2.3 Dynamic Training

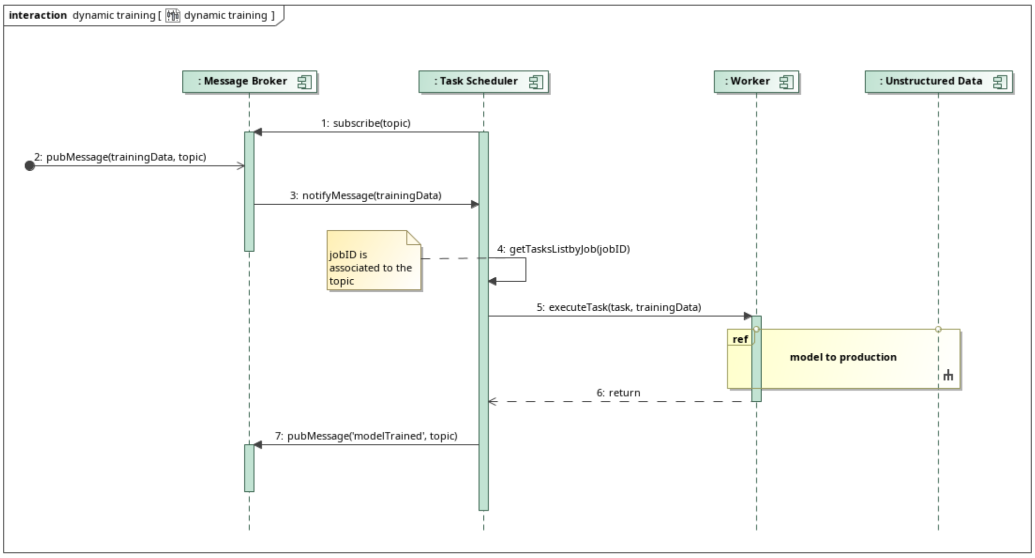

interaction dynamic training [dynamic training]
: Message Broker
: Task Scheduler
: Worker
: Unstructured Data
1: subscribe(topic)
2: pubMessage(trainingData, topic)
3: notifyMessage(trainingData)
jobID is associated to the topic
4: getTasksListbyJob(jobID)
5: executeTask(task, trainingData)
ref
model to production
6: return
7: pubMessage('modelTrained', topic)

Figure 6.4: Sequence diagram of *Dynamic Training* use-case

In this case the training is done on-demand, so the DataSet (also a single line) arrive by a message published to the *Message Broker* and thanks to a subscription of the *Task Scheduler*, it can start the workflow with the provided data.

The flow is the same of *Batch Training*, but in this case (for example) the workflow is made up a single task (no loop) and at the end of the training process the *Task Scheduler* publish on the right topic the "end of the works".

notes: the missing sequence diagrams are less significant

Chapter 7

First Deliverable

7.1 Questions

1. Identify a potential target reference architecture for the AI system that takes into account the key functionalities and the constraints of the system Identify the ASR.
   (a) Identify the ASR
   (b) Identify the architecture style/styles that can be used in the scenario
2. Evaluate the pros and cons of the following choices for using the model for prediction/ recommendation (Scoring):
   (a) Rest API (each scoring of the model is an invocation to an API Call)
   (b) In app (the application holds the trained model and performs the scoring when requested)
   (c) In database scoring: model is pushed in the database that massively scores it on the relevant dataset
   (d) Asynchronous via messaging : Implement a pub sub architecture where a stream of messages will be evaluated on the fly

7.2 Answering by points

7.2.1 ASR and architecture styles

We have tried to answer the *1.a* and *1.b* in the all above chapters using the template provided.

7.2.2 Pros and Cons of 4 model scoring style

About the *2.* we have tried to compare the 4 different approach below:

7.2.2.1 REST API

The models is put in production as web-services. This allow using a unique (remote) interface for different kind of application: web, desktop and mobile (eg. bracelet).
The request can be done Async, so the client can do something others waiting the (remote) model make the prediction (inference).
For sure we have some delay (latency) due the transmission layer (ie. connection) used:

$$send\_data + make\_prediction + receive\_response$$

The prediction could be an heavy process and not all device have the computational power to do it, so delegate this task remotely can be an advantage.

We have two different way:

- Not all features needed for make the prediction are in the payload (something could be in a centralized DB), so we can send an unique identifier and the supplementary data will be pulled from a DB.
- All the features are in the payload of message.

The choice can depend from the weight of the features or if these are available to the client.

We think is also easy to implement a caching system.
For implement a REST API we can also use FaaS or a docker container.

7.2.2.2 In app

This is what our system does in the *Offline* part, but can be also used for remote device for implement the edge cloud, thanks to specific hardware called edge accelerator (eg. on bracelet).

In this case the latency is due only to the inference time of the model running locally (ie. no transmission layer), and this can be mandatory in application that need "instant" response (eg. a self driving car that use AI to object recognition a take decision based on this).

In this case the app need the sufficient computational resource to run the prediction model.

The reason for adopt this way could be also related to the weight of the data to be send to make the prediction and at the end for legal reason or privacy requirements that don't allow storing data outside the application. Of course, local model allows the computation also without a connection.

In this case we have the issue to update the model, when a new, well trained one is released. In this last case can be useful a Pub/Sub pattern that notify the app when a new model is available.

There are different formats for store the models (see 4.2 Model format for edge cloud)

7.2.2.3 In database scoring

We have put a question about this point on the shared document, and it is still not very clear.

Pushing the model in the database that massively scores it on the relevant dataset can trigger a internal function (PostgreSQL, Cassandra, MongoDB support triggers for example) that can score particular field/row of the Database and write the result in the DB or somewhere else.

This can be useful if you store a ML model in a DB (eg. Oracle support PMML) and automatically the DB, thanks to the trigger features, links to this model (ie. writing in the DB) all the relative metrics (eg. accuracy) about that model. This allows to compare all the models and keep update the relative information.

In our point of view, an alternative scenario that involve the DB could be:

- A new row is written in the DB (eg. new film added to my "preferred list").
- The related trigger function is executed.
- This function invokes a prediction (through the predefined model) based on the new incoming data (eg. by the new updated "preferred list" of films).

- The prediction is used to update the data on the DB (eg. a new "suggestion list" of film based on my updated "preferred list").

In this case the triggering event is the data written/updated in the DB and the model used for prediction can be or not in the DB.

This approach allow a kind of real-time information update on DB, keep in sync on the triggered modification of the dataset.

7.2.2.4 Asynchronous via messaging

Asynchronous can be also the REST API (client/server point of view), but for each request of the client correspond a single answer (at certain time) from the server and the communication is point to point.

The Pub/Sub architecture instead not need the polling way to notify the client that something has happened, but it use an event-driven philosophy. It's enough the clients subscribe a specific topic and all the related notifications in this regard will be notified when they occur. This is called near real-time.

This allow the creation of a single feed channel (the topic), to which the different consumers can subscribe.

So in the bracelet scenario for example, one subscriber could be prediction model and another one, on the same event and same payload, transform and store the features (provided for the prediction) in a DB.

The Pub/Sub pattern allow easily scalability and queues/priority management.

Chapter 8

From architecture to code

8.1 Introduction

The system architecture described in all the previous chapters has been implemented brick on brick.

The our design decision to use a Hybrid Cloud (4.1), give us the maximum flexibility on "what deploy where". This means that one component (part of the architecture) could be a SaaS (on external Cloud) or installed and configured from scratch on-premise.

