Internship project: Developing a scalable and adaptable search prediction system

Eduardo Alexandre Costa Morgado

june, 2020

Contents:

1 Introduction

1.1 Project description

The main objective of Hapibot's internship project is to devise and implement a search prediction system integrating machine learning[1] into an existent project, the WeThePlayers (WTP) website, as a support for a better user experience when searching in this community gaming platform.

WTP's search system is supported by Elasticsearch, being a text matching system it assigns points to data. When a user searches for a game or a review, the system will then display results related to that search, sorted by points. As a result it was noticed that when searching for a specific game there would be times where the first result would not be in any way the user's desired search, as an example Figure 1, Figure 2 and Figure 3 display the original website without this internship project (the prediction system). As can be seen, when searching for the word *god*, *the legend* and *the legend of* the user is most likely searching for the trending games *god of war* and *the legend of zelda* respectively yet that is not the first displayed result and in the case of *the legend of zelda* that game isn't even displayed. This happens because the search system sorts results based on hits in the database and not trends.

For this reason the new prediction system was implemented to work alongside the existent search system, providing more user friendly results when searching - essentially search results would improve themselves as game search trends changed.

1.2 Project main objectives

During the development of this project the main objectives were:

- Producing a machine learning model capable of NLP[2] for word prediction;
- Enabling the produced model to predict trending content based on user searches;
- Building a server responding to user search prediction requests, inferring from the model;
- Dockerize the server for easy use;
- Build a scalable version of the server for Kubernetes integration;
- Build the project API to improve usability and adaptability for further projects looking to implement the prediction system;

1.3 Welcoming company

Hapibot Studio is a digital studio, designing and developing mobile and web apps and websites behind the scenes for global companies and startups.

Hapibot helps organizations find success in the digital space, where clients take advantage of emerging technologies, online business model innovations and changes in user behaviour allowing new products and services to become smarter and capable of fostering authentic customer engagement.

From enterprise web apps, chatbots, e-commerce to deep systems integration with mobile application development and beyond, it has a wealth of experience helping global customers define, design, and deploy sophisticated and emerging platform.

[1] The machine learning model would be implemented using TensorFlow
[2] Natural Language Processing

2 State of the Art

During the development of this project we used several technologies, although some were discarded due to difficulties while integrating said technology or simply because we managed to find better solutions.

The following is the list of all technologies used during this development:

- **Python** language for project development
- **TensorFlow** framework used to build the model and start the serving process
- **Elasticsearch**'s Kibana service to store user searches, this approach was later discarded
- **Kafka** platform for model integration, serving and generation, this technology was latter discarded
- **Flask** framework to build a simple web app for the model server and generate the project's API
- **Minio** container to test and implement a cloud storage bucket
- **Redis** database to store trending user searches used to retrain new models
- **Docker** to containerize all services, facilitating deployment
- **Kubernetes** to scale our service, balancing user requests
- **Minikube** to test Kubernetes' implementation locally on a single node cluster
- **Kubeflow** to facilitate deployment of machine learning models in a Kubernetes environment, this technology was discarded due to platform restrictions
- **Luigi** to fetch data dynamically, was later discarded
- **GraphQL** a query language for server APIs used to fetch data from the WTP's database

In the following sections we will look more closely into each technology, as well as the theory behind the model.

2.1 Machine Learning

Machine learning is a branch of Data Science the develops algorithms that adjust themselves to perform better as they are exposed to more data. For this specific project neural networks were used as the machine learning algorithms.

2.1.1 Neural Networks

Neural Networks are machine learning algorithms designed to recognize **numerical** patterns[3]. Neural Networks are part of **supervised learning** algorithms[4] aggregating and connecting multiple **perceptrons**[5] [6] into layers combining input data with weights that through a series of trainings will either amplify or dampen the significance of the input establishing complex correlations between present data

[3] This way all training data, be it text or images must be translated to numerical data

[4] Learning data both provides observations and conclusions [6]

[5] Basic unit in a neural net, it possesses an input layer a processing unit with an activation function and an output layer [6]

and future data [6]. It learns from past data to infer/predict future unseen data presented in the same format/translation [6].

Neural nets no matter their architecture always possess two basic features **feedforward** and **backpropagation** [6].

During the feedforward process the neural net, through matrix dot product operations between layers and its weights will produce/return output possible values for the provided inputs that, since the learning is supervised can then be compared with the actual expected output and calculate an error used to calculate how the immediately adjacent previous layer weights have to be adjusted, this error calculation and weight adjustment steps will be backpropagated to all the network's layers in the backpropagation process [6].

During this project neural networks were used to develop the search prediction service however, the core design of the NLP network followed a RNN[6] design (at first a LSTM[7] net that later became a GRU[8] network).

2.1.2 RNN

Recurrent neural networks are designed to recognize patterns in **sequences of data** taking time and sequence into account allowing for **context inference** being crucial in NLP [7].

RNN take as their input not just the current input but also **what they have perceived previously in time** that when combined create a feedback loop between current and past decisions, creating memory in a way storing sequence information like text in the network's hidden state [7].

The carrying of memory[9] can be described as $h_t = \sigma(Wx_t + Uh_{t-1})$ which doesn't differ much from a feedforward neural net, calculating the value of the hidden layer at the moment t is a matrix addiction between the multiplication of weight matrix[10] W by the input matrix[11] x_t for the current layer and the hidden state values of the previous timestep h_{t-1} multiplied by its transition matrix U [7]. The scalar result will then be passed through an **activation function** to separate the values [7], in order to achieve a better distribution of values a **non-linear** function, the sigmoid ($\frac{1}{1+\epsilon^{-x}}$) function is usually chosen.

Since this feedback loop occurs at each step, every hidden state **contains traces not only of a previous layer but of all preceding states** [7] allowing the network to **infer context**.

2.2 Used technologies

These are the technologies that are present in the final product.

2.2.1 Python

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost on maintenance. Python supports modules and packages, encouraging modularity and reusability, its versatility allows for broadband of developments, from backend, web/mobile app development, desktop software

[6] Recurrent Neural Network
[7] Long Short Term Memory
[8] Gated Recurring Units
[9] Information about the previous words in the sequence.
[10] This weight matrix is the collection of weights between the current hidden layers and the previous one and can be seen as a scoring system to determine the relevance to be given to each path between the two layers [7].
[11] The values of each "neuron" in the current layer.

development, big data and mathematical processing and computation.

Python has an extensive list of add-ons, packages and open-source frameworks facilitating and customizing software development. Since TensorFlow was the desired deep learning library for the project machine learning model, Python became the main development language.

2.2.2 Tensorflow

TensorFlow is an **open-source library** developed by Google to tackle deep learning applications, supporting traditional machine learning functionality.

Being originally developed for large numerical computations it would prove very useful for deep learning [13] since it worked with multi-dimensional arrays or **tensors** becoming very useful when handling large amounts of data [13].

This library provides a **high-level API** which reduces the amount of code required to develop a neural network, no longer being required to manually configure all the layer matrices' operations, TensorFlow came with built-in libraries such as **Keras**[12] that would simplify and abstract all those steps [13].

To train a machine learning model to successfully execute a desired task a large amount of data is required, data that will be fed to the model in an iterative process or **epoch**, all these tasks are rather complex and would quickly overwhelme a CPU[13]. TensorFlow introduces GPU[14] support when training a model which would speed up training quite considerably [13].

A tensor can be seen as a generalization of vectors or matrices of higher dimensions [13]. For any deep learning project, the data used for training will both be large and complex, tensors help describing more abstract data components in a more compact way. Tensors have a **dimension** (the size of each array element) and a **rank** which describes the number of dimensions used to represent the data [13].

TensorFlow works on the basis of data flow graphs that have nodes and edges [13], making it easier to execute code in a distributed environment. Tensors allow the generation of data flow graphs, graphs of nodes that are executed in **sessions** [13] all computations are represented as data flow graphs and each node represents a mathematical operation whereas each edge represents a tensor [13].

The graph will be executed and processed as more data is fed in during the training process therefore, TensorFlow has two basic working concepts, building the computational graph and executing it.

TensorFlow stores and manipulates data in three possible elements, **constants**, **variables** and **placeholders** [13]. Constants are parameters with values that don't change even during computation, variables allow for new trainable parameters/constants to be added to the graph, all variables must be defined and initialized before executing the graph [13].

TensorFlow has a third data format, placeholders allowing for new data to be fed to the model **outside** the build context, allowing for assignments after the graph is built. Placeholders are commonly used to set the model's input layer, this brings certain advantages when we feed new types of data even

[12]Keras is another standalone library for deep learning yet it's much slower when compiling the model, TensorFlow integrates and optimizes the Keras library.
[13]Central Processing Unit
[14]Graphical Processing Unit

after the graph is built, allowing for models to be retrained and fine-tuned.

As referenced before, data flow graphs are executed in **sessions**, in a session its sole job is to evaluate the nodes, for each new variable assigned, a new operation and hence a node is created and can then be evaluated in the session.

TensorFlow also allows the use of **batches** while training the model. Training can be divided into two types, **Stochastic training** and **Mini-batch** training (or batch training) [10].

Stochastic training performs training on one random input per epoch [10]. Stochastic training can easily avoid local minimums due to the randomness of each input choice but generally would take **longer time** to converge.

Batch training uses batches of training examples instead of one [10]. Batch training executes training considerably faster however, too large of a batch would lead to lower accuracy and would thus require more training steps, at one point using a large batch size would require more computational time to reach the same or even less accuracy compared to stochastic training.

TensorFlow was originally open sourced by Google on 2015, at the end of 2019, TensorFlow's team introduced a new version that would almost entirely change the library, TensorFlow 2.0. Introducing a default **eager execution** environment allowing for TensorFlow code to be executed like normal Python code, operations were now created and evaluated at once [9] as opposed to before where we would have to run a session.

This new version made conversion between NumPy and TensorFlow objects a lot easier no longer needing placeholders [9], completely making sessions obsolete.

Before the new version, TensorFlow had several model building and training APIs, training the model in one of those would lock the code to the used API and reusing the code wouldn't be straight forward [9]. As of version 2.0 Keras is the official and recommended high-level API.

Version 2.0 standardized all the ways of saving a model to an abstraction called *SavedModel* [9], integrating saved versions with the TensorFlow ecosystem which allowed for deployment in many different devices [9].

