UNIVERSITÀ DEGLI STUDI DI CATANIA

Dipartimento di Matematica e Informatica

Corso di Laurea Magistrale in Informatica

Rosario Scalia

(1000008648)

Progetto di Ingegneria dei Sistemi Distribuiti

Anno Accademico 2020 - 2021

Indice

Capitolo 1

Introduzione

1.1 Descrizione Dominio di Interesse

Negli ultimi anni, è sempre più crescente l'interesse verso soluzioni software che utilizzino il Machine Learning.

Tale interesse, non è soltanto di natura accademica ma anche e soprattutto di natura industriale.

Del resto, sono innumerevoli gli esempi di software commerciali che utilizzano algoritmi di *ML*.

Il Machine Learning ,a differenza di altre branche dell'Informatica, è un settore multi-disciplinare dato che contiene una cospicua parte Matematica (Algebra Lineare, Analisi Matematica, Ricerca Operativa, Teoria dei Segnali) oltre che alla parte strettamente Informatica (Algoritmi, Programmazione, Database, Ingegneria del Software, Sistemi Distribuiti).

Di conseguenza, non è inverosimile trovare ambienti lavorativi dove collaborino Informatici, Matematici e anche Fisici in merito ai progetti di Machine Learning.

Quest'ultima prerogativa porta *inevitabilmente* delle differenze dal punto di vista dello sviluppo software dato che i Matematici/Fisici non hanno ricevuto una formazione da Informatici.

Per alleviare tale differenza, stanno nascendo una serie di servizi Web che permettono di sviluppare algoritmi di Machine Learning in maniera *visuale* e senza scrivere codice.

Un esempio notevole di ciò, è il servizio *Azure Custom Vision* di Microsoft.

Tale servizio permette di addestrare Modelli di Machine Learning per la Classificazione di Immagini.

Il servizio presenta una comoda interfaccia Web che ci permette di lanciare gli addestramenti, visionare le prestazioni del Modelli addestrati e anche fare inferenza.

Nonostante ciò, il servizio ,nel momento in cui si scrive, soffre di svariati deficit come l'evaluation dei Modelli sul training-set oppure l'assenza di informazioni in merito ai Modelli di ML messi a disposizione.

Da tale premessa, nasce l'idea di sviluppare il progetto corrente, ovvero una soluzione software che permetta di gestire il ciclo di vita degli algoritmi di ML e contestualmente migliorare i suddetti difetti presenti in Custom Vision.

1.2 Scopo del Progetto

Il progetto corrente si prefigge lo sviluppo di un'applicazione distribuita che supporti l'utilizzo di algoritmi di Machine Learning nelle fasi di Addestramento e Valutazione.

Un ulteriore obbiettivo del progetto è lo sviluppo di un sistema *modulare*, ovvero un sistema che renda semplice l'aggiunta di nuovi Modelli/Dataset al sistema.

Capitolo 2

Descrizione dei Requisiti

2.1 Requisiti Funzionali

La *principale* funzionalità dell'applicativo è la possibilità di addestrare Modelli di Machine Learning.

Più precisamente, è possibile scegliere fra i seguenti modelli:

1. SVM
2. Logistic Regressor
3. Decision Tree
4. Random Forest
5. Naive Bayes

I dataset disponibili sono i seguenti:

1. **Iris-Fisher**, dataset multivariato contente 150 record di misure effettuate su una particolare tipologia di fiore chiamato *Iris*.

   Il task associato al dataset è la *Classificazione della specie* del fiore date le suddette misure.

2. **Height-Weight Dataset**, il dataset Height-Weight è un dataset multivariato avente *4232* record contenenti misure biologiche effettuate su un campione di individui.

   Ogni record del dataset contiene le seguenti caratteristiche:

   - Indice di Massa Corporea (BMI)
   - Altezza Individuo
   - Peso Individuo
   - Sesso Individuo

   Il task associato al dataset è la classificazione del Sesso dell'individuo a partire dalla conoscenza della sua Altezza, Peso e BMI.

L'applicazione supporta anche la scelta di specifici Algoritmi di Learning/Loss oltre che alla scelta di particolari Iperparametri legati a quest'ultimi.

Quest'ultima caratteristica è supportata anche per i Modelli e per i Dataset.

Un'ulteriore funzionalità dell'applicazione è il *Catalogo*.

Quest'ultimo contiene le seguenti informazioni:

1. Dataset disponibili
2. Modelli disponibili
3. Algoritmi di Learning disponibili
4. Loss Function disponibili
5. Misure di Valutazione Disponibili

Chiaramente, il catalogo conterrà tutti i parametri ammissibili legati ad una richiesta di training e sarà visualizzabile dal client dell'applicazione.

Oltre ad essere un'utile guida per l'utente, il Catalogo rappresenta anche un componente che facilità l'espansione futura in termini di Modelli e/o Dataset.

In particolar modo, l'idea è di consentire l'aggiunta di Modelli/Dataset senza modificare il codice sorgente della parte client dell'applicazione.

Un'ulteriore funzionalità della soluzione sviluppata è la possibilità di valutare i Modelli già addestrati.

Le metriche supportate sono *Precision* e *Recall*.

Dal punto di vista della persistenza degli esperimenti, è disponibile uno *Storage di Sessione* e uno *Storage Permanente*.

Lo storage di Sessione contiene tutti gli esperimenti *non ancora confermati* dall'utente; per tanto il loro stato sarà ancora modificabile.

Invece, quello *permanente* contiene tutti gli esperimenti confermati dall'utente; essendo confermati non sarà più possibile apportarvi modifiche.

Dal punto di vista dell'esplorazione degli esperimenti, l'applicazione permette la Visualizzazione dei dati di Sessione/Permanenti e il *Salvataggio* di un esperimento di sessione nello storage permanente.

2.2 Requisiti NON Funzionali

2.2.1 Requisiti di Sicurezza

I requisiti di sicurezza sono più che altro concentrati nell'idea di proteggere i record di sessione da modifiche *illegittime*.

Con tale scopo in mente, è stato implementato un sistema di Cifratura Simmetrico per gli Id di Sessione trasmessi in rete.

Grazie a tale caratteristica, sarà più difficile un loro utilizzo all'interno di richieste *fraudolente*.

2.2.2 Monitoraggio Applicativo

Con l'obbiettivo di migliorare l'affidabilità, è stato incluso un sistema di monitoraggio delle computazioni all'interno della soluzione sviluppata.

Più in generale, l'idea è che lo stato dell'applicazione debba essere *sempre ricostruibile*.

Inoltre, con l'obbiettivo di rendere fruibili le informazioni di Monitoraggio, è stata sviluppata una console di Amministrazione che permette una visualizzazione consona dei log comportamentali del sistema.

Chiaramente, tale console sarà accessibile solo dall'amministratore di Sistema.

Capitolo 3

Architettura dell'Applicazione

3.1 Architettura Complessiva

La soluzione sviluppata è un software distribuito, per tanto è presente un frontend e un backend.

Il Backend è stato sviluppato seguendo l'approccio dei *Microservizi*.

Ogni microservizio impiega la tecnologia REST per comunicare col client e/o con un altro microservizio.

Per le comunicazioni asincrone, viene impiegato un Message Broker.

Dal punto di vista dell'elaborazione delle richieste, esistono 3 flussi di lavoro differenti su ogni microservizio:

- **REST WORKER**, entità che resta in ascolto in merito alle chiamate verso il REST End-Point del microservizio.
- **EVENT WORKER**, entità che resta in ascolto in merito a nuovi eventi associati ad un task specifico (es. comunicazione tramite Message Broker).
- **TASK WORKER**, entità che prende in carico l'esecuzione di un task associato alla logica di dominio dell'applicazione.

Dal punto di vista dell'esecuzione concorrente, verrà sfruttato ove possibile l'async-processing.

L'unica eccezione sono il *Training ed Evaluation* dei modelli di ML.

Per tali casistiche, si andrà a sfruttare il Multi-Processing per ragioni prestazionali.

In tutti i casi, ogni microservizio viene replicato su più processi in modo da migliorare la capacità di gestione delle richieste.

La replica dei microservizi è possibile grazie ad una specifica scelta di design dell'architettura.

Nello specifico, è stata implementata un'architettura *stateless*, ovvero un'architettura dove i Microservizi sono slegati dallo stato dell'applicazione.

Per tanto, una replica di quest'ultimi su più processi non causa problemi.

Dal punto di vista del frontend, è stato sviluppato un client a linea di comando.

Quest'ultimo sfrutta le API REST messe a disposizione dal backend per lanciare le funzionalità dell'applicativo.

In maniera simile a quanto fatto nel backend, il client sfrutta l'async-processing per la gestione dell'interfaccia utente oppure per l'inoltro delle richieste al backend.

3.2 Layer di Presentazione

L'architettura generale del frontend è illustrata del seguente schema:

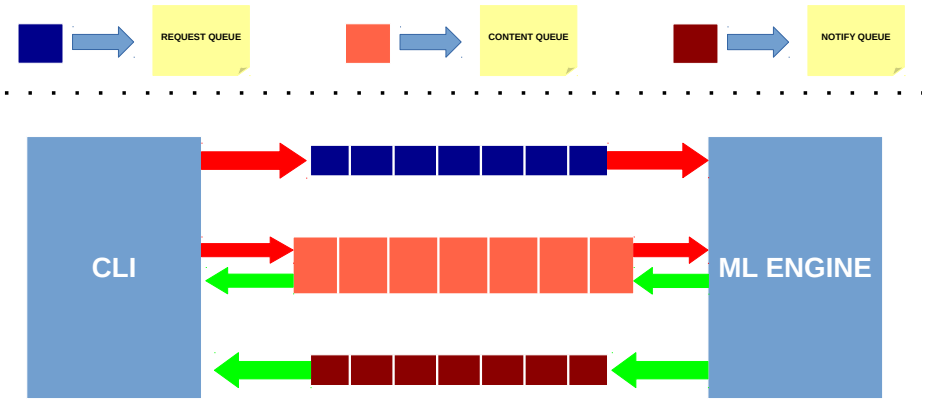

Figura 3.1: Architettura Client

Lo schema mostra la presenza di due corutine e di tre code.
La corutine *CLI* rappresenta l'interfaccia a linea di comando.
La corutine *ML Engine* permette l'inoltro delle richieste dell'utente al Backend.
Le code, permettono la comunicazione fra l'interfaccia utente e l'ML Engine e posseggono rispettivamente le seguenti specifiche:

- **Request Queue**, questa coda permette di inoltrare richieste di Operazioni al backend (es. training).

  La particolarità di tali richieste è che necessitano la restituzione di una notifica dal server.

  La notifica, è una stringa che informa il client in merito alla buona riuscita o meno dell'operazione.