As already explained the target objective is to deploy the Offline part on-promise and the Online part on the external public cloud (for the reason already discussed).
It can happen that at a certain point we need more computation power (Processing Workers), it will be easy to buy external Spark Workers (eg. Azure Databricks, Google Dataproc) and add it to our Spark cluster for distributed parallel processing.

For allowing this policy we have to install and configure each component of the system by ourself, this allow to deploy or migrate elsewhere (eg. IaaS, PaaS, SaaS) in future.

We have used the *docker* environment to bring up the entire prototype. In particular, *docker-compose* to create the tools infrastructure.

8.2 Online Resources

All the source code is available to the Github repository:

GitHub repository

In the git repository, we can find the prototype final architecture, docker images, configurations, glue codes, example ml model, etc...

notes: The repository has restricted access at this time. The source code can be retrieved also on GDrive.

A brief video that shows the prototype running locally is available here:

youtu.be/8jNK6N-cPJA

The system is also deployed on the Google Compute Engine, the IaaS offered by Google:

yamlops.uk.to

notes: The Jupyter password for all user is: *sa1920*.
The hardware is the basic one (a mono-core with 3.75 Gb RAM), due to this the system cannot be speed.

The deploy on a Public Cloud is to allow access to a real instance of the system and also to demonstrate the deploy flexibility of the system that guarantees the Hybrid Cloud approach.

8.3 Folder structure

The source code directory structure is the following:

- **docker/airflow/dags/**: contains the Airflow Task to be executed
- **dfs**: simulate a DistributedFileSytem over the components
- **docker**: contains all the docker images need a building (ie. Dockerfile), the configuration, start-up script, etc...

- **user-workspace**: the mounted personal folder (containing the experiments) of the Data Scientists
- **docker-compose.yml**: describe the whole system component by component (ie. services)

8.4 Installation

For build and run-up the entire system we only need *docker-compose*, the project source code and following the below steps:

- **cd /path/to/cloned/sa1920-project**: go to the project root directory
- **docker-compose build**: only the first time
- **docker-compose up**: to run the whole system

Wait some minutes and...That's it!

8.5 Docker-compose services (tools)

Let's see together the whole system (aka YAMLOps), exploring the "*docker-compose.yml*" file.

Now we try to describe every single service and how it fulfils the functional and non-functional requirements. We will describe briefly the services. For detailed information please refer to every single folder (ie. "*docker/<service_name>*).

notes: some variables used above are set in the "*.env*" file.

8.5.1 Nginx

```
nginx:
    image: nginx
    container_name: nginx-container
    restart: always
    volumes:
        - ./docker/nginx/nginx.conf:/etc/nginx/conf.d/nginx.conf
        - ./docker/nginx/html:/usr/share/nginx/html
    environment:
        USER_USER: ${USER_USER}
        USER1_USER: ${USER1_USER}
        USER2_USER: ${USER2_USER}
        USER3_USER: ${USER3_USER}
    depends_on:
        - adminer
        - airflow
        - mlflow
        - jupyter
    command: /bin/bash -c "envsubst '$$NGINX_HOST $$NGINX_PORT $$USER_
        USER $$USER1_USER $$USER2_USER $$USER3_USER' < /etc/nginx/conf.d/
        nginx.conf > /etc/nginx/conf.d/default.conf && exec nginx -g '
        daemon off;'"
    ports:
        - "${NGINX_PORT}:80"
```

Ngnix is not strictly part of our architecture, but it is used as a reverse proxy.

It allows serve the landing page ("*docker/nginx/html*") and route each service, that exposes a web interface, on a single port and different path. It can be also used to provide a basic authentication layer if needed. The webserver/reverse proxy configuration is stored in "*docker/nginx/nginx.conf*".

The reverse proxy does the following service web ui mapping:

- **/ (root)**: landing page
- **/mlflow**: MLFlow as *Tracking System*
- **/airflow**: Airflow as *Task Scheduler*
- **/adminer**: PostgresSQL web ui client
- **/spark-master** and **spark-worker-1**: Spark Cluster

- **/user/emanuele**, **/user/henry**, **/user/davide**, **/user/karthik**: personal JupyterLab workspace. The user can access to the classic Jupyter by append "**/tree**" to the path.

8.5.2 PostgreSQL

```
postgres:
    build: './docker/postgres'
    container_name: postgres-container
    restart: always
```

PostgreSQL is our Relational Database Management System, it will be used as the *Log History Database* for the model versioning purpose.
MLflow will save here all the ML performed experiment, params, metrics and links togheter the models, source code, datasets and others artifacts.

It's clear that for doing this we need a DB like PostgreSQL to preserve the referential integrity and the ACID philosophy, typically of the RDMS.
The same DB is used as the persistence layer of Airflow, it will maintain the Tasks information, dependency and execution of DAGs.

notes: for more information on the initial start-up, configuration refers to "*docker/postgres*" folder (eg. "*init.sql*" will create the DB, user, and access policy for MLflow and Airflow).

8.5.3 Adminer

In the "*docker-compose.yml*" there is also an Adminer service, this is a web UI client for PostgreSQL (CRUD, query and manage).

```
adminer:
    image: adminer
    container_name: adminer-container
    restart: always
    depends_on:
        - postgres
    environment:
        ADMINER_PLUGINS: 'frames'
    ports:
        - "${ADMINER_PORT}:8080"
```

8.5.4 Cassandra

How explained in chapter *Structured Data DB* (design decision 5, see 4.5) we have another type of data to be stored and managed: the DataSet for the Machine Learning training.

Apache **Cassandra** component fulfils the *Structured Data* DB in the *Processing* boundary. It allows for handling massive amounts of (un)structured data as schema-optional, column-oriented data.

It's not directly used by any tools but will be utilized by the Data Scientist through the model source code, or in the Airflow Dags to store new data (DataSet) for future reference.

The access is made by the python cassandra-driver installed in Jupyter and Airflow components by "*requirements.txt*".

```
cassandra:
    image: cassandra:3.11.5
    container_name: cassandra-container
    restart: always
```

8.5.5 Ignite

Apache **Ignite** is the latest DB of our system, but in this case, it is an in-memory data-store. It's a distributed database, used for caching, designed to store and share speedily large volumes of data across a cluster of nodes and Data Scientists Jupyter workspace.

```
ignite:
    image: apacheignite/ignite:2.7.6
    container_name: ignite-container
    restart: always
    environment:
        IGNITE_QUIET: 'false'
    restart: always
```

8.5.6 Airflow

Airflow is our *Task Scheduler*, as we have already said "depend_on" PostgreSQL for the persistence layer. It is our orchestrator, not only a simple cron.

It executes Tasks as DAG (DirectedAcyclicGraph) defined by the Data Engineer through configuration as code (Python) at a scheduled time or triggered events.

```
airflow:
    build: './docker/airflow'
    container_name: airflow-container
    restart: always
    depends_on:
        - postgres
    environment:
        LOAD_EX: y # pre-load airflow dag examples
        AIRFLOW_CONN_POSTGRES_DEFAULT: 'postgres://airflow:
            airflow@postgres:5432/airflow'
        AIRFLOW__WEBSERVER__BASE_URL: 'http://localhost:8080/airflow'
        MLFLOW_TRACKING_URI: 'http://mlflow:8080'
    volumes:
        - ./dags:/usr/local/airflow/dags
        - ./docker/airflow/plugins:/usr/local/airflow/plugins
        - ./dfs:/usr/local/airflow/dfs
        - ./dfs/mlflow:/mlflow
    ports:
        - "${AIRFLOW_PORT}:8080"
```

The DAGs are saved in "*dags*" folder, that will be injected/mounted when the docker image start and loaded by Airflwo.
Airflow can execute (train or predict) MLflow project thanks to the environment variable "*MLFLOW_TRACKING_URI*" that point to the server.