One of the major changes was the introduction of full mobile support, the deployed model could be embedded in devices like Raspberry or phones using the TF Lite converter and format [9] in these format the model would be optimized and its size would be reduced as well as an improvement to its latency with little degradation in terms of accuracy [9].

TensorFlow 2.0 also introduced a new format, the *TensorFlow.js* format allowing for model loading using JavaScript and executed, even trained in the browser [9].

All these changes introduced by version 2.0 would make the library more versatile and appealing to the general public, during our project we implemented our TensorFlow model in this more maintainable and deployment friendly version.

2.2.3 Flask

Flask is a Python micro-framework with little dependencies. It is designed for web app development but as opposed to other full-stack frameworks such as Django it is does not support functionalities out of the box such as authentication or databases.

Flask is mostly used to build a web server with all the advantages of a Python environment in a relatively small amount of time working by defining routes and its methods. Since its under a Python environment makes the integration of other Python frameworks, libraries and packages such as TensorFlow to be easily added.

This framework was used to implement the project services' being chosen since the service itself does not require a graphical interface[15], that GUI would be the current WeThePlayers project where we would expose the necessary endpoints in our service.

2.2.4 Minio

Minio is an object storage[16] server written in Go [14] implementing the same public API as Amazon S3 [1] which means that code use to set communication between a Minio server/container would be entirely reusable for Amazon S3 integration.
Like in S3, objects are organised in buckets[17].

Minio supports multiple pluggable storage backend including the local disc, Kubernetes persistent volumes and even NAS[18] [14]. In terms of data protection, Minio protects multiple nodes allowing for data replication [14]. Being a Private Cloud Storage it allows for custom access and secret keys becoming one of the best and safest type of storage, having a high performance.

Minio was used in the project development to store all the model data and necessary files during the development environment so that no change would be required during migration to an Amazon S3 storage server in a production environment.

2.2.5 Redis

Redis is an open source in-memory data structure store used as a database cache and message broker [3]. Being written in C Redis is considerably fast with a NOSQL database [3]. Redis also provides support for a wide range of languages, one of them Python.

Redis is a key-based database, storing **text** data in a format similar to a Python dictionary, each value can be a single text value or a list of values.

Redis is mostly used when working with text data and desiring fast and simple operations and functionalities. During the project's development, Redis would be used to store user searches and use them to retrain the machine learning model to account for trend.

[15] If that was the case, Django would be used for a more robust build.
[16] Can be used to store data such as videos, files, docker containers and VM images [1]
[17] Logical separators
[18] Network-attached storage

2.2.6 Docker

Docker is an open platform for developing shipping and running applications enabling for separation of applications from infrastructure allowing for packaging of the application and execution in an isolated environment, **container** [2]. This isolation and security make possible for multiple containers to be executed simultaneously on a given host directly in its kernel [2], containers are always portable lightweight[19] and stackable.

Docker allows a user to better manage the lifecycle of an application through containers, developing the application using containers that then become the unit of distribution and testing for the app after which the app can be deployed in a production environment working anywhere and with no system dependencies, other than the docker engine [2].

The docker engine is a **client-server** application where the server executes and runs the docker daemon process managing all objects like images, containers networks and volumes. The daemon is then encapsulated on a REST API so that the client can issue instructions through its CLI[20] [2] a scheme can be seen in Figure 8.

In terms of architecture, Docker has 4 main components [2]:

- Docker daemon
- Docker client
- Docker registry: this is where Docker stores images [2]. It can be configured to use images from Docker Hub[21]
- Docker objects: this are all objects from images, containers, networks volumes and services

In Docker, **images** are **read-only** templates to build containers [2]. This image can be based on other images, a user can use public images or even private ones through a **Dockerfile** this file is a collection of instruction steps needed to create the image and run it [2]. Each step will set its own layer in the image that way if the file is changed and the image rebuild only that step will change [2].

A container is then a **runnable** instance of an image that can be created, started, stopped, moved or deleted through the Docker CLI.

During the development of a large scale application several containers and images might be used, managing each individually might become cumbersome for the developer, as such Docker provides a tool to solve this problem, **Docker Compose** that can easily handle multiple containers at once.

Application can be configured in Compose through a YAML file and with simple commands all necessary steps to pull and build images, create run and stop containers are made by Compose [16]. To better understand the syntax of Docker Compose YAML file, Docker provides a documented guide.

During the development of Hapibot's project, Docker was used to package each necessary service to test it and simplify the deployment steps in a scalable production environment.

[19] Containers do don't have the guest OS [**docker-1**]
[20] Command line interface
[21] A public registry

2.2.7 Kubernetes

Kubernetes, initially developed by Google to manage containers in a cluster environment is now a portable, extensible, open-source platform for managing containerized applications and services facilitating automation completely managing the lifecycle of applications and services through methods providing predictability, scalability and availability [12].

Kubernetes is built in layers that at each level further abstract the complexity of previous layers [12] aggregating machines (virtual or not) into clusters each with a specific role and with a shared network for communication [12].

In the cluster a server can either be a **master** or a **node**. Kubernetes chooses one server as the master, in case the server goes down it chooses another this will allow a Kubernetes application to be highly available.

A master is the gateway and central logic system in a cluster exposing APIs for clients such as **kubectl**, monitoring servers in the cluster and **load balances** cluster work, as such the master is the **primary point of contact with the cluster** [12].

All other machines in the cluster are classified as nodes responsible for running workloads assigned by the master, each node runs all its applications in containers (these are Docker containers most of the times) improving the isolation, flexibility and management of a node [12].

As mentioned above, Kubernetes uses several layers of abstraction providing scaling, resilience and lifecycle management capabilities to a container [12]. The main Kubernetes objects are [12]:

- **Pods**: Kubernetes' basic unit representing one or more containers that should be viewed and controlled as a single application. Pods can be configured manually however, Kubernetes encourages configuration of higher level objects to implement additional functionality, these higher level objects will them run on pods generated by the Kubernetes engine [12];
- **Replicas**: To add scalability Kubernetes can replicate pods that can be horizontally scaled[22] by replication controllers and replication sets [12];
- **Deployments**: the most common object using replication controllers as a building block, designed to ease the lifecycle management of replicated pods[23];
- **Jobs**: introduces a different setting from deployments, where deployments assumed a long-running service jobs provide a task-based workflow where the application is **expected to exit successfully after some time** [12]. This component also allows for **cron jobs** that are jobs with a scheduling component;
- **Services**: **internal load balancer and ambassador for pods** [12] grouping together pods as a single entity allowing users to only consider a single **stable endpoint**[24]. Services can expose endpoints for cluster only services or publicly;
- **Volumes**: an abstraction layer allowing data to be shared between containers within a pod without complex mechanisms and for that data to be persisted beyond the lifetime of the containers [12]. Since the volume is destroyed once the pod is terminated/fails it is not a fault tolerant data persistence solution;

[22] Adding more machines to the pool of resources where one can be the main container and other containers can be used as helper containers to execute tasks

[23] This can be tasks like history tracking, failure recovery and application updates [12].

[24] Remains stable despite changes to its routing pods [12]

- **Persistent volumes**: storage that is not dependant on a pod's lifecycle allowing for cluster administrators to configure storage request claims;

All the layers in a cluster can be configured through a YAML configuration file, snippets 1 and 2 present examples fo configuration files.

Kubernetes was used to scale the final application for the user prediction service.

2.2.8 GraphQL

GraphQL is a query language for APIs open sourced by Facebook in 2015 that effectively substitutes REST APIs where once we would need several endpoints to perform a complex task, with GraphQL we only need to set a **single endpoint** and set data structures that then can be queried and specified by users simplifying the returned data.

With GraphQL data types can be defined and returned in a query adding a massive amount of versatility to an API. Even tough it was initially designed to work with React, GraphQL now supports several platforms and frameworks including Django, Flask and Angular.

GraphQL however, is not recommended for simple APIs as it is rather complex, WeThePlayers supports GraphQL and we will then perform requests to its API to fetch game data due to the API's versatility we can specify the search fetching data based on a timestamp that way we can check whether or not new games have been added since the last timestamp.

2.3 Considered technologies

These are the technologies that were considered during development but were later dropped.

2.3.1 Elasticsearch

Elasticsearch is an open-source, broadly-distributable, readily-scalable, enterprise-grade search engine on top of Lucene Standard Analyser for indexing and type guessing. Accessible through an extensive and elaborate API, Elasticsearch can power extremely fast searches that support your data discovery applications.

Being a full-text search and analytics engine, allows the storage, search and analysis of big volumes of data quickly and near real time [11]. It's quick to set up out of the box, having a short learning curve standing as a NOSQL database.

The backend of Elasticsearch is comprised of five main components [11]:

- **Node**: single server being part of a cluster, storing data and participating in the cluster's indexing and search functionality [11]
- **Cluster**: collection of one or more nodes, providing federated indexing and search functions [11]. Since Elasticsearch operates in a **distributed environment** with cross-cluster replication it can generate a secondary cluster as a hot backup [11].

- **Index**: collection of documents with similarities
- **Document**: base unit of information, documents can be indexed

Elasticsearch can also be used for search logging providing two components, *Kibana* and *Logstash* [11]. Kibana allows for data visualization from charts, histograms and Geo maps to custom building of a visualization tool, enabling advanced time series analysis [11].

During our project we planed to use Elasticsearch's Kibana to store all user searches and then extract them to serve as our trend data however, during development we implemented that system in a less intruding format using our prediction system service supported with a Redis database.

2.3.2 Kafka

Kafka is a fast, scalable, durable and fault-tolerant messaging system [4], most often used in real-time streaming of data most often to stream big data for analysis in platforms like Cassandra [4]. It is easy to setup and has excellent performance [4].

Kafka is able to batch data records into chunks that can then be seen end-to-end between the record producer and the consumer [4]. Using batches make data compression more efficient reducing I/O operations latency. Kafka works using clusters and therefore is fault-tolerant, if the master is down a new cluster takes place.

In terms of communication, Kafka communicates between clients and servers using a versioned and documented wire protocol[25] over TCP[26] [4].