- **Notify Queue**, questa coda contiene i messaggi di risposta (o notifica) del server alle richieste di Operazione inoltrate del Client.

- **Content Queue**, questa coda permette di inoltrare e ricevere del contenuto dal Backend (es. visualizzazione esperimenti).

  Di conseguenza, il dato restituito dal Backend non sarà una notifica (caso Request/Notify Queue) ma bensì un dato strutturato.

3.3 Layer di Business-Logic

L'architettura del backend è illustrata dal seguente schema:

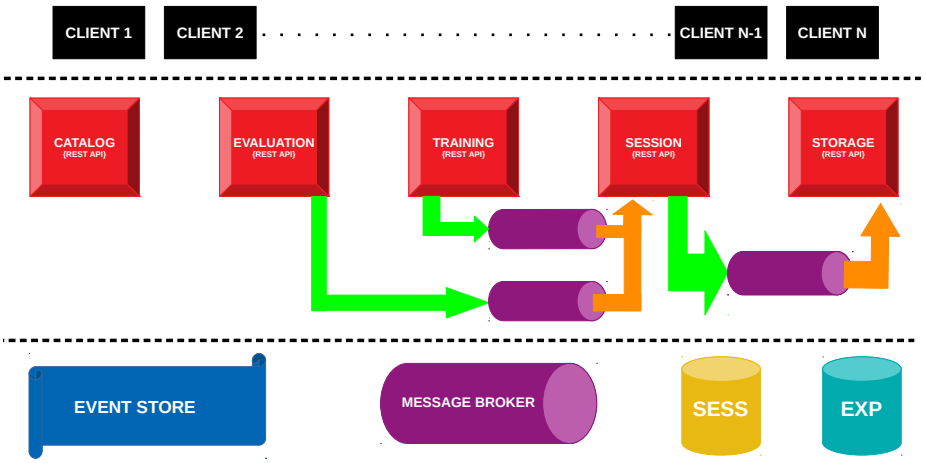

Figura 3.2: Architettura Backend

Lo schema mostra la presenza di cinque microservizi:

- **Catalog**, il microservizio Catalog espone un'API che permette il download di una copia aggiornata del catalogo.
- **Training**, il microservizio di Training permette l'addestramento di un Modello di Machine Learning.

  Una volta terminato l'addestramento, verrà generato un record contente le informazioni del training.

  Il record generato verrà successivamente scritto sul database di Sessione.

  La scrittura del record passa attraverso la comunicazione fra il microservizio di Training e il microservizio Session.

  La comunicazione fra i due microservizi avverrà attraverso il servizio di messaggistica presente nel backend.

  Un ulteriore compito di Training è quello di effettuare un controllo in merito alla correttezza delle richieste inoltrate dal client.

  A tal proposito, è previsto il download del catalogo dal microservizio *Catalog*.

  Fatto ciò, si potrà provvedere ad effettuare i dovuti controlli.

  Quest'ultima caratteristica fornisce modularità al progetto dal punto di vista di aggiunte future.

- **Evaluation**, il microservizio di Evaluation permette di valutare un Modello di ML secondo opportune metriche.

  Più precisamente, viene esposta un'API di evaluation che prende in input l'id del record di sessione associato al Modello da valutare.

Di conseguenza, vi è la necessità di verificare se l'id inviato dall'utente sia effettivamente presente sul DB.

Per tale compito, è richiesta la comunicazione col microservizio Session.

In caso positivo, verrà inviata una notifica al client e successivamente verrà avviata l'evaluation.

Al termine dell'evaluation, verrà inoltrato un messaggio al microservizio Session sfruttando un'apposita coda.

Il contenuto del messaggio comanda l'aggiornamento del record di Sessione con i nuovi dati di evaluation.

In caso negativo, verrà inviata una notifica di errore al client.

- **Session**, il microservizio Session rappresenta il fulcro del Backend.

  Quest'ultimo è responsabile della gestione dei dati di Sessione dell'applicativo.

  Più sinteticamente, Session gestisce le seguenti richieste:

  1. **Query sui dati di Sessione**
  2. **Check Esistenza Record Sessione**
  3. **Richiesta Creazione nuovo Record di Sessione**
  4. **Richiesta Aggiornamento Record di Sessione**
  5. **Richiesta Archiviazione Record di Sessione**
     Per tale richiesta, è necessario comunicare col Microservizio Storage attraverso un'apposita coda fornita dal Message Broker.

- **Storage**, il microservizio Storage permette la gestione del DB degli Esperimenti.

  Il microservizio supporta le seguenti funzionalità:

  1. **Query Dati Esperimenti**
  2. **Richiesta Scrittura Record Esperimento**
     Per tale task, sarà necessario leggere da un'apposita coda del Message Broker.

     Nella suddetta coda, saranno presenti i record da archiviare inoltrati dal microservizio Session.

Come accennato in passato, il Backend supporta l'utilizzo della messaggistica.

Nello specifico, sono presenti le seguenti code:

1. **sessionRecord**
   - **Commenti**, permette di generare un record di sessione nell'apposito database
   - **Mittente Messaggi**, Training
   - **Destinatario Messaggi**, Session
2. **sessionUpdate**

- **Commenti**, permette di modificare un record di Sessione.
- **Mittente Messaggi**, Evaluation
- **Destinatario Messaggi**, Session

3. **storageRecord**
   - **Commenti**, permette di generare un record di tipo Esperimento nell'apposito DB.
   - **Mittente Messaggi**, Session
   - **Destinatario Messaggi**, Storage

3.3.1 Training & Catalog

La logica di business associata alle richieste di training è riassumibile dal fatto che le richieste devono essere conformi a quanto prescritto dal catalogo.

Il suddetto catalogo presente il seguente schema:

- `data_lake : List[dataset]`
  - `dataset_name : string`
  - `dataset_task : string`
  - `dataset_description : string`
  - `features : List[string]`
  - `label : List[string]`
  - `n_record: int`
  - `pre-processing : List[seq_computation]`
    - `{ "step" : int , "computation" : string }`
- `models : List[model]`
  - `model_name : string`
  - `model_task : string`
  - `model_hyperparams : List[param_spec]`

```
* {
  param_name: string ,
  param_type: string ,
  default: primitive_type ,
  range_l : int | float ,
  range_u : int | float ,
  options : List[string]
  }
```

- `metrics : List[metric]`
  - `metric_task : string`
  - `metric_name : string`
- `learning : List [learning_asset]`

```
- model_reference :
  { "model_name":  string , "model_task":string }

- loss :  string
- learning_algorithm :  string
- learning_hyperparams :  List[param_spec]
    *  {
      param_name:  string ,
      param_type:  string ,
      default:  primitive_type ,
      range_l :  int | float ,
      range_u :  int | float ,
      options :  List[string]
      }
```

La logica di controllo delle richieste di training è illustrata dalla seguente lista:

- La coppia `dataset_name + dataset_task` *dichiarata* deve esistere nel Catalogo
- Lo `split_test` dichiarato deve essere un numero compreso fra 0 e 1
- Il Modello (`model_name + model_task`) *dichiarato* deve esistere nel Catalogo
- Il `model_task` *dichiarato* deve *combaciare*[1] con quello del Dataset *Scelto*
- I `model_hyperparams` *dichiarati* devono *combaciare* con quelli disponibili per il Modello Selezionato
- La coppia `loss + learning_algorithm` *dichiarata* deve *combaciare* con una di quelle disponibili per il Modello Selezionato
- I `learning_hyperparms` devono *combaciare* con quelli disponibili per il Modello Selezionato e per la Loss/Learning Algorithm selezionata

Di seguito vengono mostrati una serie di schemi che evidenziano la logica di business correlata ad alcune feature dell'applicativo:

[1] combaciare = esistere ed essere ben formato

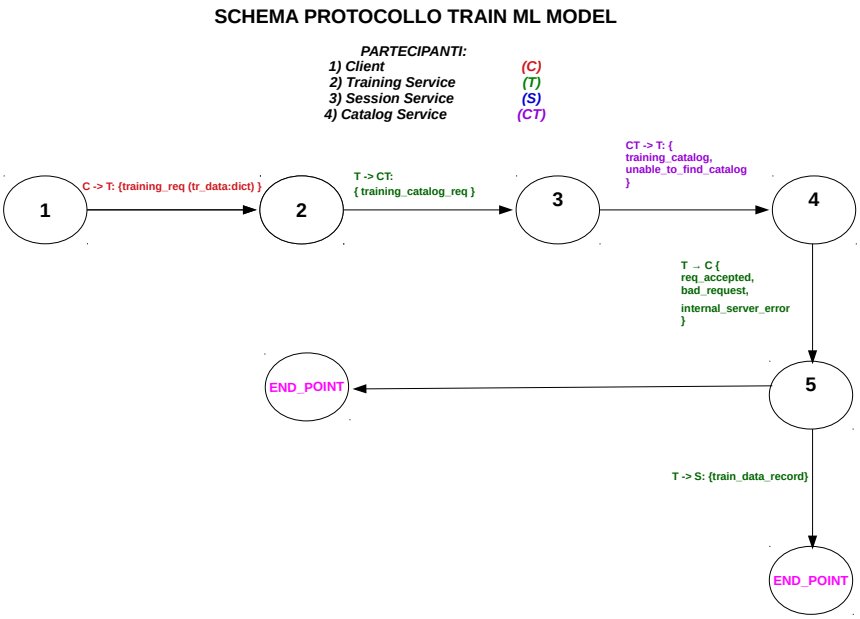

Figura 3.3: Logica Training Modello ML

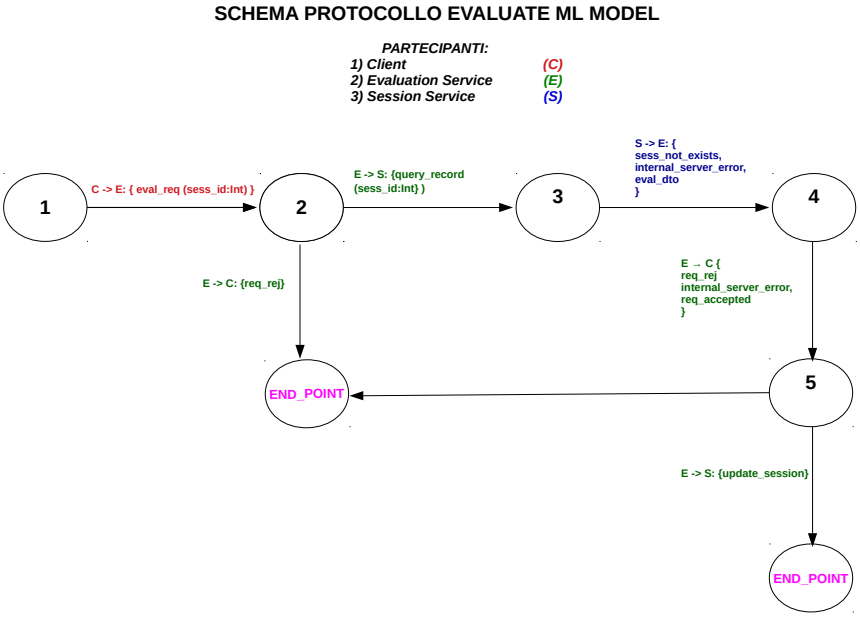

Figura 3.4: Logica Evaluation Modello ML

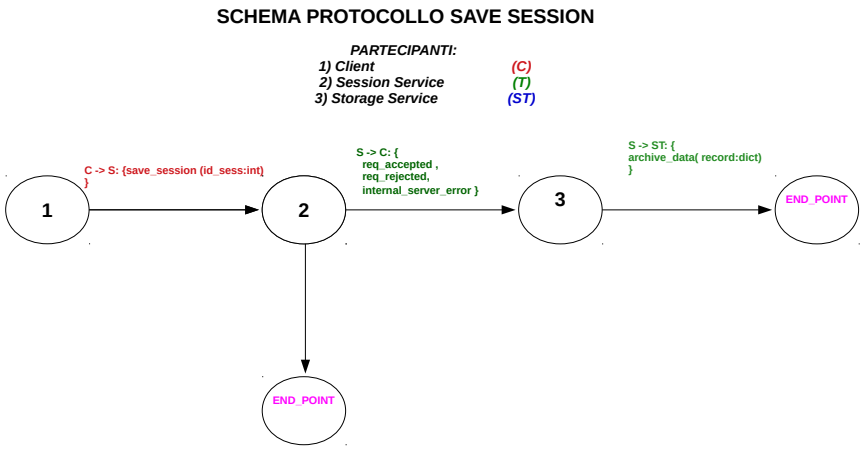

Figura 3.5: Logica Salvataggio Sessione

3.3.2 Sicurezza

I requisiti di sicurezza dell'applicazione cercano di proteggere i dati di sessione da modifiche illegittime.

Con tale obbiettivo in mente, è stato implementato uno schema di cifratura simmetrico su tutti gli Id di Sessione che viaggiano dal Frontend al Backend e viceversa.

L'algoritmo utilizzato è stato *AES* e la chiave impiegata per la cifratura è una chiave a *128 bit*.

I contesti in cui vengono cifrati gli Id di Sessione sono i seguenti:

- Richiesta Evaluation Modello di Machine Learning
- Query Dati Sessione
- Archiviazione Record di Sessione

3.3.3 Monitoraggio

Il monitoraggio è molto importante per individuare i difetti del software, a maggior ragione in un sistema a Microservizi.

La soluzione sviluppata include due sistemi distinti di monitoraggio.

Il Primo *monitora* le computazioni *interne* dei microservizi.

Nello specifico, verrà scritto un apposito messaggio informativo (su `stdout` e `file`) nei casi di errore a runtime.

Il messaggio presenta il seguente formato:

```
| DATE: DD/MM/YYYY | TIME: HH:MM:SS | [ TIPO_FLUSSO @ FUNZIONE ] ...
```

Il Secondo sistema di monitoraggio permette il controllo comportamentale dei microservizi.

Più in dettaglio, vengono tracciati gli scambi di messaggi fra i vari microservizi attraverso la generazione di appositi *record comportamentali*.

I suddetti record vengono archiviati in appositi *aggregatori* presenti nell'event-store.

A tal proposito, il sistema supporta i seguenti aggregatori:

- catalog
- training
- evaluation
- session
- sessionRecord
- sessionUpdate
- storage
- storageRecord

Lo schema di un record comportamentale è illustrato di seguito:

- **Key**
  - **Tipo**: bytes
  - **Commenti**: microservizio *duale* rispetto a quello dell'aggregatore
  - **Opzioni** (**Γ**):
    - Catalog
    - Training
    - Evaluation
    - Session
    - Storage
- **Value**
  - `source_service`
    - **Tipo**: string
    - **Opzioni**: $\longrightarrow \Gamma$
  - `destination_service`
    - **Tipo**: string
    - **Opzioni**: $\longrightarrow \Gamma$
  - `message_type`
    - **Tipo**: string
    - **Opzioni**: "send/receive"
  - `com_type`
    - **Tipo**: string
    - **Opzioni**: "sync/async"
  - `payload`
    - **Tipo**: str

- **Timestamp**
  - **Tipo**: int
  - **Commenti**: timestamp Azione in millisecondi

La tipologia di payload presente nei record è illustrata dalle seguenti tavole:

Sorgente	Destinazione	Tipo Msg	Comunicazione	Payload
CLIENT	CATALOG	`receive`	`async`	*training_catalog_req*
CATALOG	CLIENT	`send`	`async`	*training_catalog*
CATALOG	CLIENT	`send`	`async`	*unable_to_find_catalog*
...	...	...	...	...
...	...	...	...	...
TRAINING	CATALOG	`receive`	`async`	*training_catalog_req*
CATALOG	TRAINING	`send`	`async`	*training_catalog*
CATALOG	TRAINING	`send`	`async`	*unable_to_find_catalog*

Tabella 3.1: Payload Record Comportamentali Catalog

Sorgente	Destinazione	Tipo Msg	Comunicazione	Payload
CLIENT	TRAINING	`receive`	`async`	*training_req*
TRAINING	CLIENT	`send`	`async`	*req_accepted*
TRAINING	CLIENT	`send`	`async`	*bad_request*
TRAINING	CLIENT	`send`	`async`	*internal_server_error*
...	...	...	...	...
...	...	...	...	...
TRAINING	SESSION	`send`	`async`	*train_data_record*
...	...	...	...	...
...	...	...	...	...
TRAINING	CATALOG	`send`	`async`	*training_catalog_req*
CATALOG	TRAINING	`receive`	`async`	*training_catalog*
CATALOG	TRAINING	`receive`	`async`	*unable_to_find_catalog*

Tabella 3.2: Payload Record Comportamentali Training

Sorgente	Destinazione	Tipo Msg	Comunicazione	Payload
CLIENT	EVALUATION	`receive`	`async`	*eval_req*
EVALUATION	CLIENT	`send`	`async`	*req_accepted*
EVALUATION	CLIENT	`send`	`async`	*req_rejected*
EVALUATION	CLIENT	`send`	`async`	*internal_server_error*
...	...	...	...	...
...	...	...	...	...
EVALUATION	SESSION	`send`	`async`	*query_record*
SESSION	EVALUATION	`receive`	`async`	*eval_dto*
SESSION	EVALUATION	`receive`	`async`	*sess_not_exists*
SESSION	EVALUATION	`receive`	`async`	*internal_server_error*
EVALUATION	SESSION	`send`	`async`	*update_session*

Tabella 3.3: Payload Record Comportamentali Evaluation

Sorgente	Destinazione	Tipo Msg	Comunicazione	Payload
CLIENT	SESSION	`receive`	`async`	*save_session*
SESSION	CLIENT	`send`	`async`	*req_accepted*
SESSION	CLIENT	`send`	`async`	*req_rejected*
SESSION	CLIENT	`send`	`async`	*internal_server_error*
CLIENT	SESSION	`receive`	`async`	*view_sessions*
SESSION	CLIENT	`send`	`async`	*sess_summary*
SESSION	CLIENT	`send`	`async`	*unable_to_fetch_data*
...	...	...	...	...
...	...	...	...	...
TRAINING	SESSION	`receive`	`async`	*train_data_record*
...	...	...	...	...
...	...	...	...	...
EVALUATION	SESSION	`receive`	`async`	*query_record*
SESSION	EVALUATION	`send`	`async`	*eval_dto*
SESSION	EVALUATION	`send`	`async`	*sess_not_exists*
SESSION	EVALUATION	`send`	`async`	*internal_server_error*
EVALUATION	SESSION	`receive`	`async`	*update_session*
...	...	...	...	...
...	...	...	...	...
SESSION	STORAGE	`send`	`async`	*archive_data*

Tabella 3.4: Payload Record Comportamentali Session

Sorgente	Destinazione	Tipo Msg	Comunicazione	Payload
CLIENT	STORAGE	`receive`	`async`	*view_experiments*
STORAGE	CLIENT	`send`	`async`	*storage_summary*
STORAGE	CLIENT	`send`	`async`	*unable_to_fetch_data*
...	...	...	...	...
...	...	...	...	...
SESSION	STORAGE	`receive`	`async`	*archive_data*

Tabella 3.5: Payload Record Comportamentali Storage

3.4 Layer di Persistenza

Il Layer di Persistenza dell'applicazione è rappresentato dai database degli Esperimenti e di Sessione.

La logica di utilizzo dei due database è quella prescritta del design pattern *Session State*.

Per tanto, il DB di Sessione conterrà i record non ancora confermati dall'utente.

Dualmente, il DB degli Esperimenti conterrà i record già confermati dall'utente.

Se l'utente farà commit di un record di sessione, accade che quest'ultimo viene rimosso dal DB Session e scritto sul DB degli Esperimenti.

Lo schema dei dati utilizzato per entrambi i database è il seguente:

- **_id**
  - **Tipo**: int
  - **Commenti**, chiave del record, timestamp richiesta di training
- **train_data**
  - **Tipo**: Training
    Training:
    * `dataset : Dataset`
      · `dataset_name : string`
      · `dataset_task : string`
      · `split_test : float` $\wedge$ $split\_test \in [0, 1]$
      · `split_seed : int`
    * `model : Model`
      · `model_name : string`
      · `model_task : string`
      · `model_checkpoint : binary`
      · `model_hyperparams : List[HyperParam]`
      $\longrightarrow$ **HyperParam**:
      { "hyper_param_name" : str , "hyper_param_value" : float | bool | string | List[str] | List[float] | List[int] }

    * `learning : Learning`
        · `loss : string`
        · `learning_algorithm: string`
        · `learning_hyperparams : List[HyperParam]`
        $\longrightarrow$ **HyperParam**:
        { "hyper_param_name" : str , "hyper_param_value" : float | bool | string | List[str] | List[float] | List[int] }
  - **Commenti**: record contenente tutti i settaggi di training scelti dal client

- **eval_data**
  - **Tipo**: List[MetricOutcome]
    * **MetricOutcome**: `{ "metric_name" : str , "metric_value" : float }`
  - **Commenti**: lista di coppie chiave-valore dove la chiave è il nome della *metrica* e il valore è il risultato su tale metrica del Modello.

Capitolo 4

Scelte Progettuali

4.1 Stack Software Generale

Lo stack software utilizzato per lo sviluppo è basato sul linguaggio di Programmazione Python.

Quest'ultimo è stato impiegato sia per la parte client che per la parte server del progetto.

La scelta di Python è riconducibile all'enorme vastità di librerie disponibili assieme alla semplicità di codifica che lo contraddistingue.