All the python libs supported are listed in the requirements.txt.

Airflow has access to DFS (DataLake), by the mounted volume ("*./dfs:/usr/local/airflow/dfs*"). It allows access to DataSets, Models and Artifacts history.

notes: In the prototype, the DFS (DistributedFileSystem) is simulated by the volume mounting feature provided by docker. We thought no need another service in the prototype phase (ie. there is not 1 TB files).
In the production system, we suggest the HDFS (Hadoop Distributed File System) a distributed file-system that stores data providing very high aggregate bandwidth across the cluster, highly fault-tolerant and also on low-cost hardware.

8.5.7 MLflow

MLflow is a core part of the system. It allows all the *Tracking System* part of the system, but it also allows to analyze and compare each model by params and metrics.

Thanks to it, will be also easy to distribute the model both as source code model (for training) as the final model (to serve and predict).

```
mlflow:
    build: './docker/mlflow'
    container_name: mlflow-container
    restart: always
    depends_on:
        - postgres
    volumes:
        - ./dfs/mlflow/artifacts:/mlflow/artifacts
    command: 'mlflow server --backend-store-uri postgresql://mlflow:
        mlflow@postgres:5432/mlflow --default-artifact-root /mlflow/
        artifacts --host 0.0.0.0 --port 8080 --static-prefix /mlflow'
    ports:
        - "${MLFLOW_PORT}:8080"
```

notes: in the *command* we see the definition of *backend-store-uri* and *default-artifact-root*, respectively our PostfresSQL and the DataLake for store DataSets, models and artifacts.

MLFlow support may store (mysql, mssql, sqlite, postgresql, databricks, http) and DataLake (Amazon S3, Azure Blob Storage, Google Cloud Storage, FTP, SFTP, NFS, HDFS).

It's an amazing project that allows to automatically create the environment need to run a machine learning experiment (create conda env and fulfil the libraries dependencies), as a packet manager.

This allows each Data Scientist, and not only, to replicate the experiment (eg. model training), only with the git repository, without taking care of the ML library used.

```
mlflow run https://github.com/elbowz/dummy-bracelet4yamlops.git -P
    kernel=rbf -P degree=5
```

or locally

```
mlflow run /path/to/project -P kernel=rbf -P degree=5
```

It can also deploy a REST service, wrapping (also in a docker image) the predicting model trained, stored and versioned in any model format saved (eg. ONNX, PMML, Pickle and other library dependants formats).
This allows speedy deployment (*Served models*) and rollback of any pre-trained models without source code or complicated procedure.

```
mlflow models serve -m runs:/c0844c7cb78040b38dcc30a932d73a9f/model -p
    5000
```

MLflow allow the metrics, params, model and artifact recording by very simply API:

```
# Log parameters
mlflow.log_param("kernel", kernel)
mlflow.log_param("degree", degree)

# Log metrics
mlflow.log_metric("Accuracy", accuracy_test_set)
mlflow.log_metric("Accuracy TrainingSet", accuracy_training_set)
mlflow.log_metric("mae", mae)
mlflow.log_metric("rmse", rmse)
mlflow.log_metric("r2", r2)
mlflow.log_metric("mae", mae)

# Add tag to MLflow log
mlflow.set_tag('model', 'SVM')
mlflow.set_tag('stage', 'prod')

# Add DataSet to artifacts
mlflow.log_artifact('./cleveland.csv')

# Save model
mlflow.sklearn.log_model(classifier, "model")
```

8.5.8 Kafka and Zookeeper

Kafka is the Message broker that implements the Pub/Sub pattern.
Zookeeper is needed for distributed coordination (ie. replication and scalability) of Kafka. We use it for Data Ingestion and increase the DataSets from new incoming data (eg. from Served models), but can be also used to start new training, data transformation or deploy some models by message

event-driven approach. It will be directly connected, by topic subscription to Airflow.

```
zookeeper:
    image: wurstmeister/zookeeper
    container_name: zookeeper-container
    restart: always

kafka:
    image: wurstmeister/kafka
    container_name: kafka-container
    restart: always
    environment:
        KAFKA_ADVERTISED_HOST_NAME: kafka
        KAFKA_CREATE_TOPICS: 'TopicA:1:1' # create a topic called 'TopicA
            " with 1 partition and 1 replica
        KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
        KAFKA_LISTENERS: 'PLAINTEXT://:9092'
        KAFKA_ADVERTISED_LISTENERS: 'PLAINTEXT://kafka:9092'
    volumes:
        - /var/run/docker.sock:/var/run/docker.sock
    ports:
        - "${KAFKA_PORT}:9092"
```

8.5.9 Jupyter

Jupyter is the *AI Model Source Web IDE.* We have chosen the JupyterLab version and installing the Git extension (refer to Dockerfile). The Data Scientist can still use the normal version if he prefers.

JupyterLab has many features respect to base version, as an example allow access to the machine shell by a simulated web terminal. So the user is free to run or configure his workspace as he prefers.
It has also a Git simple UI interface for simplifying the interaction with the Source repository.

Part of the supported ML libraries are listed in the "*requirements.txt*":

```
kafka-python==1.4.7
Keras==2.3.1
mlflow==1.4.0
pyspark==2.4.4
cassandra-driver==3.20.2
pyignite==0.3.4
numpy==1.17.4
pandas==0.25.3
scikit-learn==0.22
scipy==1.3.3
tensorflow==1.15.0
tensorboard==1.15.0
```

- **kafka-python**: kafka client to publish and subscribe with the broker
- **keras, scikit-learn, tensorflow, tensorboard**: the most famous and utilized ML librearies
- **numpy, pandas**: DataSet manipulation
- **pyspark**: Spark cluster client
- **mlflow**: MLflow libraries for versioning, running, packaging model experiment
- **cassandra-driver**: Cassandra DB client
- **pyignite**: Ignite DB client

The folders in "*user-workspace*" are the user home folders and will be mounted at the start, it contains some Machine learning example.

For demonstrating the capability of the system we can find our bracelet scenario, with some different models implemented in the folder "*bracelet_heart_disease*".

Each Data Scientist will have his workspace protected by a password ("*sa1920*" in the proptotype).

notes: if there will be many users, we suggest *JupyterHUB* for the production version of the system in conjunction with *Jupyter Kernel Gateway* for remote Kernel.

```
jupyter:
    build: './docker/jupyter'
    container_name: jupyter-container
    restart: always
    user: root
    environment:
        JUPYTER_ENABLE_LAB: 'yes'
        NB_USER: ${USER_USER}
        NB_GROUP: ${USER_GROUP}
        NB_UID: ${USER_UID}
        NB_GID: ${USER_GID}
        CHOWN_HOME: 'yes'
        CHOWN_HOME_OPTS: -R
        GRANT_SUDO: 'yes'
        MLFLOW_TRACKING_URI: 'http://mlflow:8080'
    volumes:
        - ./user-workspace/${USER_USER}:/home/${USER_USER}/work
        - ./dfs:/home/${USER_USER}/dfs
        - ./dfs/mlflow:/mlflow
    command: "start-notebook.sh --NotebookApp.password='${SHA_PWD_SA1920
        }' --NotebookApp.base_url=/user/${USER_USER}"
    ports:
        - "${JUPYTER_PORT}:8888"
```

As in Airflow we see the MLFLOW_TRACKING_URI pointing MLflow server. This allow to versioning the experiment made by a user in the centralized *Tracking System* but also deploy a pre-trained model for prediction purpose through the MLFlow API.