In Hapibot's WeThePlayers project their main database is an SQL database however, their search engine uses the Elasticsearch service which uses a NOSQL database, Kafka is used to send data to the Elasticsearch's database, when a new entry in the project's database is made that same entry will be streamed to Elasticsearch and saved to its database having synchronized databases. During the first stages of the internship, Kafka would be used to train and deploy the machine learning model, this will be explored in the next section.

2.3.3 Kubeflow

Kubeflow Google open-source project that simplifies the deployment portability and scalability of machine learning workflows on Kubernetes [15]. Deploying production-ready machine learning services involve several components that aren't related to Data Science and that pose challenges to the DevOps team if they don't possess any knowledge about the model.

Kubflow addresses these challenges, improving collaboration in machine leaning workflows. Introducing pipelines supporting model generation in a Jupyter notebook a data scientist can develop deploy and manage ML applications with little to no changes during transition from prototyping to production on top of natively supporting Kubernetes.

[25] A way of getting data from point to point, needed if more than one application has to interoperate [5]
[26] Transmission Control Protocol

The most recent versions of Kubeflow support distributed TensorFlow training through a TF Job [15], serving a trained model using TF Serving and evaluating a model in real time with TensorBoard [15].

To implement a machine learning workflow in Kubeflow, the application needs to be developed in a GCP[27] project[28].

Kubeflow was the initial approach to a Kubernetes deployment of the search prediction project.

3 Developed work

At the start of each week we would review the work completed in the previous week as well as plan the work for the current week, Hapibot's task managing tool is Asana as such week planning would be set on sprints and tasks marked with its week's sprint.

In the next subsections we will describe the performed work during the development of the project with each subsection representing a sprint.

All sprints and task information can be accessed in the appendix A.3 or A.4 respectively. Figure 22 displays all the tasks developed by the author.

3.1 Sprint 1

This was the first sprint of the project, set at the 20th of February Figure 12 displays the sprint. All work done during this week was related to research being fundamental as it set direction of the project in terms of model architecture and frameworks that would be used in the first versions.

After testing and creating benchmarks against competitors' search systems, Figure 24, and researching types of neural networks architectures that could be applied to this specific NLP problem, Figure 23, we ended up choosing RNN as the neural net to implement, more precisely the LSTM architecture to take into account the vanishing gradient problem, the research document can be found here.

Having chosen the neural network architecture to implement and its implementing framework (TensorFlow) the author started researching TensorFlow's integration with Kafka as a first deployment method, Figure 25, that can be seen here.

Once the model integration research was completed a proof of concept implementation was made using TensorFlow and an open-source Python library textgenrnn by Max Woolf, Figure 26 shows that task.

[27] Google Cloud Platform

[28] Note that this feature cannot be used in a free and trial period subscription.

3.1.1 Tensorflow Kafka inegration

With the new version of TensorFlow and a support package it was found that TensorFlow now supports Kafka integration in two approaches [8]:

- Stream processing with model servers
- Stream processing with Kafka embedded models

In the model server integration, a model would have already been trained and it would be deployed in a TF Serving server, Kafka would then communicate and send request data through gRPC to the model server that would respond with a prediction and Kafka would then stream the prediction, Figure 9 shows the overall design [8].

In the stream processing with a Kafka embbeded model there would be no dependency on an external server as Figure 10 shows, Kafka would be responsible for building, training and serving the model through a Java API [8].

3.2 Sprint 2

On the second week, at the 2nd of March a review was made from all the tasks completed in the previous week's sprint, after testing the proof of concept model some limitations were found, the model could predict the next word or the next char but not both. Figure 13 shows this week's sprint.

The main goal for this week was to develop a solution (another model) to solve that limitation and test its viability as well as testing the integration of continual training with trends.

Even though the char model had been built, after some testing it was noticed that predictions would never be as precise as the word predictions since the sequences used to train would not provide enough context when compared to the sequences used for words.

After working with the textgenrnn proof of concept model the author started development of a NLP LSTM model from scratch, Figure 27 shows that task, in this development it was first required to preprocess the data to clean irrelevant data as well as converting the text data to a more friendly format, numerical as neural networks only work in that format. Listing 3 provides an example of the process the following subsections look into each step during preprocessing.

Figure 40 represents the overall structure for functionality developed during this sprint. All the work developed managed to complete the MVP[29] for the model development which would mean that in the next sprints we wouldn't have to worry much with the model as it would mostly be a backend and DevOps job.

After generating the model we would test it and tweek it until it performed as expected, predicting the next word in a sequence[30], we started adding the functionality to train it again but this time on trend/user search data observing that the model would quickly adapt to the new data and would start to incline based on trends[31]

[29] Minimum viable product

[30] Since it classified a word based on its significance in the entire database as opposed to the scoring method in Elasticsearch we noticed that it would produce results more appealing in terms of popularity when compared to the search system.

[31] The initial build took a long time because it was generating all the weights and the structure, once the model was

3.2.1 Data translation

Since a neural net only works with numerical vectors we would have to translate the text data, to do so we developed some **tokenizers**[32] that would build a map where **each key would represent a word with an unique id value**.

Once the tokenizer was built we would now need to generate a numerical vector for this reason we converted sequences of words to sequences of integers where each element in the sequence/array would be a word id.

3.2.2 Data segmentation and padding

For NLP problems it is most common to consider continuous fixed size sequences, for instance, all phrases in a book considered as continuous sequences of one or more sentences however, for the prediction system since it was designed to predict game entries in the database it would have to consider each entry/line in the DB as a single data point and not part of a collection therefore we would segment the data in an ngram. At the end we would have a mapping of sequences to sequences.

Since the model had an input and output embedding layer where one would feed it an encoded sequence another decode its output to all possible mapped words, each had to be fixed length therefore we had to ensure that **all sequences had the same length** through a process called **padding**, we reserved the id 0 to represent the padded value and the model would then ignore all the zeros allowing for different size sequences to be used for training.

3.3 Sprint 3

This sprint, Figure 14, made at the 23rd of March reviewed the work of more or less two weeks. During that time and after the completion of model generation we started the deployment on a testing environment, locally. Looking at TensorFlow's documentation they recommended building their Docker image however, we ran into problems with Bazel where it would conflict with several TF versions as such we decided to pull their base image and customize it to our own needs.

We would serve the model using a container and would then perform REST or gRPC requests to it requireding for the model to be exported from a h5 format to a protocol buffer (pb) format building the Python network communication scripts afterwards. Figure 41 represents a diagram with the functionality implemented at this point in the sprint.

After some testing we stopped using the Docker container and moved to a more stable and gRPC only server package, the **tensorflow_model_server**.

Once the local testing was completed and performed as expected, the next challenge was **scalability** since this was to be deployed in a cluster environment with thousands of possible requests per hour the model server had to be scalable. The second challenge was the fact that the **model could not be**

built training on new data was faster since the dataset would be smaller and the model would only increase or decrease the weights' significance.

[32] A translator between the two formats, text and numerical that was able to encode and decode data into each format.

static, it had to keep updating as the game database expanded and had to update based on trends.

As such, Kubernetes was the next obvious integration in the project where the data processing and model training would be in their own single pods and would then update the model in a common bucket with the scalable replicated pods of the user server[33]. The prepocess and train pods would be cron jobs repeated daily or weekly depending on the need.

Figure 43 shows the initial general design for the Kubernetes implementation where Figure 44 represents the data preprocess container, Figure 45 represents the training container and Figure 46 represents the scalable user prediction server.

Before testing Kubernetes deployment in a cloud service we tested the scalability in a single local cluster using Minikube[34], Figure 42 shows the plan for could serving, locally.

Figure 28 shows the task addressing this sprint's works being fundamental in the final structure as it promoted a script restructure to be more modular and easily configurable through Kubernetes config files using environment variables.

3.4 Sprint 4

This sprint, Figure 15, was set at the same time as the previous one, at the 23rd of March however, its task, Figure 29, was delayed since its work was more related to research and could easily be accommodated once the model was deployed on Kubernetes.

During this sprint and task, we would research ways of expanding the model outputs, the model would have an embedding layer with size/number of cells equal to the **number of unique words in the database**, the problem came from the fact that the database is updated daily, with new possible words being added at each update, as such it would be advantageous to increase the number of output classes dynamically as needed.

However, since with each update new sequences might be added that were bigger/had more words than the previous version we would also have to extend the input layer being far easier to just build a new model from scratch therefore **this sprint's research would not be applied** for now[35].

Although this research was not used it was important not only to improve the model but also to start evaluating model performance more than before, this would later cause us to integrate batch training in the model development.

[33] This server's job would be to encode user prediction requests and decode and return the model prediction for that request.

[34] As referenced before Minikube allows a user to host a cluster with a single node locally this way a user can test if an application's Kubernetes configuration and structure works, if so moving to a cloud system doesn't introduce changes.

[35] In a future section we tackle this problem again and improve its implementation.

3.5 Sprint 5

During this task, Figure 16, set at the 6th of April, we looked back at the work completed during sprint 3. From this planning two main objectives were set, integrating the model development in Kubeflow, a TensorFlow's toolkit for Kubernetes, and generating the project API.

Figure 30 shows the task related to the Kubeflow integration, as referenced in the previous section, Kubeflow provides an environment that allows for an easier deployment of models in a Kubernetes environment as such we had great interest in integrating it in our workflow, building specific pipelines for data processing and model training. Having native support for Jupyter Hub and TensorBoard, Kubeflow would be ideal to continuously test the model's architecture without having to redeploy the entire server.

However, several issues were encountered during its integration, even though Kubeflow could be added in a limited fashion to a local cluster system, such as Minikube, our available resources (16GB of RAM) proved to be insufficient and had to move to a GCP environment. Despite being open source and providing instalation tools, Kubeflow required a paid GCP account which does not include the trial membership thus and since one of the objectives during development was to keep the project as modular and adaptable/platform independent as possible, **integration of Kubeflow would not be added** and we would have to change direction once again looking for ways of implementing our own pipelines.

The API would be structured to provide an endpoint to perform direct predictions and later, admin access endpoints for server configuration and monitoring, this way the Hapibot's search prediction project could be applied to other future projects and not just to the current WTP product integration.

Despite the fact the this sprint's main goal, the integration of Kubeflow, was unsuccessful it proved to be a milestone during the project allowing us to evaluate our project both in how modular and platform independent it had to be as well as how cost efficient it needed to be to become a viable product.