Attorno al già citato linguaggio, è stato impiegato il seguente stack software:

Service	Component
SERVER	Python
CLIENT	Python
MULTI-PROCESSING	multiprocessing
THREADING	threading
ASYNC-PROCESSING	asyncio concurrent.futures
IPC	multiprocessing.Queue
ITCC	janus
ICC	asyncio.Queue
HTTP SYNC REQ	requests
HTTP ASYNC REQ	aiohttp
SERIALIZATION ENG	json + pickle
CRYPTOGRAPHIC LIB	cryptographic.fernet
MACHINE LEARNING	scikit-learn
SOFTWARE VERSIONING	git

Figura 4.1: Stack Software utilizzato per lo sviluppo

La lista mostrata evidenzia una serie di scelte ormai consolidate (es. *git*) assieme ad altre che meritano una disamina, ovvero la libreria di *async-processing* chiamata *asyncio*.

Prima di illustrarne il funzionamento, occorre fare una premessa; quest'ultima è riconducibile alle motivazioni che hanno spinto lo sviluppo di tale tipologia di libreria.

Essenzialmente, uno fra i più grandi difetti di Python è l'essere un linguaggio *single-thread*.

Ciò significa, che non è possibile *eseguire Parallelamente* due o più thread nonostante l'hardware supporti la funzionalità.

Il suddetto vincolo è una diretta conseguenza del modo in cui è sviluppato l'interprete Python.

Più in dettaglio, Python ,nella sua implementazione canonica ovvero `CPython`, utilizza il *Reference Counting* per tenere traccia della memoria allocata durante il ciclo di vita dell'applicazione.

La suddetta tecnica (che tra l'altro non prevede il *Garbage Collector* tipico di linguaggi come Java) prescrive il conteggio dei riferimenti ad ogni variabile istanziata.

Se il conteggio arriva a zero, la memoria viene deallocata.

Il meccanismo appena illustrato aiuta parecchio le perfomance dei codici Single-Thread.

Nonostante ciò, l'impiego di tale tecnica porta dei grossi rischi di *race-condition* nei casi di codici Multi-Thread.

Una soluzione, sarebbe impostare un lock sulle strutture dati condivise, ma fare ciò significherebbe andare incontro ad un'altra tipologia di problema, i *Deadlock*.

Per tanto, gli sviluppatori di Python hanno pensato di inserire un *lock globale* sull'interprete (GIL, global interpreter lock).

Di conseguenza, un solo thread può acquisire l'interprete e quindi modificare le strutture dati condivise con gli altri thread.

La scelta di inserire un lock sull'interprete porta dei benefici assieme ad una serie di svantaggi tra cui la già citata limitazione del solo thread in esecuzione in un singolo istante di tempo.

Dal punto di vista dello sviluppatore, esistono due soluzioni per aggirare *parzialmente* il problema:

- **Multi-Processing**, costruzione di più processi che evolvono in parallelo.

  Il Multi-Processing ha i suoi vantaggi assieme ad una serie di svantaggi come il maggior consumo di memoria rispetto al Multi-Threading.

- **Async-Processing**, l'idea del paradigma è implementare un *Multi-Tasking Cooperativo* al livello del singolo Thread in esecuzione.

  Essendo un multi-tasking cooperativo, saranno le routine (o meglio le *corutine*) del multi-tasking a cedersi la CPU utilizzando uno schema *NON Competitivo*.

  In tutti i casi, l'idea rimane quella di eseguire un codice Single-Thread, ma se vengono implementati dei context-switch *veloci e slegati dal SO* si possono ottenere *prestazioni* paragonabili al Multi-Threading.

Chiaramente, l'approccio funziona bene con i flussi di lavoro I/O Bound, mentre funziona molto male con i flussi di lavoro CPU-Bound ad alto tasso di Parallelizzazione.

Per tanto, i contesti applicativi dell'async-processing rimangono i task che si interfacciano con la rete, con i file oppure con l'Utente.

Per i task ad alto utilizzo della CPU, è consigliato utilizzare il paradigma del Multi-Processing.

Detto ciò, è proprio tale paradigma ad essere implementato dalla libreria *asyncio*.

Andando ad analizzare la libreria, possiamo dire che quest'ultima è stata costruita attorno ai seguenti 5 concetti:

1. **Event-Loop**, rappresenta lo scheduler delle cortutine.

   Essenzialmente, è un componente che iterativamente schedula quale corutine eseguire oppure risponde alla ricezione dei segnali dall'esterno.

2. **Corutine**, la corutine è un flusso di lavoro del già citato Multi-Tasking cooperativo.

   Più pragmaticamente, parliamo di una funzione che può interrompere la sua esecuzione restituendo il controllo all'Event-Loop che a sua volta deciderà a quale altra corutine dare la CPU.

   Chiaramente, una corutine interrompe la sua esecuzione perchè è in attesa di un particolare evento; per tanto quando quest'ultimo sarà accaduto succede che la corutine viene marcata come *ready* e per tanto ne verrà schedulata l'esecuzione alla prossima iterazione dell'Event-Loop.

3. **Attesa Asincrona**, l'attesa asincrona è lo strumento che permette di interrompere l'esecuzione di una corutine e restituire il controllo all'event-loop.

   Nell'implementazione asyncio, ciò corrisponde alla keyword `await` anteposta alle funzioni che *supportano* il meccanismo appena raccontato.

4. **Coda Asincrona**, le code asincrone sono lo strumento built-in che permette la comunicazione *asincrona* fra corutine.

   Per tanto, è possibile condividere *in maniera safe* una coda fra più corutine.

   Inoltre, sarà possibile *attendere in maniera asincrona* la ricezione di un messaggio dalla coda.

5. **Libreria Nativa**, asyncio è una libreria nativa del linguaggio, per tanto rappresenta un builiding-block su cui sono state costruite molte altre librerie.

   Più in generale, qualunque codice asincrono Python si appoggia ai meccanismi della libreria.

4.2 Stack Software per il Backend

Lato Backend, è stato impiegato il seguente stack software:

Service	Component
REST END-POINT	fastAPI
WEB SERVER	uvicorn
MESSAGE BROKER	aio-pika { rabbitMQ }
EVENT STORE	aiokafka confluent-kafka
DATA STORE	motor { mongoDB }
TEST	pytest pytest-asyncio unittest
DOC GEN	fastAPI { OpenAPI } sphinx pdoc3
DEPLOY ENV	docker

Figura 4.2: Stack Software utilizzato per lo sviluppo della parte Server dell'applicazione

In merito al REST End-Point, è stata scelta la recente libreria *fastapi*. Quest'ultima, porta in dote le seguenti peculiarità:

1. **Web Server Asincrono**, fastapi utilizza *uvicorn* come Web Server.

   Quest'ultimo supporta il già citato meccanismo di *Multi-Tasking Cooperativo*.

   Per tanto, sarà possibile attendere in maniera asincrona gli eventi scaturiti dalle richieste effettuate dagli utenti.

2. **Validazione Automatica dei Dati**, fastapi implementa un meccanismo di definizione degli schemi in merito ai dati ricevuti dal frontend.

   Più pragmaticamente, sarà necessario definire una classe avente una serie di *campi tipati* rappresentanti lo schema che un input deve avere per passare i controlli.

Per tipare i campi, si vanno a sfruttare le funzionalità di *Type Annotation* presenti nelle ultime release di Python.

Chiaramente, gli schemi sono utilizzabili sia per i parametri POST che per i parametri di Query che per i parametri dell'URL.

Per quanto detto, associare uno schema ad un parametro di una richiesta significa impostare un controllo automatico di tipo su quest'ultimo.

Se il controllo non va a buon fine, viene restituito automaticamente un errore HTTP al client.

3. **Generazione Automatica Documentazione**, fastapi genera automaticamente la documentazione delle API REST andando a sfruttare lo standard OpenAPI.

Le comunicazioni HTTP non sono l'unico strumento impiegato all'interno del Backend.

Un ulteriore servizio utilizzato è la *messaggistica*.

La messaggistica è un servizio cloud che permette la comunicazione asincrona con *lasco accoppiamento* fra componenti software.

In pratica, l'idea è di inoltrare una serie di messaggi ad un Message Broker.

Ogni messaggio, sarà indirizzato ad un apposita Coda; è proprio tale caratteristica che promuovo il lasco accoppiamento fra mittente e destinatario dei messaggi.

In tutti i casi, sarà compito del Broker il corretto e affidabile recapito dei messaggi ai destinatari.

Un ulteriore compito del broker è quello di passivare i messaggi su disco in modo da evitare perdite di dati in caso di interruzioni di alimentazione.

Nel progetto corrente, è stato scelto RabbitMQ come Message Broker.

Di conseguenza, è stata impiegata la libreria asincrona *aio-pika* come libreria *rabbitmq* su Python.

Dal punto di vista del monitoraggio, è stato impiegato un Event-Store.

Quest'ultimo è una particolare tipologia di database adatta alla registrazione di informazioni a flusso continuo come quelle presenti nelle applicazioni di Streaming.

Nel corrente progetto, è stato scelto *Apache Kafka* come Event-Store.

Kafka è una piattaforma di streaming distribuita che implementa numerosissime funzionalità tra cui quella dell'Event-Store.

Nella sua essenza, Kafka è basato sui seguenti 6 concetti:

1. **Record**, un record rappresenta l'unità elementare di informazione che viene scritta all'interno di Kafka.

   Più precisamente, un record contiene le seguenti informazioni:

   (a) **Chiave**, chiave del record in formato binario, utile per indicizzare i record.

   (b) **Timestamp**, timestamp UNIX creazione del record

   (c) **Payload**, dati in formato binario rappresentanti l'informazione inserita all'interno del record.

2. **Topic**, rappresenta un aggregatore di record appartenenti ad una specifica *Categoria*.

3. **Partizione**, è l'unità *elementare* che compone i topic.

   Più semplicemente, una partizione è una collezione *immutabile* di record ordinati cronologicamente e appartenenti allo stesso Topic.

   Al livello applicativo, saranno proprio le partizioni a fornire l'accesso ai record contenuti in un topic.

   Inoltre, le partizioni permettono anche di incrementare il parallelismo nella lettura/scrittura dei record.

   Tutto ciò, in virtù della possibilità di replicare quest'ultime su più macchine.

4. **Consumer Group**, un consumer group è un raggruppamento di Consumatori Kafka.

   La caratterizzazione di ogni raggruppamento è che verrà assegnata una *singola partizione* ad ogni singolo consumatore del Consumer Group.

   Tutto ciò può essere utile per distinguere i consumatori di applicazioni differenti.

5. **Log Transazionale**, dal punto di vista più tecnico, possiamo dire che Kafka è un sistema di logging transazionale e distribuito.