The access to DFS (DataLake), mounted in volumes
("*./dfs:/home/$USER1_USER/dfs*") allow the Data Scientist to access (read-only) to DataSets, Models and Artifacts history.

8.5.10 Spark

```
spark-master:
    image: bde2020/spark-master:2.4.4-hadoop2.7
    container_name: spark-master-container
    restart: always
    environment:
        - INIT_DAEMON_STEP=setup_spark
    ports:
        - "${SPARK_MASTER_PORT}:8080"

spark-worker-1:
    image: bde2020/spark-worker:2.4.4-hadoop2.7
    container_name: spark-worker-1-container
    restart: always
    depends_on:
        - spark-master
    environment:
        - "SPARK_MASTER=spark://spark-master:7077"
    ports:
        - "${SPARK_WORKER1_PORT}:8080"
```

Spark master and worker (only one in the prototype) is our *Processing Workers.*

The Data Scientist and the Production Engineers can exploit it, if they need the distributed and parallel job execution on the Spark Cluster. This is allowed by the pyspark python library installed on the Jupyter and Airflow services. The cluster could be extended, as already explained above, by adding slave workers from public cloud (eg. Azure Databricks or Google Dataproc).

notes: the Repository for the *Model Source Code* is GitHub for the prototype. In the production version, as usually, can be also internal (eg. GitLab gitlab/gitlab-ce) if we need.

notes: the prototype part don't provide a load balancer for the Served Model. For the production release of the system we can use Azure Load Balancer, Google Kubernetes Engine or AWS Load Balancer, in fact we already have the docker image (made by MLFlow) of the Served Model (part of the Online system).

Chapter 9

Second deliverable

9.1 Introduction

This part of the project will be focused on a real scenario: "Personal smart bracelet with heart rate monitor to alert with a possible heart Disease".

To make a metaphor, in the first deliverable we have designed and described a bicycle architecture in all its parts (wheels, brakes, chain, ratios, gears, pedals...) and how they interact together. After that, we have realized a real prototype of our bicycle assembling all the parts.
Now, it's time to sit on the seat of our prototype and try if it really works on a road (ie. test drive).

Our "road" is the **bracelet scenario**.

9.2 Scenario description

We have used the digram of *Overall architecture* (see 3.5) to the describe by points, the involved part of our architecture during the execution of the bracelet scenario.

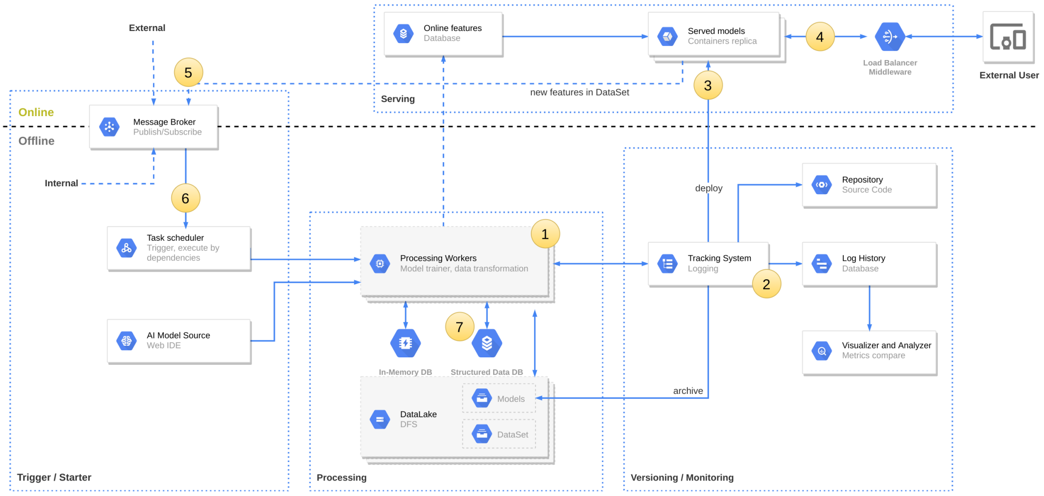


Figure 9.1: Bracelet scenario description.

Let's see the bracelet scenario, describing the steps:

1. Train a model on the Heart Disease dataset archive.ics.uci.edu/ml/datasets/Heart+Disease
2. Tracking all data related the training process, included the model trained, by MLFlow
3. Deploy the just trained model in a REST service to predict incoming JSON request from bracelet
4. Response to the requests with the predictions
5. Publish on a specific Kafka topic the incoming features sent during the request
6. Trigger an Airflow DAG (or directly) process for store the features
7. Store the incoming data for future reference in Cassandra (Structured Data DB)

At this point, the features are available for all system and any future use.

notes: the features collected with this process is equal (in the structure) to the dataset used for training, but missing the output value. So in this form is not possible to use them for further (supervised) training.

9.3 Online resources

Respect to the first deliverable, this part is more implementative, so we are focused on writing code. We have avoided reporting the source code in the document description, but you can easily access it through the available resources below.

All the source code is available on the same Github repository:

GitHub repository

notes: the repository has restricted access at this time, but as usual the source code can be retrieved also on GDrive.

A new video, with subtitles, that shows the scenario execution is available here:

https://youtu.be/rGJxirePeF0

The updated prototype is always deployed on the Google Compute Engine:

yamlops.uk.to

notes: the Jupyter password for all user is always: sa1920.

Now we have also the REST endpoint for model prediction, available here:

yamlops.uk.to/api/predict/bracelet

notes: it accepts a JSON request in this format:

```
{"age": 45, "sex": 1, "cp": 3, "trestbps": 112, "chol": 230, "fbs": 0, "
    restecg": 2, "thalach": 165, "exang": 0, "oldpeak": 2.5, "slope": 2,
    "ca": 1, "thal": 7}
```

Feel free try new training, prediction, make experiments and new workflows on the online prototype platform.

9.4 Source code change

For allowing the scenario execution we have added some code (eg. glue-code) and new "debugging" tools (for visualize data flow):

- **docker REST prediction container**: wrap the REST web service that exposes to the external user the prediction model (see *"docker/rest-prediction", docker-compose.yaml, nginx.conf*)
- **model REST web service**: the webserver that loads the trained model by the MLFlow *<run-id>* and answers (JSON) the prediction requests made by REST (see *"docker/rest-prediction/rest-prediction.py"*)
- **bracelet simulator**: a little REST client that send requests to our endpoint, simuling the external user behaviour (see *"test/rest-prediction-client.py"*)
- **new Airflow DAGs**: they use the libs created (*kafka.py and cassandra.py*) to manage the ingest and store of new features through a DAG process (see *ingest_bracelet.py, insert_bracelet_in_cassandra.py*)
- **airflow trigger**: it calls an Airflow DAG when new kafka message arrive or insert directly in the DB. It uses the same libs created for Airflow (see *kafka2Airflow.py*)
- **kafdrop and cassandra-web**: they are web ui interface client to visualize Cassandra and Kafka, now available on the right menu. These allow to query and see the internal status of the tools (see *docker-compose.yaml, nginx.conf*).

notes: there are some minor change and adding, please refer to the git logs for more detailed information.