3.6 Sprint 6

This sprint, Figure 17, set at the 20th of April introduced major changes, after the unsuccessful integration of Kubeflow we shifted the structure designing our own data pipelines which effectively substituted Kubeflow. We would still follow a design close to Figure 43 separating each feature into its own service/container.

During this sprint we would once again tackle the model extension researched during sprint 4, Figure 33 shows this task. In order not to overload the training server resources we started looking and ended up implementing a **transfer learning** step in our train scripts where in a process similar to surgery we removed both the model's input and output neural layers adding new ones that accommodated for possible new words and bigger sequence sizes performing some training steps over the entire dataset afterwards so as to connect[36] the new layers to the actual LSTM layer that would remain unchanged.

Transfer learning provided very good results in terms of training time since the number of attributes required to train would be far less when compared to normal training yet the accuracy would drop to a point where we would have to train for more epochs additionally the model would quickly overfit rendering the whole process useless causing us to look in a new direction, **batch training**, aggregating random train data and training it as an entire batch as opposed to a single sequence at a time. This

[36] Change weight values according to the prediction error values calculated during backpropagation.

would greatly reduce the training time even when compared to the transfer learning, at one point we managed to reduce the time from 15+ minutes per epoch to a matter of seconds per epoch[37].

Batch training proved to be very useful but would come with new concerns, at some point if too big of a batch was used it would cause the model to converge faster and overfit, another problem came from the fact that as the batch size increases so does the number of epochs required to achieve the same accuracy[38].

During this sprint several problems and milestones were tackled and several creative solutions were found to keep our implementation as fast and simple as possible without affecting functionality.

3.6.1 User prediction server container

This was the actual server for user predictions and would handle all user requests, encoding them, performing inference and then decoding the prediction into a text format. At this point to keep things simple we would use a flask server to handle the user requests, in the flask server for each user request we would perform the prediction requests to a running tensorflow_model_server server with our trained and exported model.

This workflow was advantageous since we wanted to keep updating the model as new data came in if we conferred to all containers/services a **shared persistent volume** we could just update the model there and the model server would load it and serve it without any issues or delay.

Figure 31 displays the task addressing this implementation and Figure 49 represents the diagram for this container.

3.6.2 Data fetch and model training

Since we wanted to have an updated model and register searches we required a service that could handle all of that, a service that was able to read the game database entries and user searches[39] and would then preprocess and tokenize them, storing the necessary tokenizers in a common volume. Figure 47 represents the design following these requirements.

At first we started developing a Luigi server to fetch the data, Luigi would dynamically fetch files as needed while also being scalable. Not long after, Luigi was dropped mostly because we would only have to read a single file whereas Luigi was an excellent tool for multiple files and therefore would be overly complicated to implement for a simple feature. Our service would now also perform data fetch requests to the WTP's database through its GraphQL API.

A second container was also needed to train the model, the initial design can be seen in Figure 48. The training would be separated depending on its type be it a new train from the database or a trend train with user searches, the training would then run in a Kubernetes TF Job that would restart anytime an error during training occurred. Once the model was trained it would be stored in the common volume updating the server model in the user prediction service.

[37] This values were observed in a GPU running in a Colab notebook, on a normal GPU performing training without batches could take more than 30 min per epoch. After some tests with the GPU we found that the model reached reasonable values at around 50 epochs

[38] Each epoch would train faster but if the batch reached very large numbers it would take almost as much or even more time that training with no batches.

[39] These searches being stored in Elasticsearch with the help of Kibana.

Although this two services were initially planned as separate services this design was redundant moving to a better scheme where we fetched the database data periodically and if new games were found we would build a new model otherwise we would train the existing model on trend data, if it was found, this trend training also applied to new models. This feature, checking for new games, was easily integrated compared to other features since the WTP's API was in a GraphQL API we could store the last request timestamp and query the database from that point checking whether or not new data was present/returned. Figure 32 represents this service task.

3.7 Sprint 7

In this sprint, Figure 18, set at the 4th of May the MVP was completed, the data fetch and train service as well as the user prediction service were working and the training steps now incorporated batch training and some errors were fixed Figure 35.

Once again, we tried to maintain and add more functionality to the product while keeping it platform independent, we further increased its API and documentation, **displayed predictions probabilities** so that the product buyer could choose whether or not to consider the model predictions, we also saved searches in a Redis database meaning that the product would not introduce major changes in an application (it only needed to be added a new route), added a live training feature so that users can experience real time trends and added a general shared storage that could be easily integrated into an already existing storage.

3.7.1 Prediction probabilities

The user prediction service would be able to predict the next word, the sequence of x next words or event display the next or nth next prediction with more than one possible prediction sorted from most probable to least probable. However, we noticed that if a search was made with errors (typos, gibberish and so on) or if it reached the end of the context sentence the model would still return **prediction with close to zero certainty** as such we decided to add a functionality to **display each prediction probability** that way the application using the system could consider a probability threshold to filter viable predictions.

Figure 34 shows this functionality task and Figure 11 provides an example of this feature. Taking a closer look at that example we can see that initially searching "god" the model predicts "of" with 35.5% of certainty however, once it considers "god of" for the next prediction that new prediction of "war" has a 78.9% of certainty meaning that, **as the number of words provided increase the more certain the model will be when prediction the next word** until the end of that sequence's context is reached, the end of the game entry.

The returned JSON data from the prediction request (Figure 11) has four main sections, the status of the request (if it was successful or not) under *success*, the text sequence of predictions under *predictions* notice that as the number in **next** increases in the request the number of words displayed in *predictions* also increases. One other section is the *probabilities* this field returns an ordered list where each element is the inference probability for **each word in the *predictions*** field. The last field, *possible*, returns a list of predictions, the number of elements is set by the *multiple* field in the request if not set defaults to one, in each element in the *possible* list they have associated their actual prediction value and the

prediction probability.

3.7.2 Saving user searches and live training

Initially we planned to have a coworker implement the functionality to store user searches in Kibana and we would then perform requests to read it however, it required the integration of further changes in the WTP' backend thus we moved to a less intrusive method where we would expose an endpoint in our prediction service that would be used in the WTPs' frontend to send the user actual search, once he selected the desired game. The search would then be saved and accessed in a Redis database, further endpoints would be made to monitor and clear the trend data.

One of the main reasons to do this implementation was to add **live training** to the deployed model where with each trend request not only would we save the search in the database but also retrain our model on that search this way we didn't need to wait for the model in the data fetch service to finish training and update trends, now they would be immediate. This new implementation would mean that we no longer used the tensorflow_model_server server to host our model now being loaded into the service's memory. Figure 36 shows this task.

Using this new method to save trends would also mean that the trend training made in the data fetch service would also change accommodating for the Redis database, the task describing that change can be see in Figure 37.

3.7.3 Shared storage

One of the other tasks developed was the task in Figure 38, since we needed to train new models and deploy them to the user prediction service the best approach would be to use a common persistent volume/bucket. In the final product the bucket would be a bucket with an AWS S3 bucket structuretherefor to test and implement the shared storage locally we would use Minio with the Python boto3 package this way **no changes would be required** to move from a testing environment to a production one. Once the new model in the fetch service was completed it would save it to the bucket and notify the user service.

3.8 Sprint 8

The work for this sprint, Figure 19, set at the 18th of May was mostly testing the changes introduced in the previous sprint. Testing the trend caption and live model training, the communication with the Redis database and shared Minio storage in both data fetch and user prediction service.

It ended up being an important step to ensure that everything worked, that the planned scheme was actually viable. If all tests were successful, which they were, we would be closer to this project's completion.

3.9 Sprint 9

When this sprint, Figure 20, was set at the 26th of May all the services were operational and the project was divided into two services the **data fetch and model training service** shown in Figure 50 and the **user prediction and trend caption service** shown in Figure 51.

Since all the services were up and running the next plans were to containerize and test each service and then, in an effort to manage the containers more easily we would build a Docker Compose file. Listing 4 shows an example for this file. Once the Compose file was made some deployments errors related to Compose were encountered and solved in pair programming sessions with the internship's supervisor, Figure 39 shows this task.

3.9.1 Data fetch and model training service

This service would periodically check the WTP's database for new games if new games were found the service's game data would be updated, tokenizers built and a new model generated and trained.

After that even if no new game data was found the service would retrieve all saved trends from the Redis database if trend data was found it would train the model be it a new one or not on that trend data after that it would backup in the bucket all generated data and would then deploy the model and tokenizers to the serving bucket, notifying the user prediction service of the existence of a new model.

This flask service also possessed more endpoints to force training, retraining and deploying used for testing as well as running status monitoring and access to each steps' logs[40] all of them requiring admin access.

3.9.2 User prediction and trend caption service

The main feature for this service would be the model inference through the */predict* endpoint where it would encode a user request performing model inference and decoding its response returning it to the user. This service loaded the tokenizers and model retrieved from the buckect to its memory.

The service would also provide further endpoints related to trends, from clearing them, listing them and training on them. The main endpoint is */tend* where it would take the user search of a list of searches and would perform **live model retraining** accounting for trending searches and would then save the request trends in the Redis database to be used by the data fetch service.

The model also provided a Cluster IP endpoint where the data fetch service would notify this service that a new model was added/updated in the serving bucket at which point this service would reload all its data (tokenizers and model).

One thing to note is that although the service works with a memory loaded model it can be configured (through an environment variable) to also work with the first implementation version, using the tensorflow_model_server package.

For the final product, only the */predict* and */trend* routes would have to be added to the frontend where the first returned data that had to be displayed (the trend data). This allowed for a minimal

[40] Fetching data, preprocessing it, building and retraining models would all generate their own log files that could be accessed through the REST API.

intrusion in an already running service.

3.10 Sprint 10

During this sprint, Figure 21, set at the 2nd of June the Compose deployment was completed and working, we monitored the memory use for each service to access the server requirements.

At this point during development all that was left to do was the integration and testing of this finalized project in the WTP's project staging environment before being added as a permanent feature by the DevOps team as such all the objectives fro this internship project were completed.

To see the **project's API** for the first release you can view this image.

4 Conclusion

The initial objective for this project was to build a search prediction system, at the start of the project the overall design wasn't very specific as we didn't yet have an idea of how best to integrate the final product. Meanwhile, during development, several technologies were used and tested to adapt the project to the current needs. This iterative development approach meant we were able to build progressively on the top of a previous build, all with the purpose of building a functional, scalable and adaptable product.