   Per tanto, possiamo vedere una partizione come uno stream di record che facilmente raggiunge dimensioni ragguardevoli.

   Per facilitare l'esplorazione dello stream, Kafka mette a disposizione una serie di puntatori abbinati alle partizioni.

   Uno fra i puntatori più importanti è quello di *commit*.

   Quest'ultimo, è disponibile per ogni Consumer Group e punta all'ultimo messaggio letto da un qualunque consumatore di ogni Consumer Group.

   Ciò permette di riprendere la lettura dei record senza dover rileggere completamente tutto il log.

6. **Broker**, insieme di 1 o più server che permettono la gestione dei record, la consegna dei record ai consumatori oppure la loro scrittura sulla specifica partizione.

Dal punto di vista delle librerie Python, sono state scelte le seguenti librerie:

1. **aiokafka**, libreria asincrona che permette di interfacciarsi con Kafka.

   Essendo asincrona, è stata sfruttata per la parte di logging del sistema.

2. **Confluent-Kafka**, libreria sincrona che permette di interfacciarsi con Kafka.

   Utilizzata all'interno della console di amministrazione in virtù delle maggiori opzioni disponibili rispetto ad aiokafka.

Continuando sul layer di persistenza, è stato scelto *MongoDB* come database per archiviare le informazioni degli esperimenti generati dall'applicazione.

La scelta è dettata in prima battuta dall'enorme versatilità di Mongo con i dati a *Schema Variabile*.

Un'ulteriore motivazione è riconducibile al fatto che Mongo è basato sul formato di interscambio JSON che per inciso ha un ottimo supporto nei tipi built-in Python.

La specifica libreria scelta è stata *Motor*.

Motor è un client MongoDB per Python che sfrutta le potenzialità del già citato asyncio, per tanto è un API asincrona.

Entrando in argomento testing, possiamo dire che la libreria di riferimento è stata in larga parte *Pytest*.

Quest'ultima è una libreria che permette di automatizzare la scrittura ed esecuzione degli *Unit Test*.

La libreria supporta anche l'esecuzione di test asincroni.

Detto ciò, l'ultima libreria che rimane da analizzare è *Sphinix*.

Sphinix è una libreria che permette la generazione automatica della documentazione HTML a partire dal codice sorgente.

Per fare ciò, vengono sfruttati dei *particolari* commenti chiamati *docstring*.

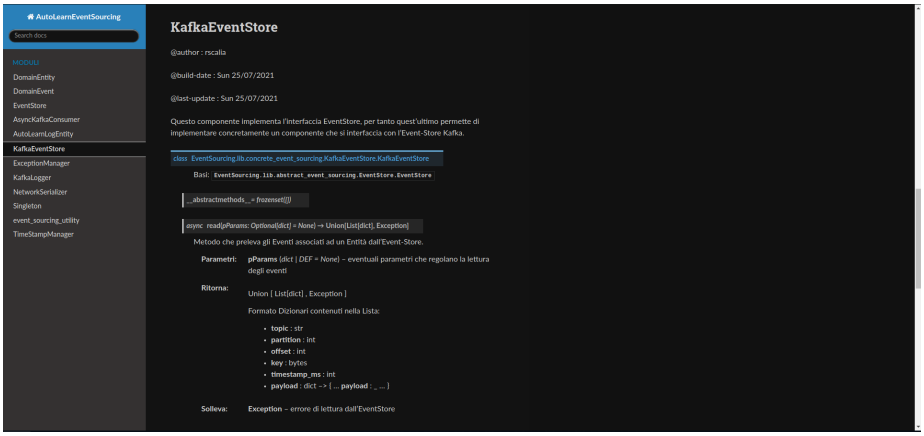

Figura 4.3: Esempio Documentazione Generata con Sphinix

Capitolo 5

Architettura del Software

Il software sviluppato sfrutta i seguenti design pattern/framework:

1. Session State
2. Remote Proxy
3. Forward-Receiver
4. Remote Facade
5. Data Transfer Object
6. Serialized LOB
7. Streaming Pipeline
8. Event Sourcing
9. Singleton

Il design pattern *Session State* è stato implementato attraverso la già citata separazione fra DB di Sessione e DB Esperimenti.

Il design pattern *Remote Facade* viene implementato *in senso lato* dai REST End-Point dei microservizi.

Del resto, ogni REST End-Point raccoglie più informazioni da restituire in un unico oggetto; tutto ciò simula quanto fatto dai *metodi bulk accessor* del design pattern.

Un ulteriore design pattern del progetto è il *Data Transfer Object*, quest'ultimo prescrive la serializzazione di un oggetto in modo tale da poter trasmettere le sue informazioni (attributi) in rete.

Il suddetto pattern è stato implemento con una piccola variazione, nello specifico è stata scelta una particolare tipologia di oggetti da serializzare: i *Dizionari Python*.

La tecnica di serializzazione impiegata è quella testuale e il rispettivo formato è il *JSON*.

Restando in tema serializzazione, possiamo enunciare l'utilizzo del Design Pattern *Serialized LOB*.

Serialized LOB è stato impiegato per serializzare i checkpoint dei Modelli di ML prima di scriverli sul DB.

Più in dettaglio, è stata scelta la serializzazione in binario per tale task.

Un altro design pattern impiegato per il progetto è il *Singleton*.

Quest'ultimo è un pattern che impedisce istanziazioni ripetute della stessa classe per tutto il ciclo di vita dell'applicativo.

5.1 Remote Proxy + Forward-Receiver

I design-pattern *Remote Proxy e Forward-Receiver* sono stati implementati *solamente* sul lato client dell'applicazione.

Di seguito viene mostrato il diagramma UML dei pattern:

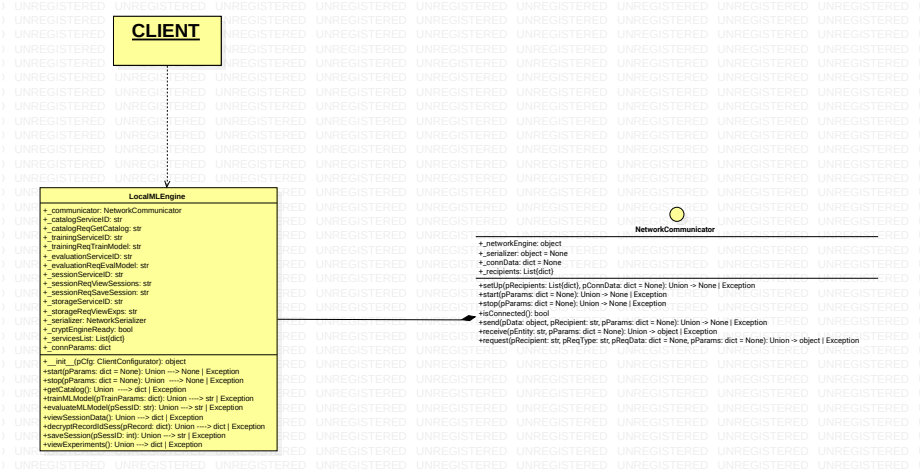

Figura 5.1: Diagramma UML per DP Remote Proxy e Forward-Receiver

Lo schema mostrato esibisce le seguenti caratteristiche:

- La classe `LocalMLEngine` interpreta il ruolo di *Remote Proxy*.
- L'interfaccia `NetworkCommunicator` permette di implementare la suddivisione in *Peer* dettata dal design-pattern *Forward-Receiver*.

  `NetworkCommunicator` rappresenta anche un'interfaccia standard per l'implementazione di una connessione remota.

  Nello specifico, è supportato sia lo scambio di messaggi fra processi (vedi metodi `send` e `receive`) che l'inoltro di richieste *RPC* verso una macchina remota (vedi metodo `request`).

  Inoltre, è di utilità sottolineare il fatto che l'interfaccia sia generica rispetto alle *tecnologie di serializzazione e di connessione*.

  Infine, si vuole segnalare la presenza di una lista di *destinatari* all'interno del componente.

  Quest'ultima può essere utile per l'implementazione di eventuali politiche di filtraggio dei messaggi.

5.2 Event Sourcing

Il design-pattern *Event Sourcing* è lo strumento che ha permesso l'implementazione del meccanismo di *Monitoraggio Comportamentale dei Microservizi*.

La filosofia dell'Event Sourcing è condensata nell'idea di registrare tutti i cambiamenti (o *eventi*) accaduti ad un'entità di dominio a partire da uno stato iniziale.

Si osservi come tale filosofia sia diametralmente opposta rispetto a quella *canonica* impiegata nella progettazione del software.

Del resto, l'approccio canonico implicherebbe la sola memorizzazione dell'ultimo stato dell'entità andando quindi a perdere tutti i cambiamenti intermedi.

In generale, l'Event Sourcing trova applicazione nelle soluzioni software in cui il *tempo* è cruciale rispetto ad altri parametri.

Per quanto detto, possiamo dire che gli eventi accaduti ad un entità sono *indispensabili* per poter ricostruirne il comportamento (e lo stato).

Di conseguenza, è stata implementata una strategia di *persistenza* di quest'ultimi ond'evitare perdite di informazioni causate da interruzioni di alimentazione.

La suddetta strategia, prevede il salvataggio degli eventi su una particolare tipologia di database: gli *Event-Store*.

La scelta di questo tipo di DB è riconducibile alla sua ottimizzazione nel gestire cospicui stream di dati *da e verso* la base di dati.