9.5 Heart disease dataset

The Heart Disease dataset is composed by 14 attributes, 13 features and the predicted value (label):

- **age**
- **sex**: male or female
- **cp (chest pain type)**: angina, abnang, notang, asympt

- **trestbps (resting blood pres)**
- **chol (cholesteral)**
- **fbs (fasting blood sugar) $<$ 120**: true or false
- **restecg (resting ecg)**: norm, abn, hyper
- **thalach (max heart rate)**
- **exang (exercise induced angina)**: true or false
- **oldpeak**
- **slope**: up, flat, down
- **ca (number of vessels coloured)**
- **thal**: norm, fixed, rever
- **num (the predicted attribute: label)**: buff or sick

The symbolic attributes are transformed into numeric (eg. cp become an integer [1,4]). About the predicted value *"num"*, the dataset contains values [0,4] that we map with 0 is healthy and [1,4] is sick.

notes: you can find the final processed dataset used for traing in *"/user-workspace/*/bracelet_heart_disease/cleveland.csv"*.

9.6 Scenario flow

In the "flow chart" below we can see a more detailed view of the steps involved during the bracelet scenario. Try to see also the attached video (see 9.3) for well understand the workflow.

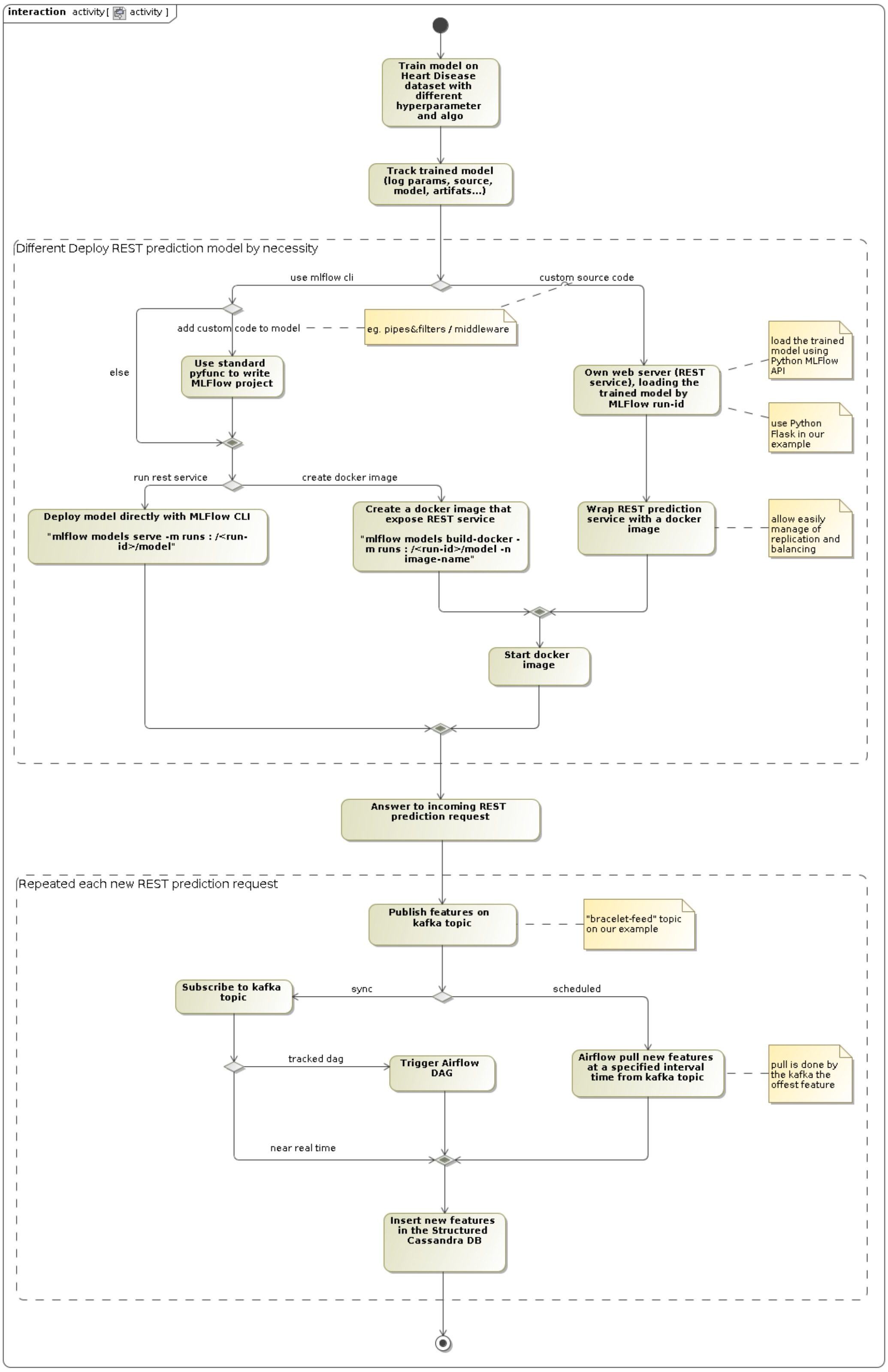

interaction activity[activity]
Train model on Heart Disease dataset with different hyperparameter and algo
Track trained model (log params, source, model, artifats...)
Different Deploy REST prediction model by necessity
use mlflow cli
custom source code
add custom code to model
eg. pipes&filters / middleware
else
Use standard pyfunc to write MLFlow project
Own web server (REST service), loading the trained model by MLFlow run-id
load the trained model using Python MLFlow API
use Python Flask in our example
run rest service
create docker image
Deploy model directly with MLFlow CLI
"mlflow models serve -m runs : /<run-id>/model"
Create a docker image that expose REST service
"mlflow models build-docker -m runs : /<run-id>/model -n image-name"
Wrap REST prediction service with a docker image
allow easily manage of replication and balancing
Start docker image
Answer to incoming REST prediction request
Repeated each new REST prediction request
Publish features on kafka topic
"bracelet-feed" topic on our example
sync
scheduled
Subscribe to kafka topic
tracked dag
Trigger Airflow DAG
Airflow pull new features at a specified interval time from kafka topic
pull is done by the kafka the offest feature
near real time
Insert new features in the Structured Cassandra DB

Figure 9.2: Flow chart of the bracelet scenario.

We have already seen in the above documentation the part relative to *Train a new model* and *Track all related data* (the first two blocks).

At this point, we have an MLFlow *<run-id>* with linked all the tracked data, in particular, the trained model that we want to use to deploy a REST service to serve prediction.

We don't need only to make a prediction, but we have also to collect the incoming requests and store them for future reference in Cassandra. For doing this, we have added a middleware to the web request process.

Thanks to MLFlow, we have different ways to deploy the REST service (these are exposed in the diagram under "Different Deploy REST prediction model by necessity" label).

We have preferred to build our custom web server. This allows us the best flexibility on the language, behaviour and technology used.
In our case, we have chosen the Python Flask framework. We have written a little REST web service (*"docker/rest-prediction/rest-prediction.py"*), that simply by changing the CLI params (*–run-id*) on start, allow us to decide which trained model, tracked and labelled with *<run-id>* in MLFlow, deploy.

```
python ./rest-prediction.py --run-id <run-id>
```

And we can test the REST endpoint with the bracelet simulator:

```
# send 10 REST request
python rest-prediction-client.py --url 'http://yamlops.uk.to/api/predict
    /bracelet'

# send a user predefined REST request
python rest-prediction-client.py --url 'http://yamlops.uk.to/api/predict
    /bracelet' --manual '{"age": 45, "sex": 1, "cp": 3, "trestbps": 112,
    "chol": 230, "fbs": 0, "restecg": 2, "thalach": 165, "exang": 0, "
    oldpeak": 2.5, "slope": 2, "ca": 1, "thal": 7}'
```

notes: there are also other params for *rest-prediction.py*, like the kafka topic where publish. Please refer to the commented source code or *–help* param to more info.