In terms of work, this project was highly ambitious for the amount of time and for a single developer which in my opinion proved to be a very interesting challenge and an incredible learning experience working with several recent useful technologies and schemes, ending up achieving and even surpassing expectations.

As can be seen by Figure 4, Figure 5, Figure 6 and Figure 7 this project integration provided a second dimension with the search experience **without interfering with the already established search system** this way we were able to give users a better experience where we could speed up their search without reducing the amount of information they have access to in the preview not only that but we were also able to satisfy their need in terms of desired search, with this implementation the problem stated during the introduction of this project was no longer present.

Since we take into consideration the user searches to calculate trends, one prediction might not be present the first time a user searches for it however, as more users search for it the higher its weight will be in the neural net and thus it's prediction probability will increase, one such example we experiences was the search for the game *horizon zero dawn*, at first the search would not display its prediction but after a couple searches this game was immediately predicted when searching "horiz" this way we can conclude that the search will **evolve in a direction set by the user** without any influence from the website leaving users feeling more connected with the service.

Even though this projects' presence in WTP's website seams minimal when compared to the amount of work required that is most often the case when building user experience improvement features.

As for the company environment, everyone was always eager to provide some help if I ever asked despite having their own projects and problems to attend to. We always had weekly meetings even during the pandemic (through video calls) where we would all discuss the project, from the overall structure to its integration. These meetings and their general support became very important to me, feeling that all my opinions and ideas were never dismissed and were always considered, being given the liberty to test them and actually decide the direction of the project. I believe that was extremely important, having that kind of autonomy and at the same time responsibility, meant that I had to be more critical of my work and consider every action in the long run as well as stand by my actions which motivated me to keep improving the project.

Less pleasant experiences, if any, were mostly related with certain technologies that I later dropped such as Kubleflow, either because they were technologies that took time to learn and then ended up being inadequate for this project or because using them proved to be quite frustrating with bad documentation or system requirements that were understandably unavailable in an internship environment[41].

Considering all my work and all I've learned during this internship, as well as the level of difficulty in the project and the structure of this report I believe it should be seen and evaluated as an excellent internship project work. I once again would like to thank Hapibot Studio for always being supportive and providing me with an amazing learning experience.

Note: This prediction system is not yet available in the live WeThePlayers website - it is currently deployed in the staging environment undergoing further tests and improvements for a future launch.

[41] This mostly had to do with Google's Kubeflow and its docs.

Bibliografy

[1] Felix Bartels. ***Minio – The secure alternative to Amazon S3 for object storage***. mar 20, 2018. URL: https://www.univention.com/blog-en/2018/03/minio-the-secure-alternative-to-amazon-s3-for-object-storage/. (last accessed on june 5, 2020).

[2] ***Docker overview***. URL: https://docs.docker.com/get-started/overview/. (last accessed on june 5, 2020).

[3] Prateek Gogia. ***Redis: What and Why?*** feb 14, 2018. URL: https://codeburst.io/redis-what-and-why-d52b6829813. (last accessed on june 5, 2020).

[4] Jean-Paul Azar. ***What Is Kafka? Everything You Need to Know***. aug 9, 2017. URL: https://dzone.com/articles/what-is-kafka. (last accessed on june 5, 2020).

[5] Wikipedia. ***Wire protocol***. sep 18, 2017. URL: https://en.wikipedia.org/wiki/Wire_protocol. (last accessed on june 5, 2020).

[6] Eduardo Morgado. ***Implementação de Rede Neuronal***. may 14, 2019. URL: https://github.com/thejoblessducks/A-Neural-Network-That-Could/blob/master/Relat%C3%B3rio%20IA.pdf. (last accessed on june 5, 2020).

[7] Eduardo Morgado. ***NLP: Recommendation and Inference using Tensorflow***. june 11, 2019. URL: https://drive.google.com/file/d/10Zs0vm2OqJi7pEqitmy03lTIickLV_Vh/view?usp=sharing. (last accessed on june 13, 2020).

[8] Eduardo Morgado. ***TensorFlow and Kafka Integration for Data Processing***. june 11, 2019. URL: https://drive.google.com/file/d/1gcEaX2buOlgbJ_ovpoTrMt_3UMqxkHsN/view?usp=sharing. (last accessed on june 13, 2020).

[9] Thalles Silva. ***Everything you need to know about TensorFlow 2.0***. jun 26, 2019. URL: https://hackernoon.com/everything-you-need-to-know-about-tensorflow-2-0-b0856960c074. (last accessed on june 5, 2020).

[10] Rising Odegua. ***Stochastic vs Mini-batch training in Machine learning using Tensorflow and python***. jul 17, 2018. URL: https://medium.com/coinmonks/stochastic-vs-mini-batch-training-in-machine-learning-using-tensorflow-and-python-7f9709143ee2. (last accessed on june 5, 2020).

[11] Giovanni Pagano Dritto. ***An Overview on Elasticsearch and its usage***. mar 27, 2019. URL: https://towardsdatascience.com/an-overview-on-elasticsearch-and-its-usage-e26df1d1d24a. (last accessed on june 5, 2020).

[12] Justin Ellingwood. ***An Introduction to Kubernetes***. may 2, 2018. URL: https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes. (last accessed on june 5, 2020).

[13] Simplilearn. ***What is Tensorflow: Deep Learning Libraries and Program Elements Explained***. april 28, 2020. URL: https://www.simplilearn.com/tutorials/deep-learning-tutorial/what-is-tensorflow. (last accessed on june 5, 2020).

[14] Insights. ***Minio Distributed Object Storage Architecture and Performance***. oct 26, 2018. URL: https://www.xenonstack.com/insights/minio/. (last accessed on june 5, 2020).

[15] ***Kubeflow***. april 4, 2020. URL: https://www.kubeflow.org/docs/about/kubeflow/. (last accessed on june 5, 2020).

[16] ***Overview of Docker Compose***. URL: https://docs.docker.com/compose/. (last accessed on june 5, 2020).