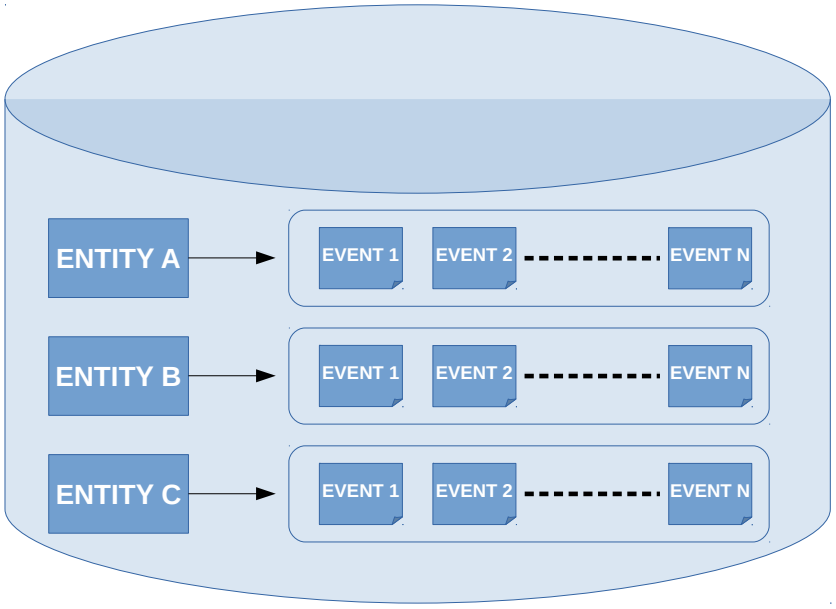

Figura 5.2: Rappresentazione Grafica Event-Store

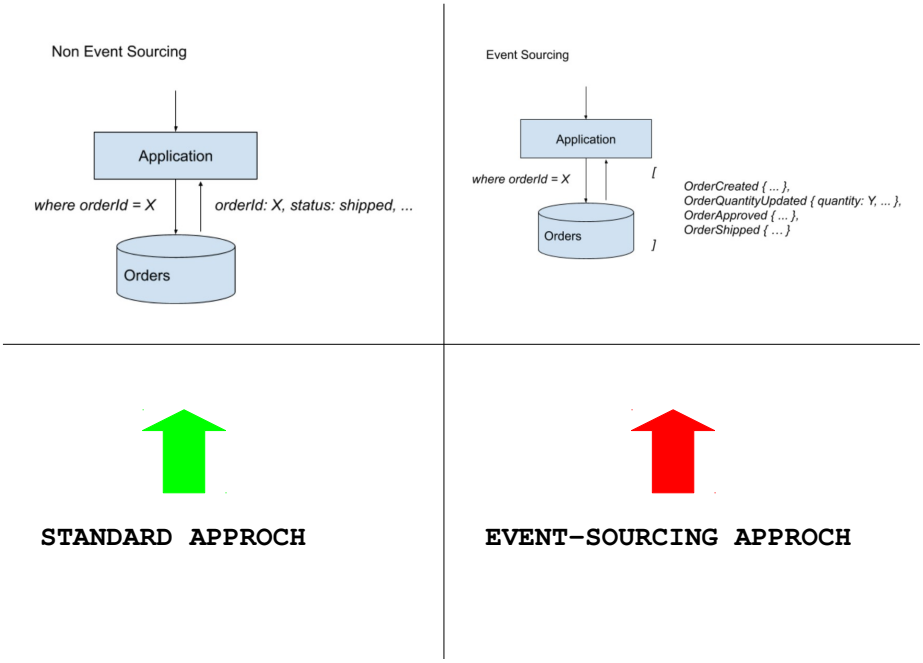

Figura 5.3: Confronto NON Event Sourcing vs Event Sourcing

Dal punto di vista logico, il pattern promuove le seguenti *tre operazioni* per ogni Entità:

- **Create**, permette di costruire una nuova entità, sia in memoria che sull'Event-Store.
- **Rewind**, l'operazione di `rewind` permette di ripristinare lo stato *passato* di un'entità.

  Più in dettaglio, vengono eseguite le seguenti operazioni:

  1. Prelievo dall'event-store di tutti gli eventi accaduti all'entità a partire dallo stato iniziale.
  2. Riapplicazione eventi all'entità in ordine cronologico *evitando la loro riscrittura* su event-store.
- **Emit**, l'operazione di `emit` permette di applicare un evento ad un'entità e per tanto commutarne lo stato.

  Quest'ultima operazione necessità i seguenti passaggi:

  1. Emissione Evento da parte dell'entità, permette di costruire l'oggetto contenente tutte le informazioni che permettono di caratterizzare l'evento in maniera univoca.
  2. Applicazione Evento, permette di commutare lo stato dell'entità e contestualmente scrivere il nuovo evento accaduto sull'Event-Store.

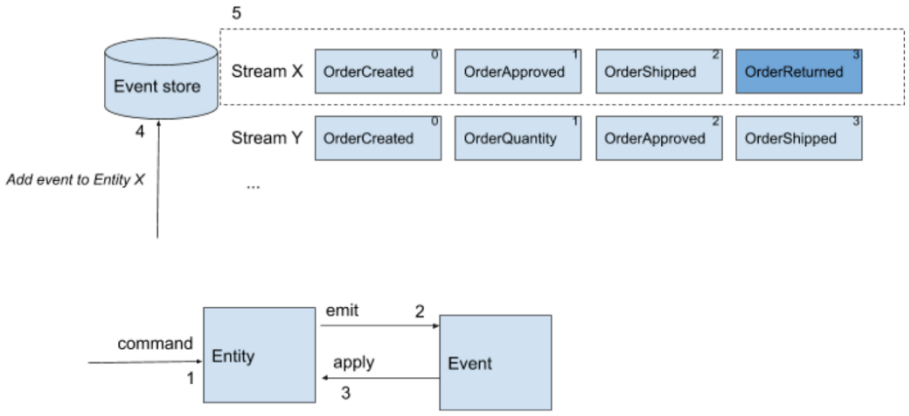

Figura 5.4: Rappresentazione Grafica Logica Commutazione Stato Entità

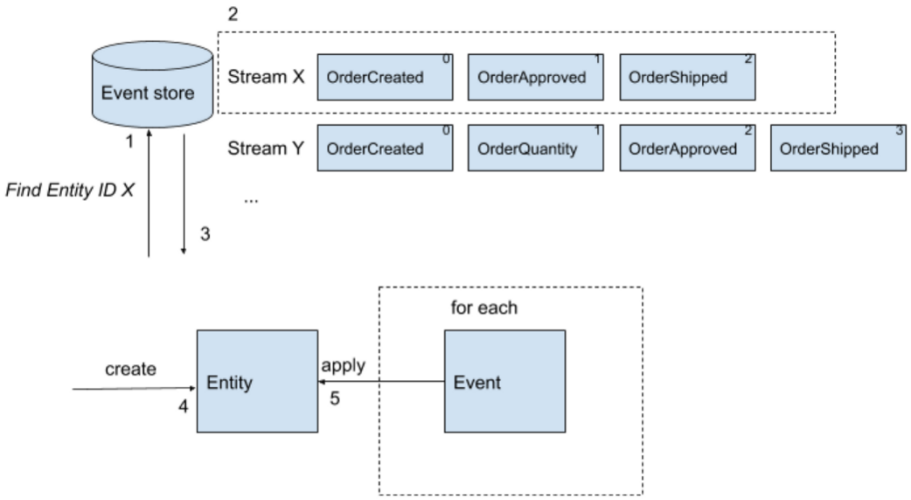

Figura 5.5: Rappresentazione Grafica Logica Rewind

I ruoli del design pattern sono illustrati di seguito:

- `DomainEntity`, rappresenta un'*entità di dominio* dell'applicazione.

  Quest'ultima possiede uno stato *iniziale* che è soggetto a cambiamenti durante il ciclo di vita dell'applicazione.

  I suddetti cambiamenti sono *innescati* ,da appositi *eventi*, attraverso il metodo `emit()`.

  L'interfaccia supporta anche l'operazione di *Rewind* degli eventi attraverso l'omonimo metodo `rewind()`.

  Dal punto di vista dello stato, *DomainEntity* mantiene la lista degli eventi (`_eventList`) e lo stato attuale `_status` assieme ad un puntatore all'oggetto che si interfaccia con l'event-store (`_eventStore`).

- `DomainEvent`, rappresenta un record comportamentale contente tutte le informazioni che contraddistinguono un evento.

  Nello specifico, un evento è contraddistinto da:

  1. `_entityId`, identificativo dell'entità a cui viene applicato l'evento.
  2. `_eventTimeStamp`, timestamp UNIX applicazione evento all'entità.
  3. `_eventPayload`, ulteriori dati di dominio associati all'evento.

- `EventStore`, quest'interfaccia permette di leggere e scrivere Eventi sull'Event-Store.

  Per fare ciò, vengono sfruttati degli appositi oggetti chiamati `_readModel` e `_writeModel`.

  La caratterizzazione dei suddetti oggetti necessità la scelta dello specifico event-store, per tanto ,in tale sede, ne verrà evitata la disamina.

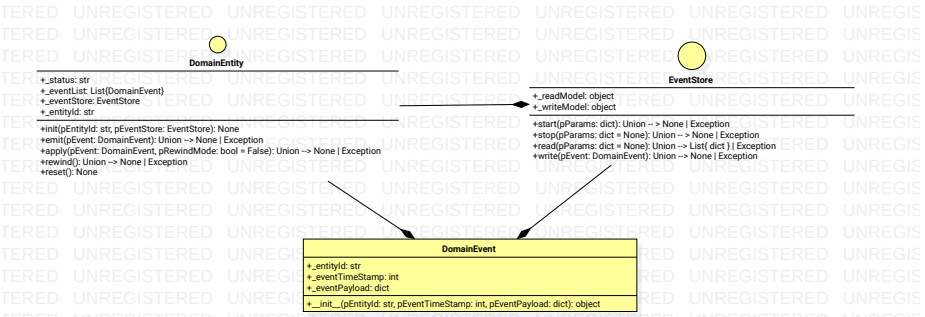

Figura 5.6: Diagramma UML per DP Event Sourcing

5.3 Streaming Pipeline

Il design-pattern *Streaming Pipeline* permette l'implementazione di un job sequenziale formato da una serie di task, chiamati *Pipe* nel gergo del pattern.

Il design-pattern va essenzialmente ad emulare il principio di funzionamento di una *Catena di Montaggio*.

Il termine *Streaming* all'interno del nome del DP va a sottolineare la natura del flusso in ingresso alla pipeline.

Dal punto di vista architetturale, siamo di fronte ad una singola classe replicata su più istanze in modo da formare la già citata pipeline.

Lo schema UML della classe è il seguente:

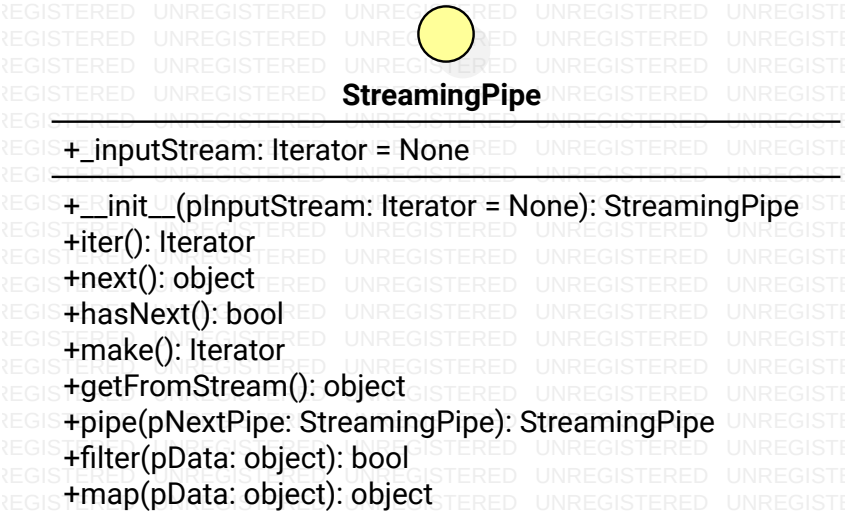

Figura 5.7: Diagramma UML per DP Streaming Pipeline

La classe mostrata rappresenta un'interfaccia di una *Pipe*.
Quest'ultima espone le seguenti feature:

- **Self-Iterator**, l'interfaccia *StreamingPipe* è un Iteratore.