Of course as already said, the webserver publishes on kafka (topic: "bracelet-feed" for default) and set the features requested as payload.

Our webserver is wrapped in a docker image (see *"docker/rest-prediction/"*), so we can easily manage replication, balancing, adding and removing docker container of the same image.

```
docker-compose run --rm --entrypoint 'python ./rest-prediction.py --run-
    id <run-id>' rest-prediction-cotainer
```

Arrived here, we have the "bracelet-feed" kafka topic filled and growing with the new features incoming.

From the project specification is not clear if we have to gather these new data in (near)real-time, low-latency or scheduled, so we have implemented three different way, reported also in the flow chart above:

- **sync**: each time a new REST request prediction arrived, it is stored in the DB
  - **(near)real-time**: *"python kafka2airflow.py –mode cassandra"* - it doesn't pass data through Airflow, but directly insert data in Cassandra.

    This allows the best (low-)latency, but we lost all the benefits Airflow brings (DAG process, reliable, track, log tasks...)
  - **tracked DAG**: *"python kafka2airflow.py –mode airflow"* - it triggers the Airflow DAG *"insert_bracelet_in_cassandra"* and pass it the features.

    In this case, we have the overhead of Airflow (time execution increase), but all the process is more robust and tracked.
- **scheduled**: here we do not need of any trigger system for Airflow. It pulls the information directly from the kafka topic at a specific scheduled time (20 minutes on prototype). All the new features published since the last pull are retrieved thanks to the "kafka offset system". The DAG (*"ingest_bracelet"*) is made up of two tasks *"kafka_consumer"* and *"insert_cassandra"*.

  This approach is the most robust and reliable, but the data are pushed to Cassandra not when they arrive (respecting the scheduled time).

notes: all these three different ways use the same libraries created in the path: *"docker/dags/libs"*.

Finally, we have stored the features in Cassandra and we can use it in Jupyter (by cassandra-driver) or from any other components of our system.

Chapter 10

Conclusions

We think to have reached the goal requested in the project specification and also more of what had been requested. All the documentation, source code, online prototype and videos should be the proof.

The architecture design decisions are been correct and all the project has been implemented. The system allows managing without effort the bracelet scenario.

The prototype deployed online seems quite mature, it fulfils all the functional requirements, but we don't have enough budget (ie. money on Google Cloud) to create an adequate machine to respect the non-functional ones (this is normal for a prototype).

Anyway, all the choices done was oriented to allow the system to scale very easily at necessity, thanks to the docker approach and the distributed philosophy of the utilized tools (eg. Kafka, Spark, Cassandra...), so to respect all the non-functional requirements in the production deploy.

Appendices

Appendix A

Thanks to

- LinuxFellows Team for the base LaTeX template
- Apache foundation and the company that have donated the base code (Google, Microsoft, AirBnb, Amazon, ..) for the amazing Open Source environment developed and released
- All the braves have read all the documentation :)