A Figures

A.1 WTP searches without the prediction system

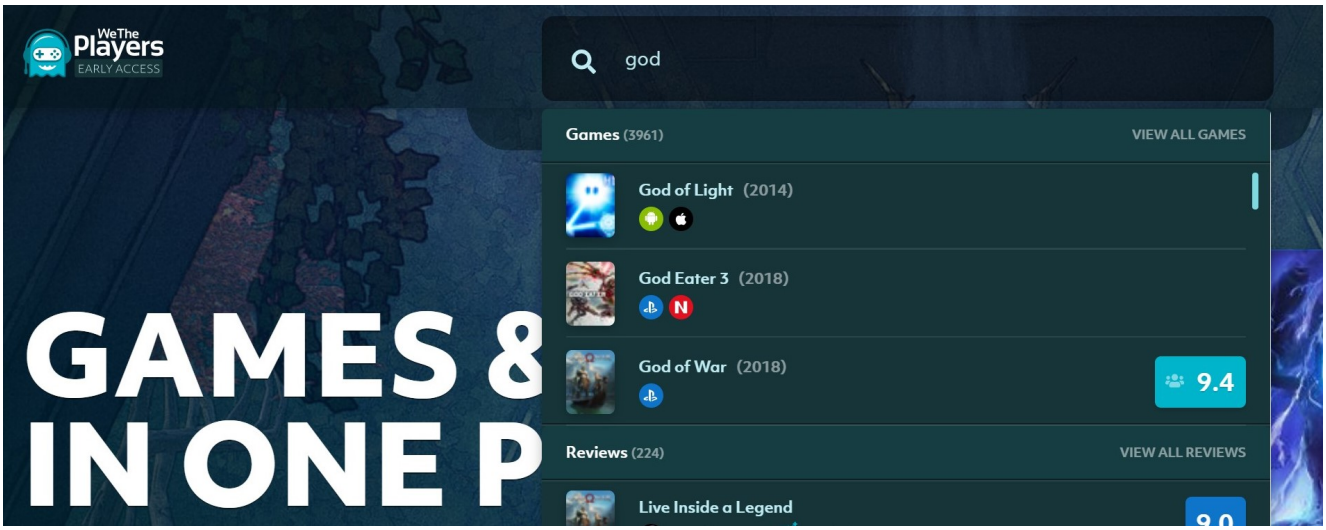


Figure 1: WTP game search "god" **without** the prediction system

Figure 2: WTP game search "the legend" **without** the prediction system

Figure 3: WTP game search "the legend of" **without** the prediction system

A.2 WTP searches with the prediction system

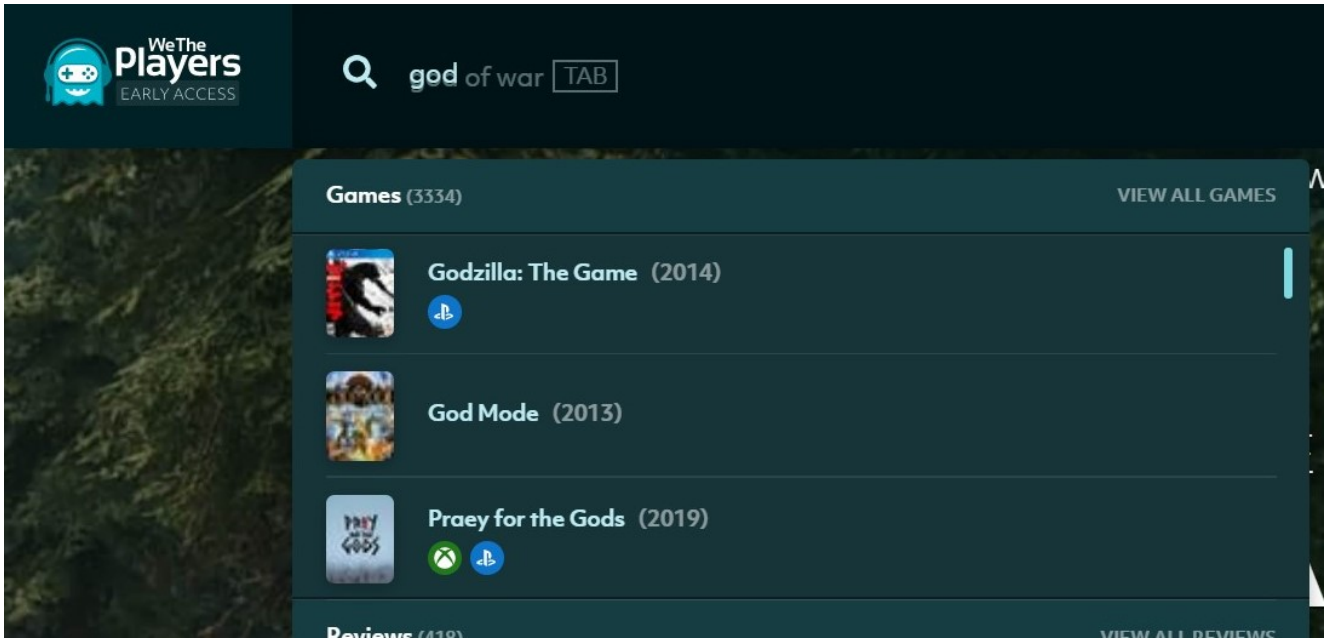


Figure 4: WTP game search "god" **with** the prediction system

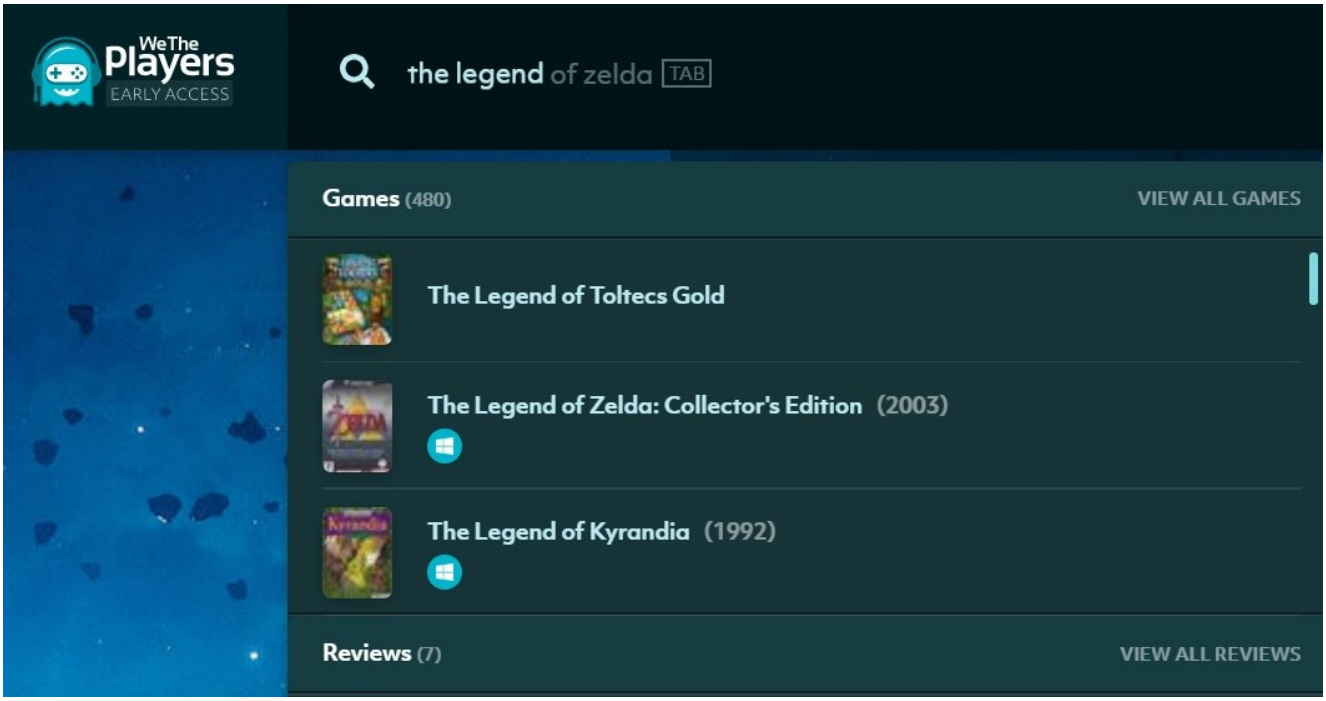


Figure 5: WTP game search "the legend" **with** the prediction system

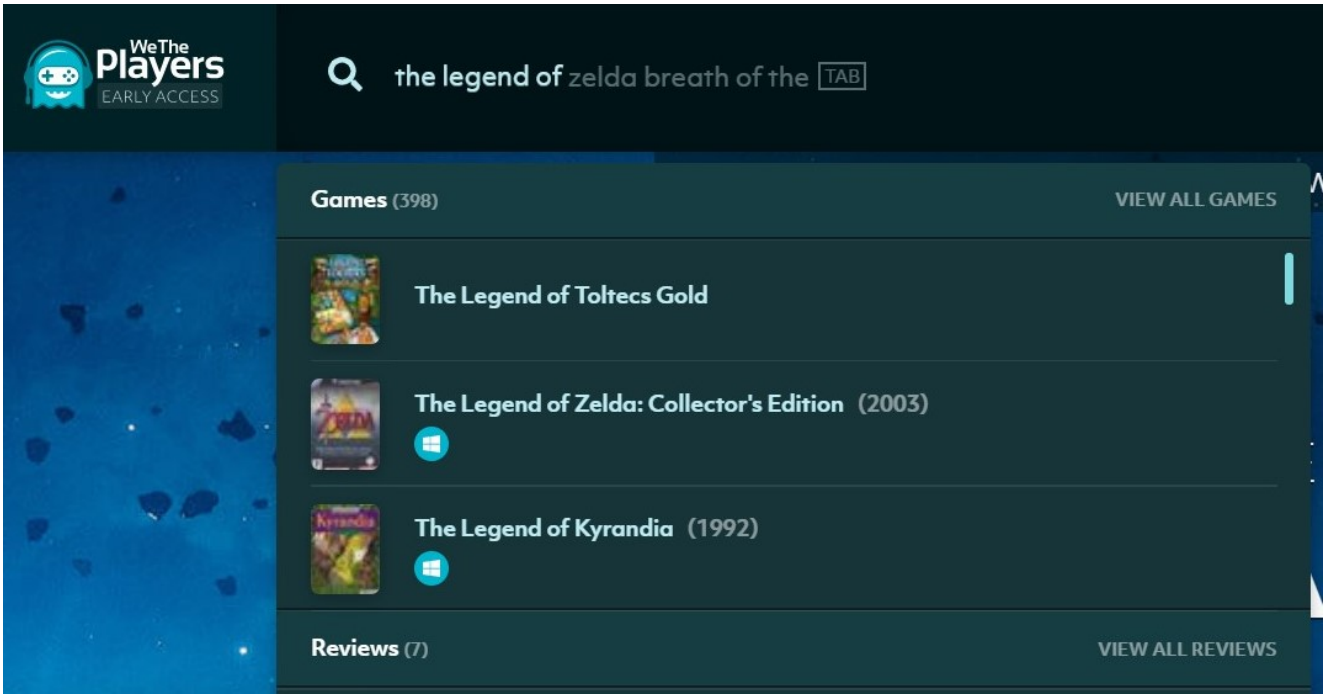


Figure 6: WTP game search "the legend of" **with** the prediction system

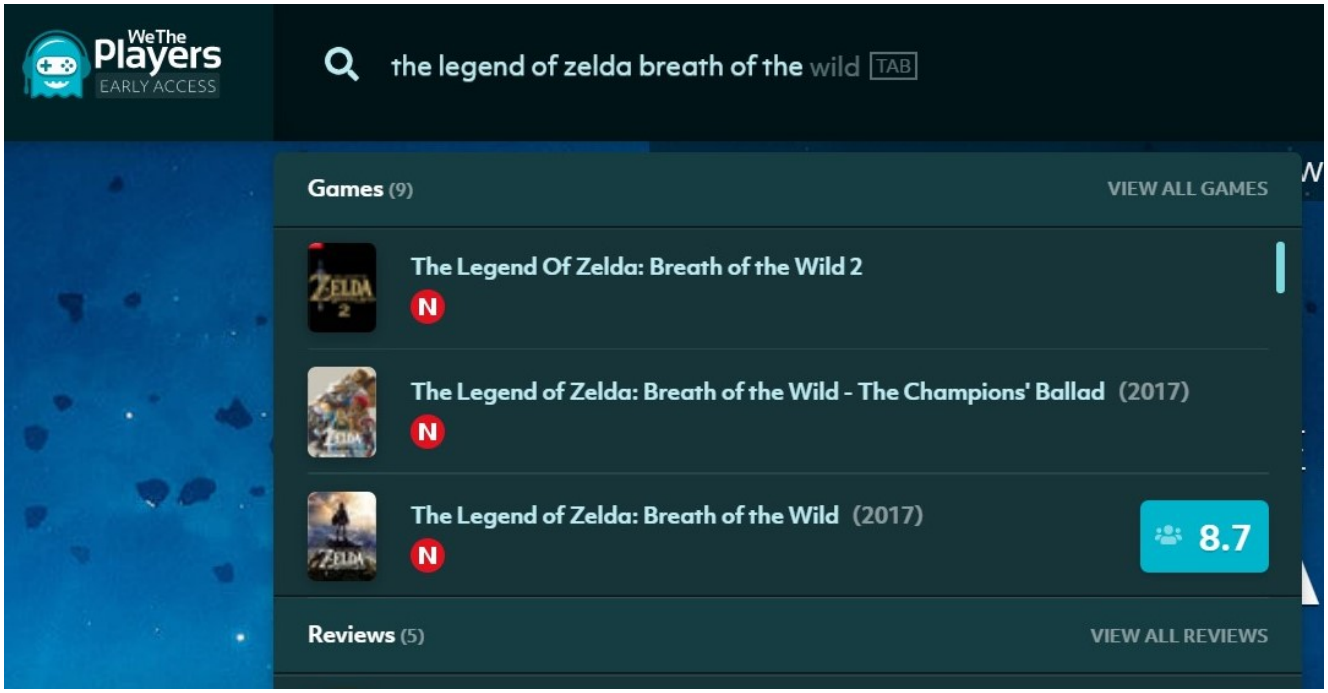


Figure 7: WTP game search "the legend of zelda breath of the" **with** the prediction system