  Tutto ciò in virtù dei metodi:

  - `iter()`, permette di ottenere l'*Iteratore Interno*.
  - `next()`, permette di iterare sull'*iteratore Interno*.
  - `hasNext()`, permette di verificare se è possibile iterare sull'*iteratore Interno*.

- **Iteratore Interno**, l'iteratore interno di una StreamingPipe è l'iteratore su cui saranno prelevati realmente i dati.

  Del resto, una chiamata al metodo `next()` implica anche l'estrazione del prossimo elemento dal suddetto Iteratore Interno.

- **Iteratori Annidati**, la classe StreamingPipe è formata da due *"Iteratori Annidati"*: il *Self-Iterator* e l'*Iteratore Interno*.

  L'*Iteratore Interno* è l'iteratore che permette il prelievo del prossimo elemento da elaborare.

  Il *Self-Iterator* è l'iteratore associato alla classe e quindi al task.

  Un'iterazione su quest'ultimo implica la seguente elaborazione:

  1. Prelievo del prossimo elemento da elaborare dall'iteratore interno.
  2. Filtraggio elemento prelevato.
  3. Trasformazione elemento filtrato.

4. Restituzione elemento elaborato come prossimo elemento del *Self-Iterator*.

- **Pipe fra Task**, l'interfaccia *StreamingPipe* supporta il chaining dei task in modo da costruire una *Pipeline di Task*.

  La suddetta pipeline viene costruita col metodo `pipe()`.

  Il metodo `pipe` permette di collegare l'iteratore interno presente in un task con il Self-Iterator dell'oggetto che sta invocando il metodo Pipe.

  In pratica, stiamo connettendo un task al successivo andando a formare una pipeline.

  In questo modo, un `next` sull'ultimo anello della pipeline implica l'elaborazione sequenziale di tutti gli altri anelli che vengono prima nella pipeline.

  Quando gli altri step della pipeline avranno finito, potrà avvenire l'elaborazione dell'ultimo anello.

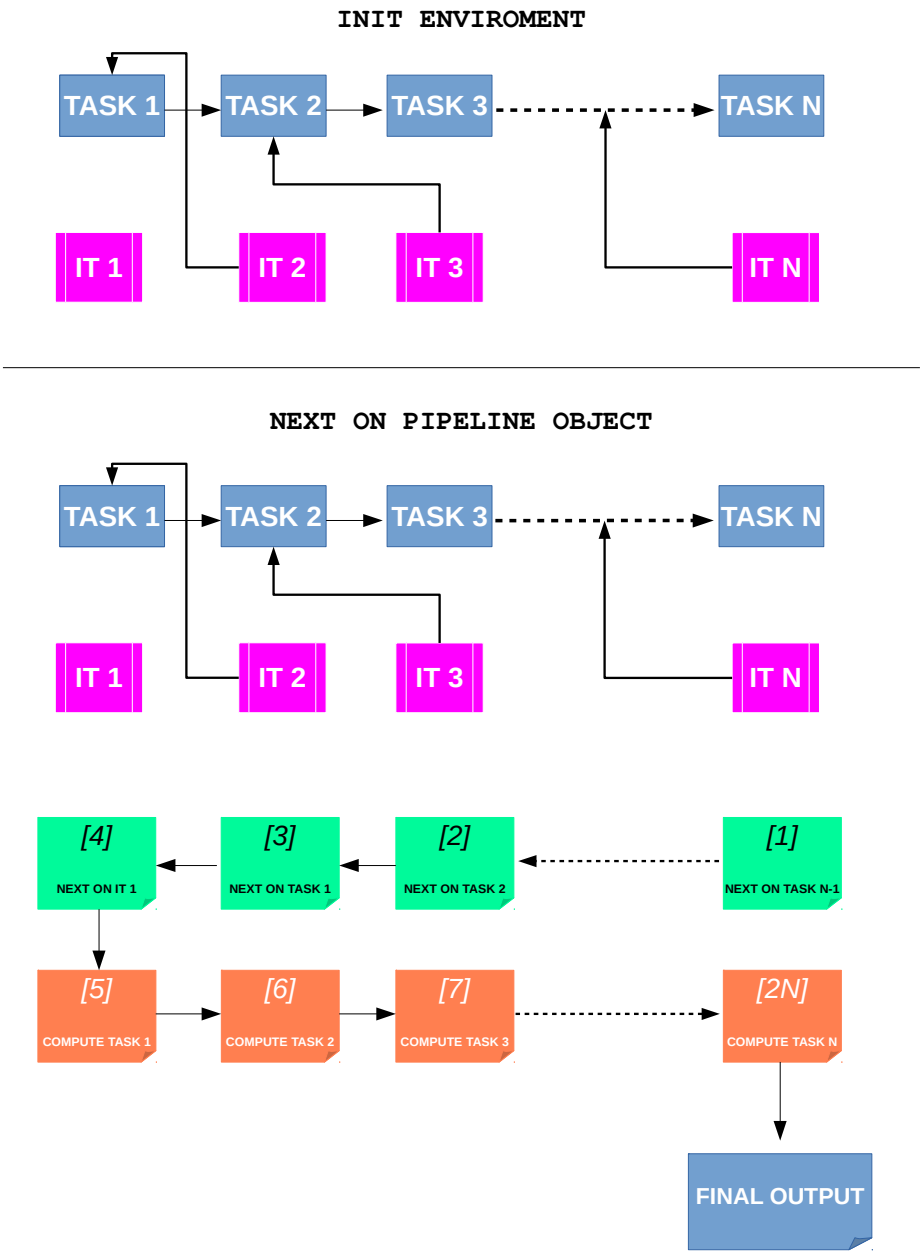

Figura 5.8: Schema Computazione DP Streaming Pipeline

5.4 MLEngine

Il framework MLEngine rappresenta uno schema standard per aggiungere al progetto nuovi Modelli o Dataset.

Il framework supporta anche l'aggiunta di *Loss Function* e *Algoritmi di Ottimizzazione* al panorama attualmente sviluppato.

Di seguito, viene illustrato il diagramma UML del framework:

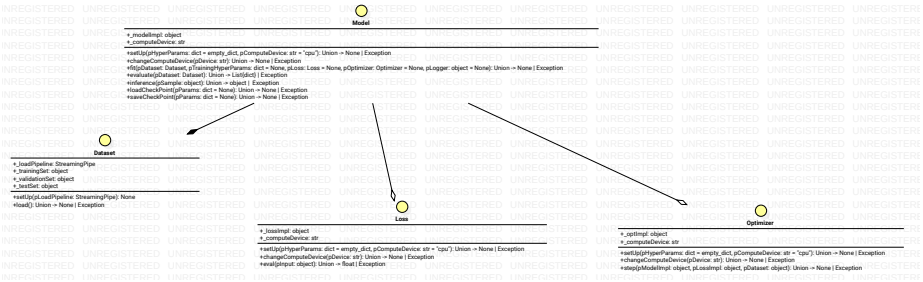

Figura 5.9: Diagramma UML per MLEngine

Lo schema mostrato evidenzia le seguenti caratteristiche:

- Il framework è agnostico dal punto di vista dell'implementazione dei Modelli, delle Loss e degli Optimizer.

  Per tanto, si potrà scegliere una qualunque libreria di ML come *Spark* o *PyTorch*.

- Il framework supporta l'esecuzione su un'ampia classe di hardware tra cui:

  - CPU
  - TPU
  - GPU
  - FPGA

  Per selezionare il device, viene esposto il metodo `changeComputeDevice` sulle classi `Model`, `Loss` e `Optimizer` oltre che all'apposito parametro da passare in fase di costruzione degli oggetti.

- L'interfaccia `Dataset` permette il caricamento dei set di dati attraverso una pipeline conforme al già citato *Streaming Pipeline*.

  Il componente supporta anche la predisposizione per lo splitting del dataset nei classici training, validation e test-set.

- L'interfaccia `Model` permette l'implementazione di tutte le fasi del ciclo di vita di un modello di ML, ovvero: training, validation , evaluation e inferenza.

  L'interfaccia supporta anche la scelta degli Iperparametri del Modello assieme alla possibilità di salvare/caricare un checkpoint di un Modello Addestrato su/da disco.

  Inoltre, il metodo che avvia gli addestramenti permette anche la scelta di una serie di Iperparametri di Training.

- L'interfaccia `Optimizer` rappresenta l'astrazione dell'algoritmo di learning all'interno dell'applicativo sviluppato.

Il metodo principale dell'interfaccia è `step`.

Quest'ultimo permette di eseguire un'iterazione del suddetto algoritmo

- L'interfaccia `Loss` rappresenta l'astrazione dell'obbiettivo dell'ottimizzazione all'interno dell'applicativo.

  Per tanto, la Loss sarà una funzione vettoriale $\mathcal{L} : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ da valutare.

  Il metodo che permette la sua valutazione su di un input è `eval`.

Capitolo 6

Test & Deploy della Soluzione

Il testing della soluzione sviluppata è stato affidato in larga parte agli *Unit Test*.

Quest'ultimi includono anche test appositi per l'event-sourcing.

Dal punto di vista dei test di Sistema, ne è stato scritto uno all'interno dei test del client.

Nello specifico, è stato sfruttato l'unit-test del Remote Proxy per testare l'intero sistema.

Entrando in argomento deploy, possiamo dire che ogni microservizio è stato immesso in un apposito container *Docker*.