Appendix B

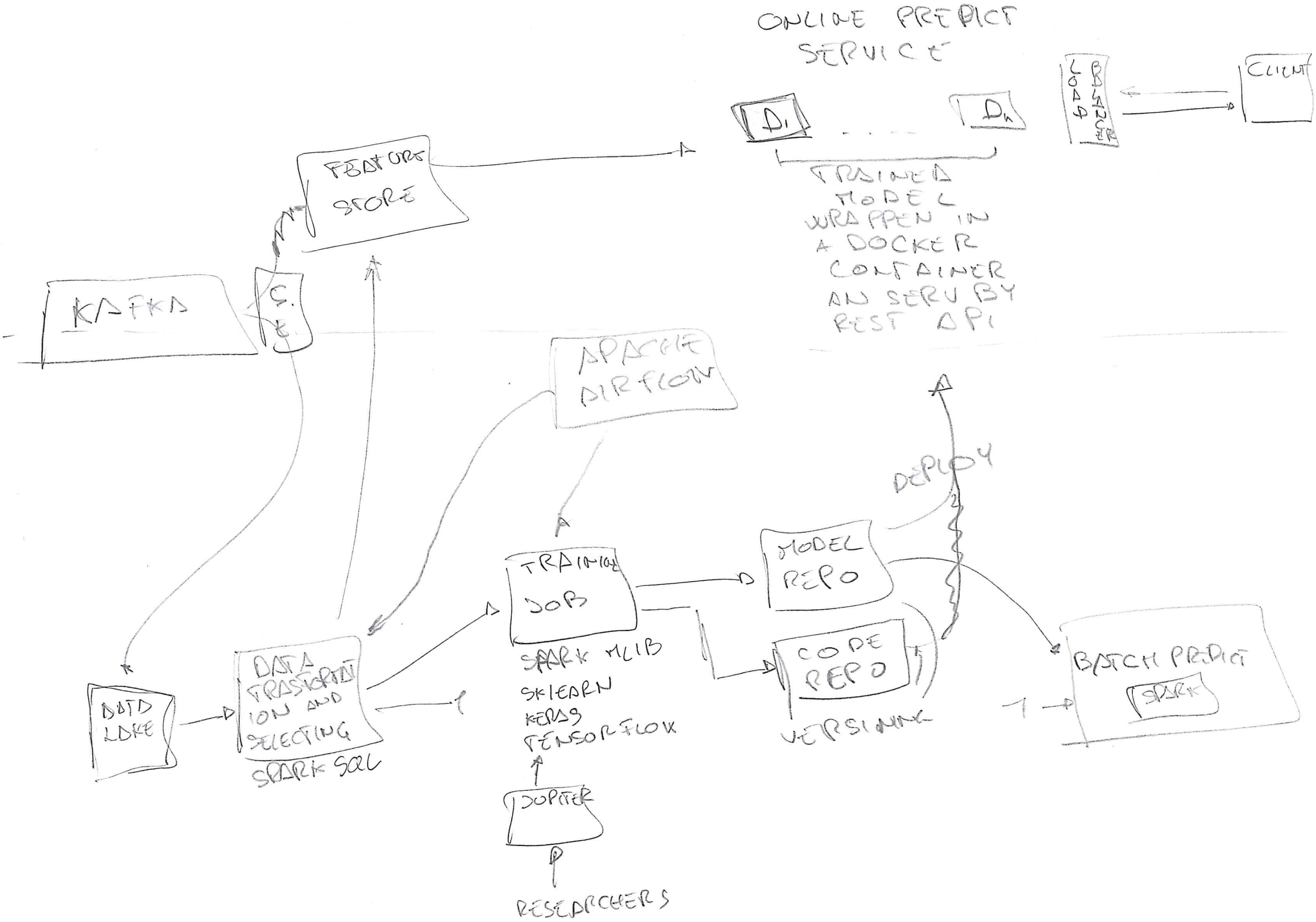
Figures and Tables

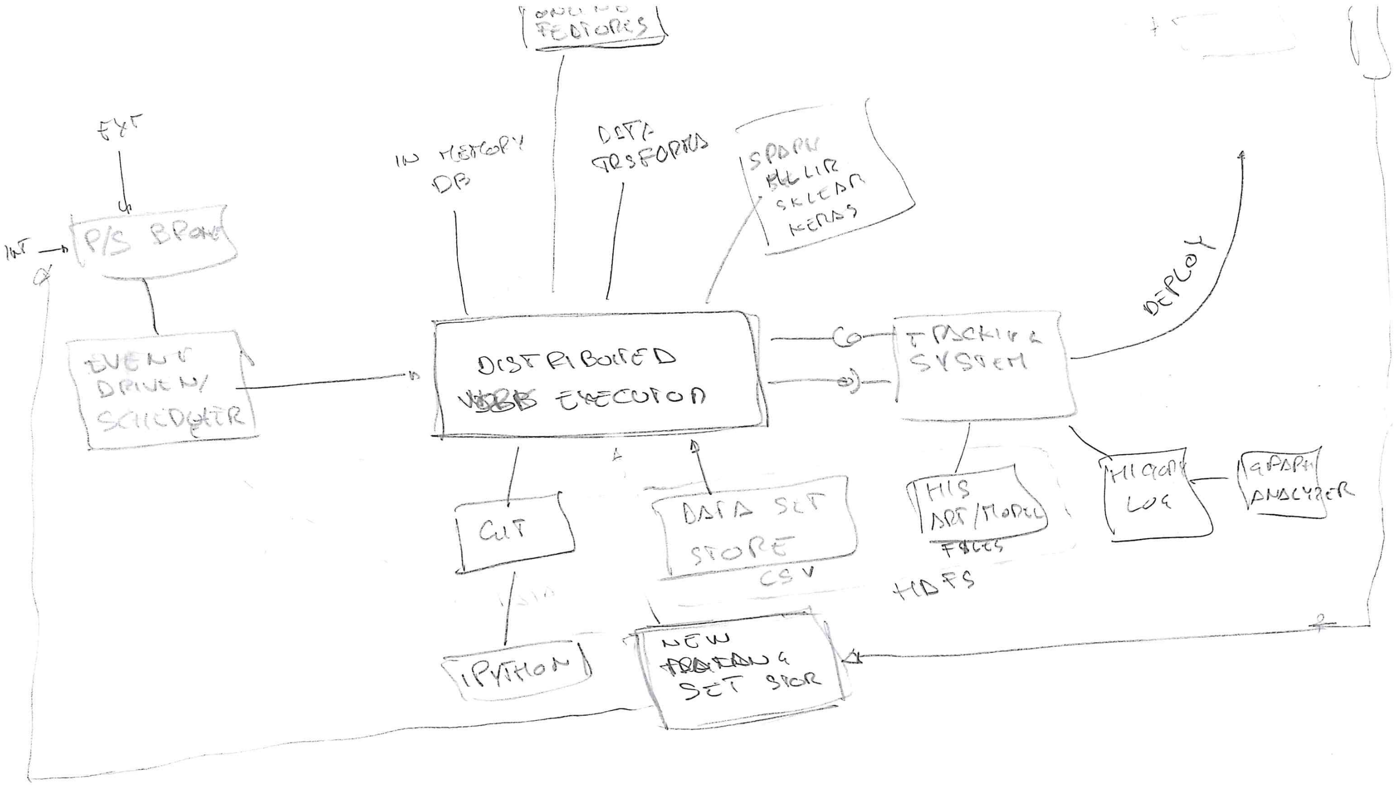
List of Figures

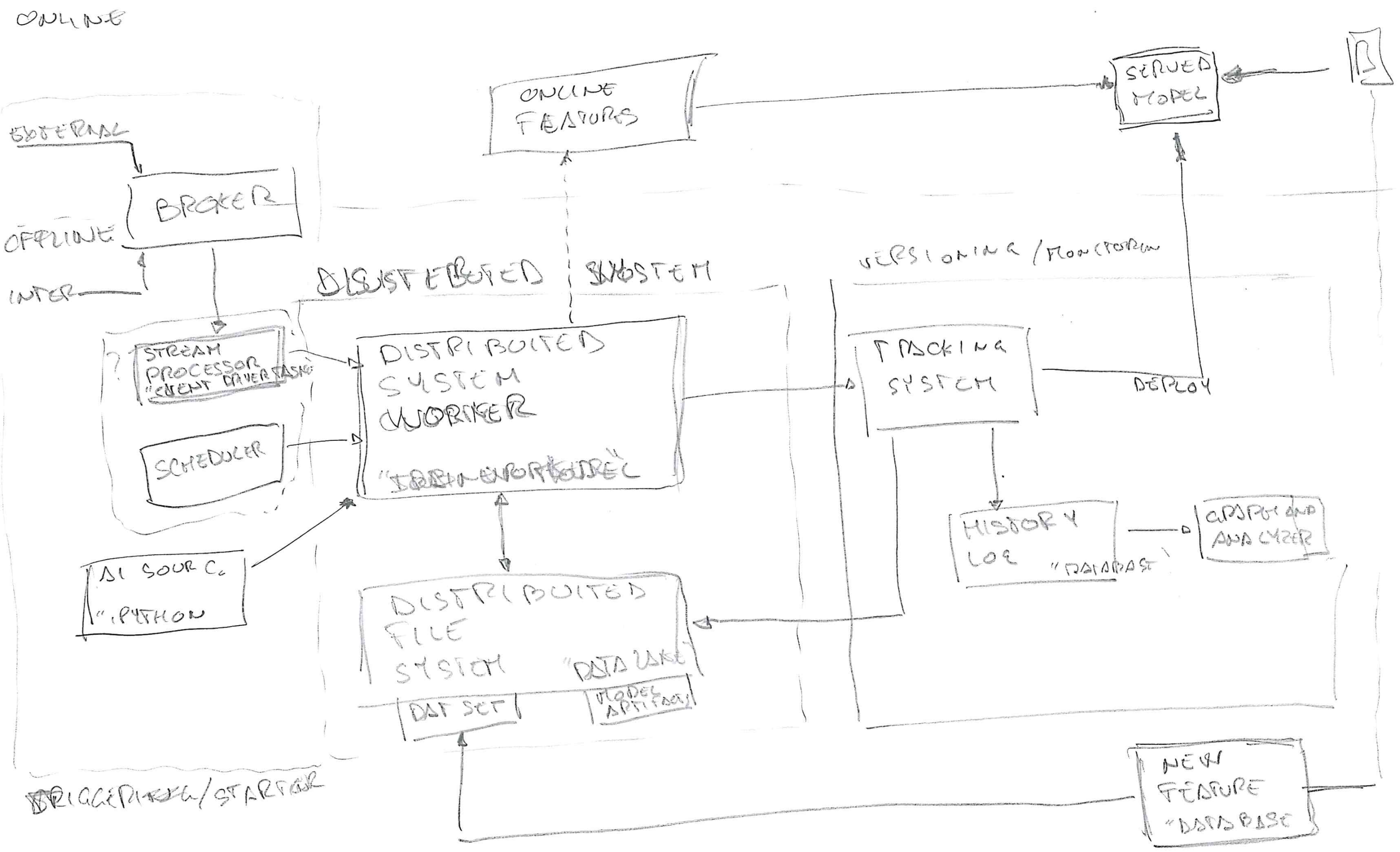
List of Tables

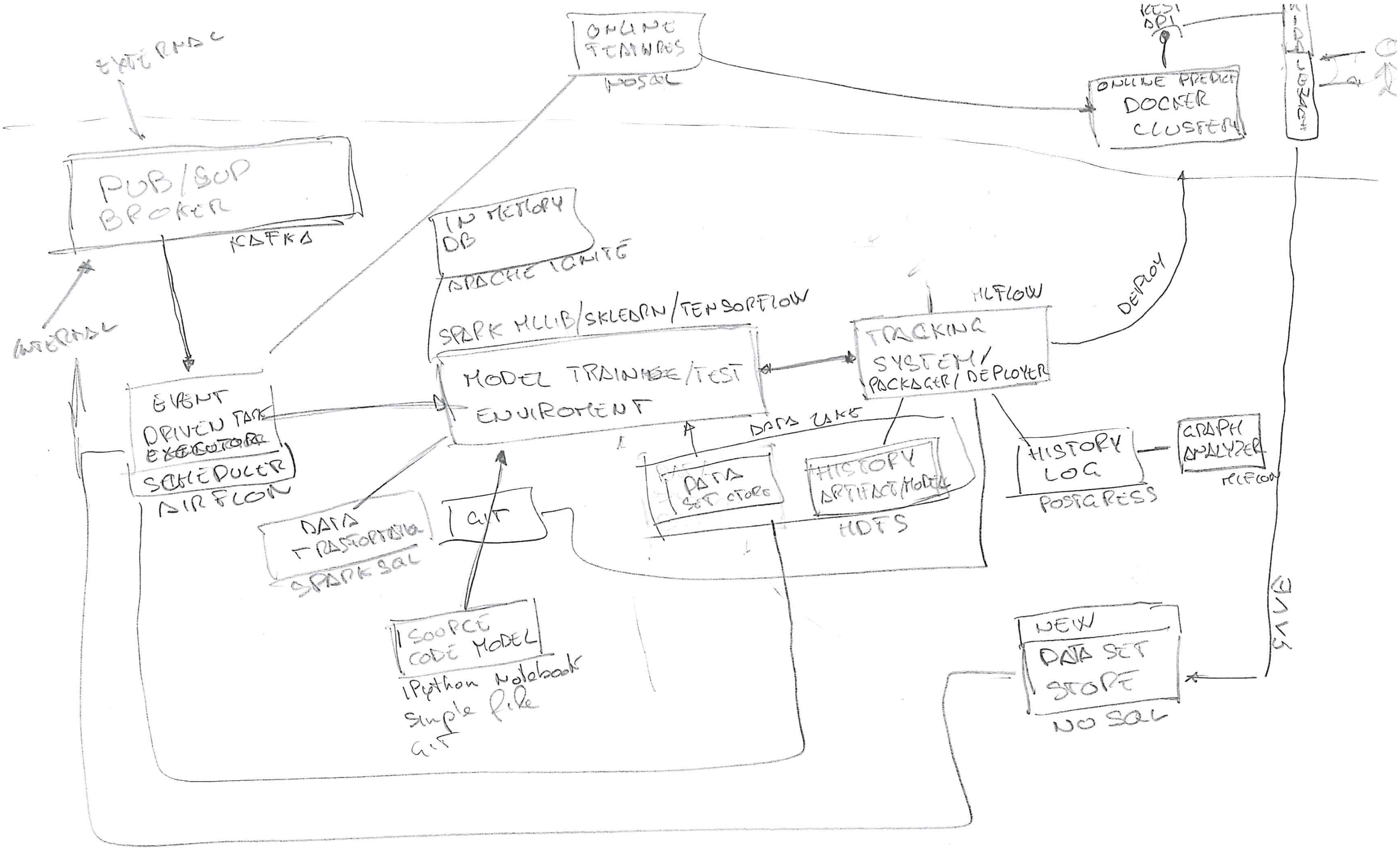
Appendix C

Paper to Architecture


ONLINE PREDICT SERVICE
D1
....
Dn
LOAD BALANCER
CLIENT
FEATURE STORE
TRAINED MODEL WRAPPED IN A DOCKER CONTAINER AN SERV BY REST API
KAFKA
S. E.
APACHE AIRFLOW
DEPLOY
TRAINING JOB
MODEL REPO
CODE REPO
VERSIONING
BATCH PREDICT
SPARK
DATA LAKE
DATA TRASFORMATION AND SELECTING
SPARK SQL
SPARK MLIB
SKLEARN
KERAS
TENSORFLOW
JUPITER
RESEARCHERS


ONLINE
FEATURES
EXT
INT
P/S BROKER
IN MEMORY
DB
DATA
TRSFORMS
SPARK
MLLIB
SKLEAR
KERAS
EVENT
DRIVEN/
SCHEDULER
DISTRIBUTED
~~WEB~~ EXECUTION
TRACKING
SYSTEM
DEPLOY
GIT
DATA SET
STORE
CSV
HIS
ART/MODEL
FILES
HDFS
HISTORY
LOG
GRAPH
ANALYZER
IPYTHON
NEW
~~TRAINING~~
SET STOR


ONLINE
ONLINE FEATURES
SERVED MODEL
EXTERNAL
BROKER
OFFLINE
INTER
DISTRIBUTED
SYSTEM
VERSIONING / MONITORING
?
STREAM PROCESSOR "EVENT DRIVEN TASKS"
SCHEDULER
DISTRIBUTED SYSTEM WORKER
"[illegible]"
TRACKING SYSTEM
DEPLOY
AI SOURCE "PYTHON"
HISTORY LOG "DATABASE"
GRAPH AND ANALYZER
DISTRIBUTED FILE SYSTEM "DATA LAKE"
DATASET
MODEL ARTIFACT
TRIGGERING / STARTER
NEW FEATURE "DATABASE"


EXTERNAL
INTERNAL
PUB/SUP BROKER
KAFKA
ONLINE FEATURES
NOSQL
REST API
ONLINE PREDICT
DOCKER CLUSTER
IN MEMORY DB
APACHE IGNITE
SPARK MLLIB/SKLEARN/TENSORFLOW
MODEL TRAINING/TEST ENVIROMENT
MLFLOW
TRACKING SYSTEM/ PACKAGER/ DEPLOYER
DEPLOY
EVENT DRIVEN TASK ~~EXECUTOR~~
SCHEDULER
AIRFLOW
DATA LAKE
DATA SET STORE
HISTORY ARTIFACT/MODEL