Quanto detto porta in dote al seguente docker-compose:

```
version: '3'
services:

    config_service:
        image: 'bitnami/zookeeper:latest'
        hostname: zookeeper
        container_name: zookeeper
        restart: always
        environment:
            - ALLOW_ANONYMOUS_LOGIN=yes
        volumes:
            - ${CONFIG_SERVICE_DATA_LAKE}:/bitnami/zookeeper


    event_store:
        image: 'bitnami/kafka:latest'
        hostname: kafka
        container_name: kafka
        restart: always
        depends_on:
            - config_service
        environment:
            - KAFKA_BROKER_ID=1
            - KAFKA_INTER_BROKER_LISTENER_NAME=CLIENT
```

```
        - KAFKA_CFG_ZOOKEEPER_CONNECT=zookeeper:2181
        - ALLOW_PLAINTEXT_LISTENER=yes
        - KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=CLIENT:PLAINTEXT
        - KAFKA_CFG_LISTENERS=CLIENT://:9092
        - KAFKA_CFG_ADVERTISED_LISTENERS=CLIENT://kafka:9092
    volumes:
        - ${EVENT_STORE_DATA_LAKE}:/bitnami/kafka


broker:
    image: rabbitmq:3-management
    hostname: rabbitmq
    container_name: rabbitmq
    restart: always
    ports:
        - ${BROKER_PORT_DASHBOARD}:15672
    volumes:
        - ${BROKER_DATA}:/var/lib/rabbitmq
        - ${BROKER_LOGS}:/var/log/rabbitmq/log


storage_db:
    image: mongo
    container_name: storage_db
    hostname: storage_db
    restart: always
    environment:
        MONGO_INITDB_ROOT_USERNAME: ${STORAGE_DB_USERNAME}
        MONGO_INITDB_ROOT_PASSWORD: ${STORAGE_DB_PASSWORD}
    volumes:
        - ${STORAGE_DB_DATA}:/data/db
        - ${STORAGE_DB_CONF}:/data/configdb


session_db:
    image: mongo
    container_name: session_db
    hostname: session_db
    restart: always
    command: mongod --port ${SESSION_DB_PORT}
    environment:
        MONGO_INITDB_ROOT_USERNAME: ${SESSION_DB_USERNAME}
        MONGO_INITDB_ROOT_PASSWORD: ${SESSION_DB_PASSWORD}
    volumes:
        - ${SESSION_DB_DATA}:/data/db
        - ${SESSION_DB_CONF}:/data/configdb


storage_db_dashboard:
    image: mongo-express
```

```
    restart: always
    container_name: storage_db_dashboard
    hostname: storage_db_dashboard
    environment:
        ME_CONFIG_MONGODB_ADMINUSERNAME: root
        ME_CONFIG_MONGODB_ADMINPASSWORD: example
        ME_CONFIG_MONGODB_SERVER: storage_db
    ports:
        - "${STORAGE_DB_DASHBOARD_PORT}:8081"


session_db_dashboard:
    image: mongo-express
    restart: always
    container_name: session_db_dashboard
    hostname: session_db_dashboard
    environment:
        ME_CONFIG_MONGODB_ADMINUSERNAME: root
        ME_CONFIG_MONGODB_ADMINPASSWORD: example
        ME_CONFIG_MONGODB_SERVER: session_db
        ME_CONFIG_MONGODB_PORT: ${SESSION_DB_PORT}
    ports:
        - "${SESSION_DB_DASHBOARD_PORT}:8081"


training:
    image: py:sw_eng
    container_name: training
    hostname: training
    restart: always
    depends_on:
        - broker
        - event_store
    command: sh -c "sleep 20 && cd /workspace/code/training/ && ./exec.sh"
    environment:
        SERVER_PORT: ${TRAINING_SERVICE_PORT}
        HOST_NAME: training
        WEB_SERVER_WORKERS: ${WEB_SERVER_WORKERS}
        BROKER_LOGIN_TOKEN: ${BROKER_LOGIN_TOKEN}
        EVENT_STORE_HOST_NAME: ${EVENT_STORE_HOST_NAME}
        EVENT_STORE_PORT: ${EVENT_STORE_PORT}
    ports:
        - ${TRAINING_SERVICE_PORT}:${TRAINING_SERVICE_PORT}
    volumes:
        - ${TRAINING_CODE_LAKE}:/workspace/code
        - ${TRAINING_DATA_LAKE}:/workspace/data


evaluation:
    image: py:sw_eng
```

```
    container_name: evaluation
    hostname: evaluation
    restart: always
    depends_on:
        - broker
        - event_store
    command: sh -c "sleep 20 && cd /workspace/code/evaluation/ && ./exec.sh
    environment:
        SERVER_PORT: ${EVALUATION_SERVICE_PORT}
        HOST_NAME: evaluation
        WEB_SERVER_WORKERS: ${WEB_SERVER_WORKERS}
        BROKER_LOGIN_TOKEN: ${BROKER_LOGIN_TOKEN}
        EVENT_STORE_HOST_NAME: ${EVENT_STORE_HOST_NAME}
        EVENT_STORE_PORT: ${EVENT_STORE_PORT}
    ports:
        - ${EVALUATION_SERVICE_PORT}:${EVALUATION_SERVICE_PORT}
    volumes:
        - ${EVALUATION_CODE_LAKE}:/workspace/code
        - ${EVALUATION_DATA_LAKE}:/workspace/data


session:
    image: py:sw_eng
    container_name: session
    hostname: session
    restart: always
    depends_on:
        - broker
        - event_store
        - session_db
    command: sh -c "sleep 20 && cd /workspace/code/session/ && ./exec.sh"
    environment:
        SERVER_PORT: ${SESSION_SERVICE_PORT}
        HOST_NAME: session
        WEB_SERVER_WORKERS: ${WEB_SERVER_WORKERS}
        SESSION_DB_HOST_NAME: ${SESSION_DB_HOST_NAME}
        SESSION_DB_PORT: ${SESSION_DB_PORT}
        SESSION_DB_USERNAME: ${SESSION_DB_USERNAME}
        SESSION_DB_PASSWORD: ${SESSION_DB_PASSWORD}
        BROKER_LOGIN_TOKEN: ${BROKER_LOGIN_TOKEN}
        EVENT_STORE_HOST_NAME: ${EVENT_STORE_HOST_NAME}
        EVENT_STORE_PORT: ${EVENT_STORE_PORT}
    ports:
        - ${SESSION_SERVICE_PORT}:${SESSION_SERVICE_PORT}
    volumes:
        - ${SESSION_CODE_LAKE}:/workspace/code
        - ${SESSION_DATA_LAKE}:/workspace/data


storage:
```

```
    image: py:sw_eng
    container_name: storage
    hostname: storage
    restart: always
    depends_on:
        - broker
        - event_store
        - storage_db
    command: sh -c "sleep 20 && cd /workspace/code/storage/ && ./exec.sh"
    environment:
        SERVER_PORT: ${STORAGE_SERVICE_PORT}
        HOST_NAME: storage
        WEB_SERVER_WORKERS: ${WEB_SERVER_WORKERS}
        STORAGE_DB_HOST_NAME: ${STORAGE_DB_HOST_NAME}
        STORAGE_DB_PORT: ${STORAGE_DB_PORT}
        STORAGE_DB_USERNAME: ${STORAGE_DB_USERNAME}
        STORAGE_DB_PASSWORD: ${STORAGE_DB_PASSWORD}
        BROKER_LOGIN_TOKEN: ${BROKER_LOGIN_TOKEN}
        EVENT_STORE_HOST_NAME: ${EVENT_STORE_HOST_NAME}
        EVENT_STORE_PORT: ${EVENT_STORE_PORT}
    ports:
        - ${STORAGE_SERVICE_PORT}:${STORAGE_SERVICE_PORT}
    volumes:
        - ${STORAGE_CODE_LAKE}:/workspace/code
        - ${STORAGE_DATA_LAKE}:/workspace/data


catalog:
    image: py:sw_eng
    container_name: catalog
    hostname: catalog
    restart: always
    depends_on:
        - event_store
    command: sh -c "cd /workspace/code/catalog/ && ./exec.sh"
    environment:
        SERVER_PORT: ${CATALOG_SERVICE_PORT}
        HOST_NAME: catalog
        WEB_SERVER_WORKERS: ${WEB_SERVER_WORKERS}
        EVENT_STORE_HOST_NAME: ${EVENT_STORE_HOST_NAME}
        EVENT_STORE_PORT: ${EVENT_STORE_PORT}
    ports:
        - ${CATALOG_SERVICE_PORT}:${CATALOG_SERVICE_PORT}
    volumes:
        - ${CATALOG_CODE_LAKE}:/workspace/code
        - ${CATALOG_DATA_LAKE}:/workspace/data


admin:
    image: py:sw_eng
```

```
container_name: admin
hostname: admin
restart: always
depends_on:
    - event_store
command: sh -c "cd /workspace/code/admin_console/ && ./admin_console_st
tty: true
stdin_open: true
environment:
    SERVER_PORT: ${ADMIN_SERVICE_PORT}
    HOST_NAME: admin
    EVENT_STORE_HOST_NAME: ${EVENT_STORE_HOST_NAME}
    EVENT_STORE_PORT: ${EVENT_STORE_PORT}
volumes:
    - ${ADMIN_CODE_LAKE}:/workspace/code
    - ${ADMIN_DATA_LAKE}:/workspace/data
```

La configurazione mostrata evidenzia la presenza di 13 servizi.

Infine, è da sottolineare la separazione dei due DB su due container diversi; quest'ultimo scelta segue fedelmente lo *stile di sviluppo* dei Microservizi.

Conclusione

L'esperienza maturata in questo progetto porta al sottoscritto numerose nozioni nuove sia dal punto di vista delle tecnologie che dal punto di vista dei pattern di sviluppo.

Una fra le più importanti nozioni apprese è la gestione del codice *Difensivo*.

Un ulteriore nozione acquisita è lo sviluppo a microservizi, davvero comodo in ambiente distribuito.

Dal punto di vista degli sviluppi futuri, potrebbe essere utile aggiungere i seguenti requisiti al sistema già sviluppato:

- *Feature*
  - Autenticazione
  - Sito Web Applicazione
  - Sito Web Admin Console
  - Modelli di Deep Learning nel Catalogo
  - Builder Visuale per Reti Neurali all'interno del Sito dell'Applicazione
  - Monitor in tempo reale del training dei Modelli di Deep Learning:
    - Presenza di Grafici in *RealTime*
    - Possibilità di Stoppare il training
  - Upload dei dataset da parte degli Utenti con la possibilità di selezionare il pre-processing da applicare.
  - Microservizio di Inferenza
  - Possibilità di Esportare un checkpoint di un Modello Addestrato in vari formati tra cui:
    - ONNX
    - TFLite (se è stato impiegato TensorFlow per il training)
- *Software*
  - Authentication Service
  - API Gateway
  - Service Discovery
  - CircuitBreaker
  - LoadBalancer
  - Configuration Service

- Aggiunta libreria *Apache Spark* al progetto.
  Spark è una libreria di ML che sfrutta il calcolo distribuito ,e in particolare il paradigma `DataFlow`, per addestrare i Modelli di Machine Learning.
- Aggiunta librerie di Deep Learning (PyTorch e TensorFlow).
- Sostituzione della Cifratura Simmetrica con quella Asimmetrica (ad esempio, RSA).
  La sostituzione è scelta in virtù dei difetti della crittografia simmetrica in merito alla distribuzione sicura del *segreto* (chiave crittografica) che permette di cifrare/decifrare i messaggi.
- Deploy del sistema in un cluster di macchine High-Performance dotato di hardware vettoriale (es. GPU o TPU).

Con l'aggiunta delle feature elencate, il Sottoscritto crede che il progetto abbia davvero le carte in regola per diventare industrializzabile.