HDFS
HISTORY LOG
POSTGRESS
GRAPH ANALYZER
MLFLOW
DATA TRASFORMATION
SPARK SQL
GIT
SOURCE CODE MODEL
IPython Notebook
simple file
GIT
NEW
DATA SET STORE
NO SQL
SAVE

Bibliography

[1] MLflow,
https://www.mlflow.org/

[2] Apache kafka,
https://kafka.apache.org/

[3] Apache Ignite,
https://ignite.apache.org/

[4] Apache Cassandra,
http://cassandra.apache.org/

[5] Apache Spark,
https://spark.apache.org/

[6] Apache Hadoop,
https://hadoop.apache.org/

[7] Apache Airflow,
https://airflow.apache.org/

[8] Docker,
https://www.docker.com/

[9] Nginx,
https://www.nginx.com/

[10] PostgresSQL,
https://www.postgresql.org/

[11] JupyterLab,
https://jupyterlab.readthedocs.io/en/stable/

[12] Meet Michelangelo: Uber's Machine Learning Platform,
https://eng.uber.com/michelangelo/

[13] Overview of the different approaches to putting Machine Learning (ML) models in production,
`https://medium.com/analytics-and-data/overview-of-the-different-approaches-to-putting-machinelearning-ml-models-in-production-c699b34abf86`

[14] Using Kafka-Python to illustrate a ML production pipeline,
`https://github.com/jrzaurin/ml_pipelines`

[15] Empowering Spark with MLflow,
`https://towardsdatascience.com/empowering-spark-with-mlflow-58e6eb5d85e8`

[16] Building Production Machine Learning Systems,
`https://heartbeat.fritz.ai/building-production-machine-learning-systems-7eda2fda0cdf`

[17] Keeping your ML model in shape with Kafka, Airflow and MLFlow,
`https://medium.com/vantageai/keeping-your-ml-model-in-shape-with-kafka-airflow-and-mlflow-143d20024ba6`

[18] MLOps on Azure,
`https://github.com/microsoft/MLOps`