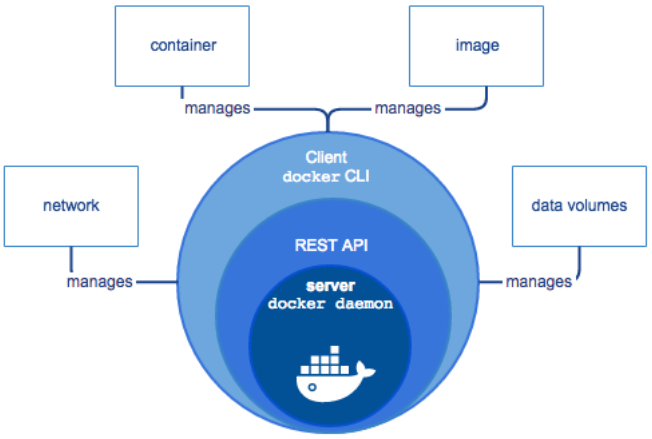


Figure 8: Docker engine structure [2]

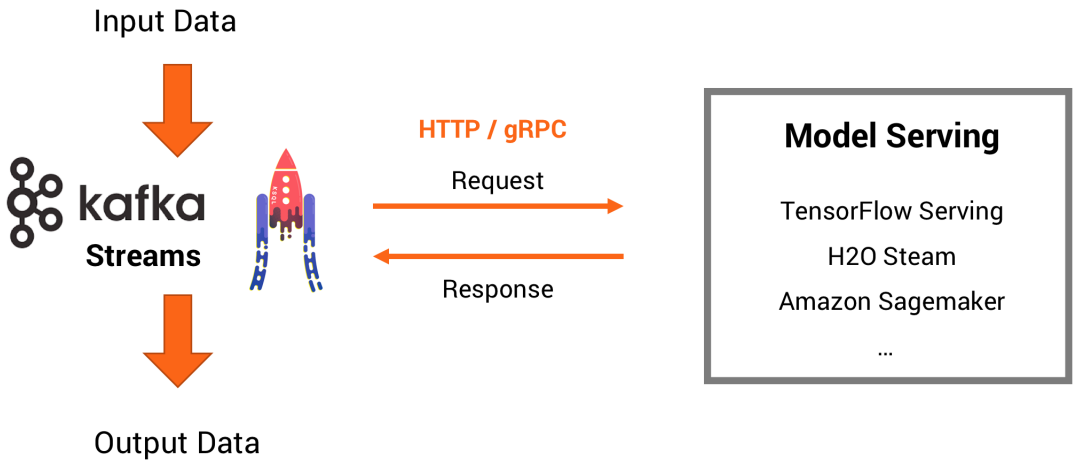


Figure 9: Kafka stream processing with model servers

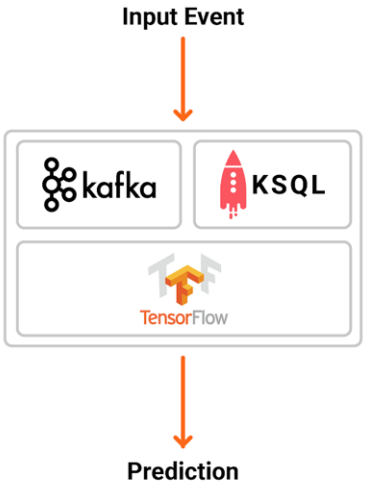


Figure 10: Kafka stream processing with Kafka embedded models

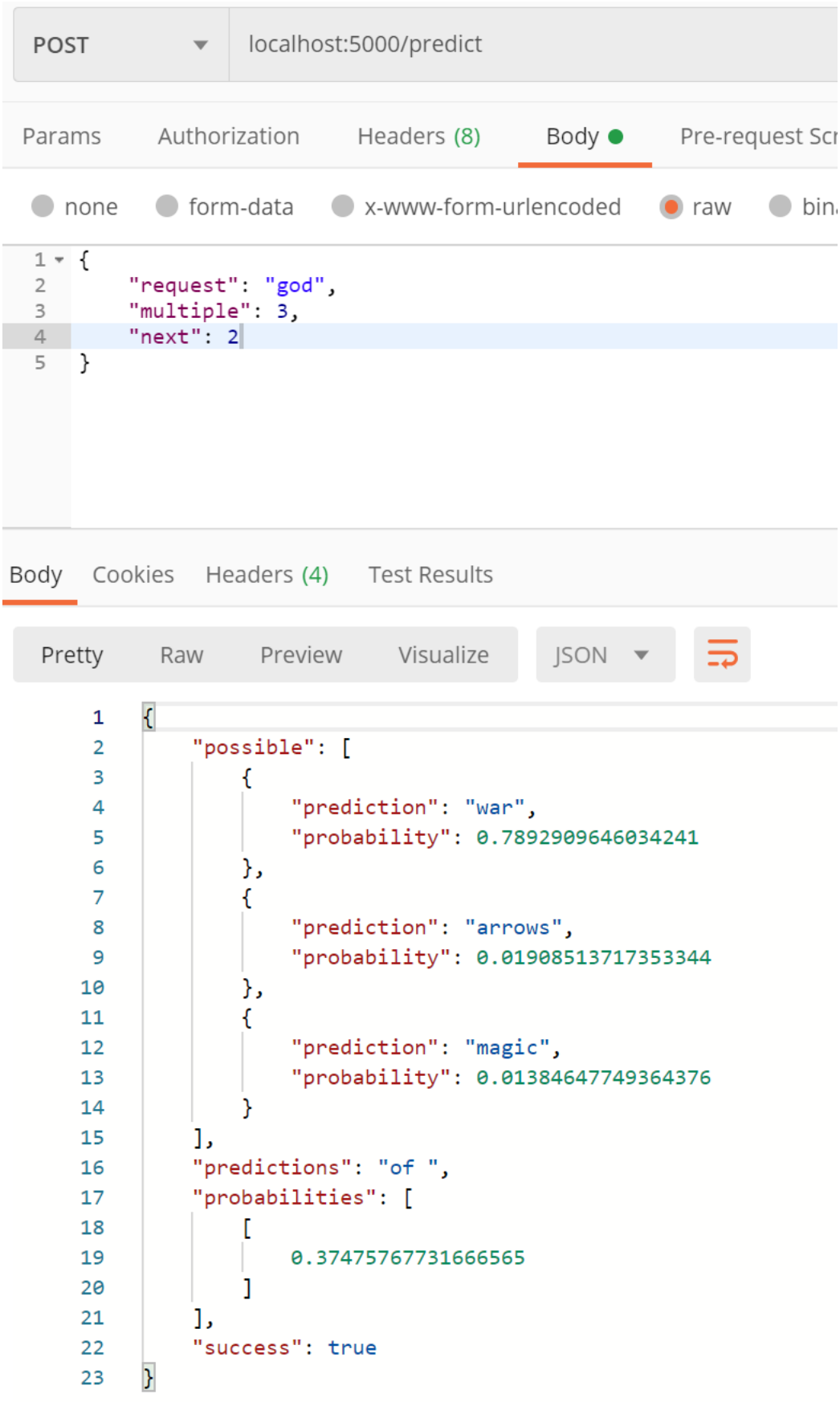


Figure 11: User prediction service /predict endpoint post request and response

A.3 Sprints

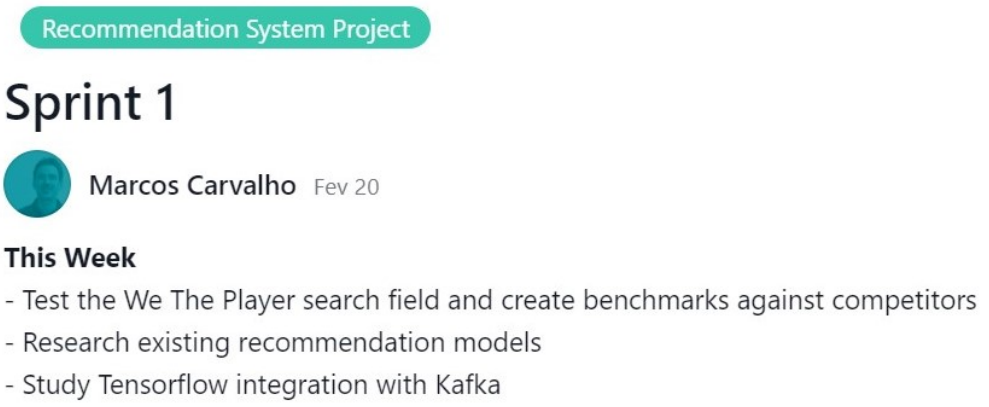


Figure 12: Sprint 1 - 20/2/2020

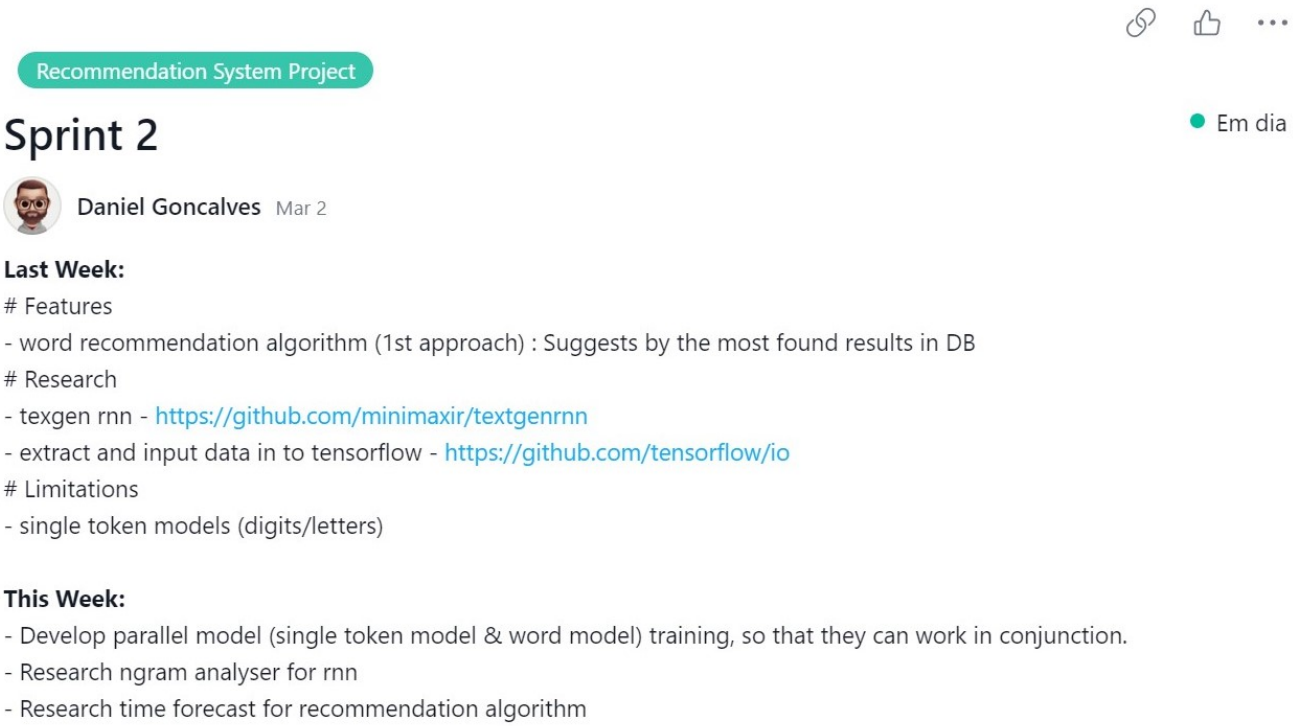


Figure 13: Sprint 2 - 2/3/2020

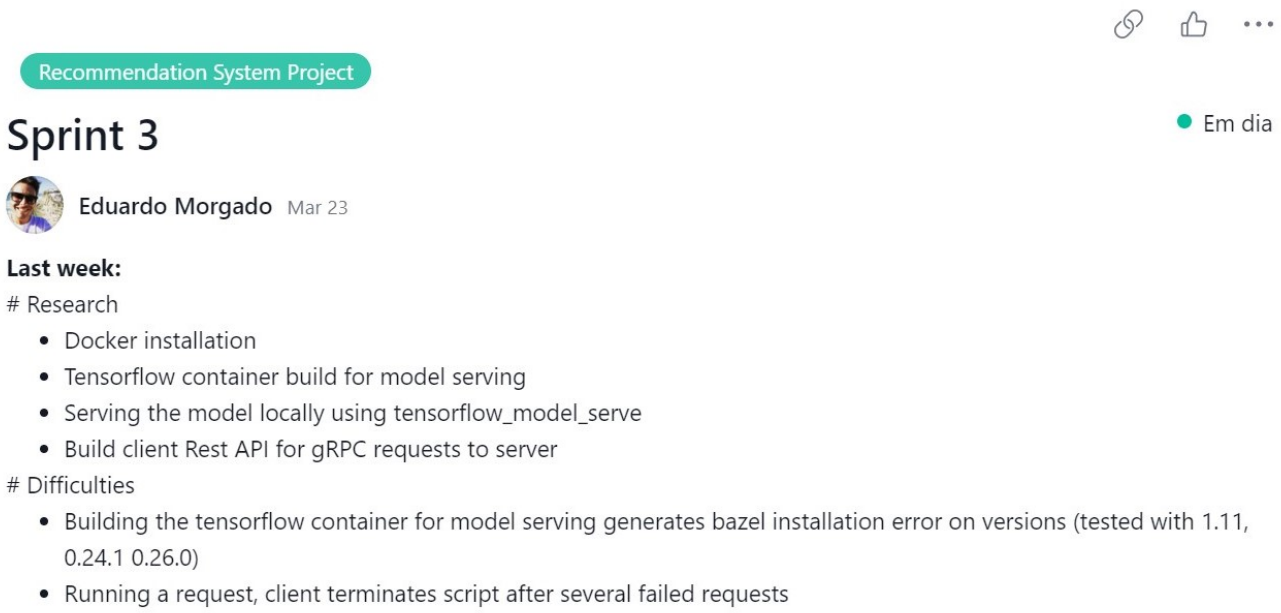


Figure 14: Sprint 3 - 23/3/2020

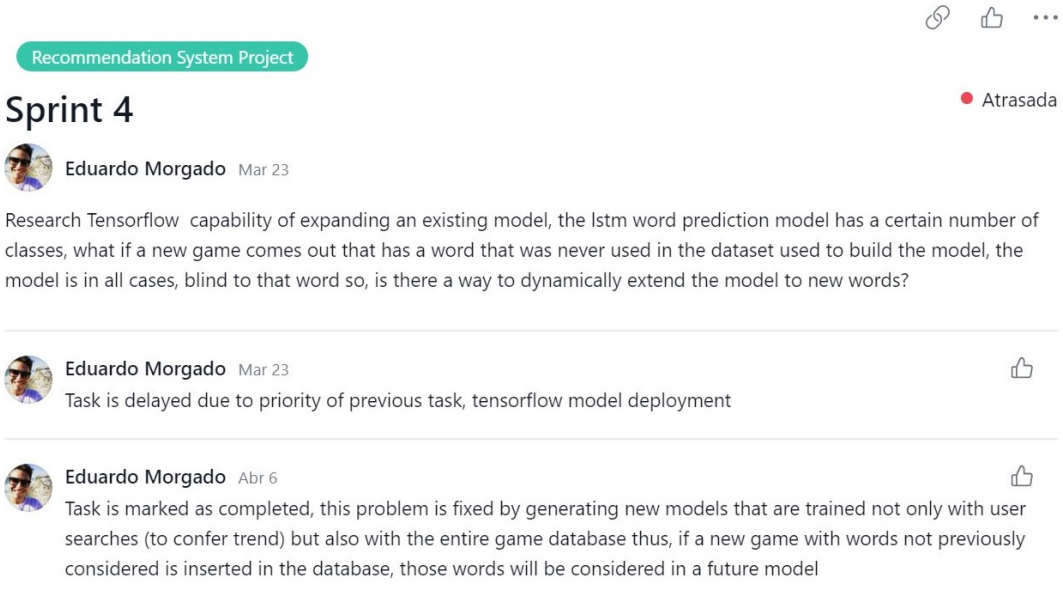


Figure 15: Sprint 4 - 23/3/2020

Sprint 5

Eduardo Morgado Abr 6

- **Current state:**
  - Data processing classes are completed
  - Model classes are completed
  - Scripts for model training, testing and retraining are completed
  - Code is compacted in easy of use scripts
  - Base model trained and servable created
  - Docker image containing servable model and TF serving image created
  - Script for client REST and gRPC requests created
  - Tested serving locally
  - Kubernetes deployment and service ymal files created to configure cluster
  - Service created supporting both REST and gRPC requests
  - Tested model serving on running kubernetes cluster
  - Started documentation of project
- **Previous challenges:**
  - Model conversion to servable, fixed with new TF version
  - Generating docker image
  - Building request scripts
  - Generating cluster and deploying model
- **Objectives for this week:**
- **Objectives for this week:**
  - Add Kubeflow support to cluster
  - Generate pipelines for training and serving on Kubeflow
- **Next Challenges**:
  - Generate project API
  - Integrate project API with wetheplayers project

Figure 16: Sprint 5 - 6/4/2020

Sprint 6

Em dia

Eduardo Morgado Abr 20

- **Last week:**
  - Tried installation of Kubeflow unsuccessful
- **Objectives for this week:**
  - Generate data and train pipelines platform independently
  - Generate cron service to retrieve training data and preprocess it, saving results to common bucket
  - Generate sequential service to data for model training
  - Generate service to deploy models (generate docker with serving image)
  - Launch serving service on replicas
  - Generate service/API to pre process user requests and perform requests to serving service, this could be done using apps like Flask to perform the requests

---

Eduardo Morgado Abr 21

Project structure was discussed:

- Main components:
  - Data container
  - Train container
  - Server container
  - Common volume

Data Container:

- Responsible for fetching data periodically, separating between database content and daily searches.
- Builds tokenizer on database words saving it to common volume

Data Container:

- Responsible for fetching data periodically, separating between database content and daily searches.
- Builds tokenizer on database words saving it to common volume
- Generates train n-gram data on database words and saves it to common volume
- Generates retrain n-gram data on searches (to consider trend) and saves it to common volume

Train container:

On volume directory update:

- Loads tokenizer from volume
- Encodes (tokenize and padding) train data (stored in volume)
- Launches TF Job:
  - Trains new LSTM model on train data
  - If fail, restart
- Encodes retrain data
- Launches TF Job
- Exports new model to protocol buffer format
- Saves new serve model on volume

Server Container

- Installs tensorflow_model_server
- Launches model server with model loaded from common volume
- Launches Flask app responsible for:
  - Receiving user request
  - Encoding request to be model friendly (integer)
  - Requests prediction to model server on encoded user request
  - Decodes model server prediction response to be user friendly (string)
  - Returns decode prediction to user

Eduardo Morgado Abr 21

For a visual guide: https://drive.google.com/open?id=1nhKZXWvSawmueL7SbZm-62ssuhny-ZEn

Figure 17: Sprint 6 - 20/4/2020

Sprint 7

Em dia

Eduardo Morgado Maio 4

Last week:

- Serve container was finished, allowing for encoding/decoding of predictions
- Data fetch system was finished with flask, since we will be fetching single files with multiple content, instead of multiple files
- Model functionality was changed to incorporate transfer learning and batch training
- Training was added to fetch service

Objectives for this week:

- Test all components in kubernetes
- After fetching, training and serving services, develop a system to allow fetch service to retrieve multiple data entries without overloading the server

Daniel Goncalves Maio 5

@Rodrigo Flaminio:

- Guardar as pesquisas (searches) do ES, utilizar o Kibana

Eduardo Morgado Maio 5

@Eduardo Morgado

- Show probabilities for model predictions
- Correct embedding lookup error in data fetch model build
- Update data fetch request to monitor DB changes and build/train model accordingly
- Test shared fs in docker using Minio
- Prepare data fetch system for trend integration (recent searches)

Daniel Goncalves Há 24 dias

Continue previous sprint with updates in search history retrieval. Instead of using elasticsearch, store each search after prediction in redis list per day. The cron job will consume the daily list and delete after.

Figure 18: Sprint 7 - 4/5/2020

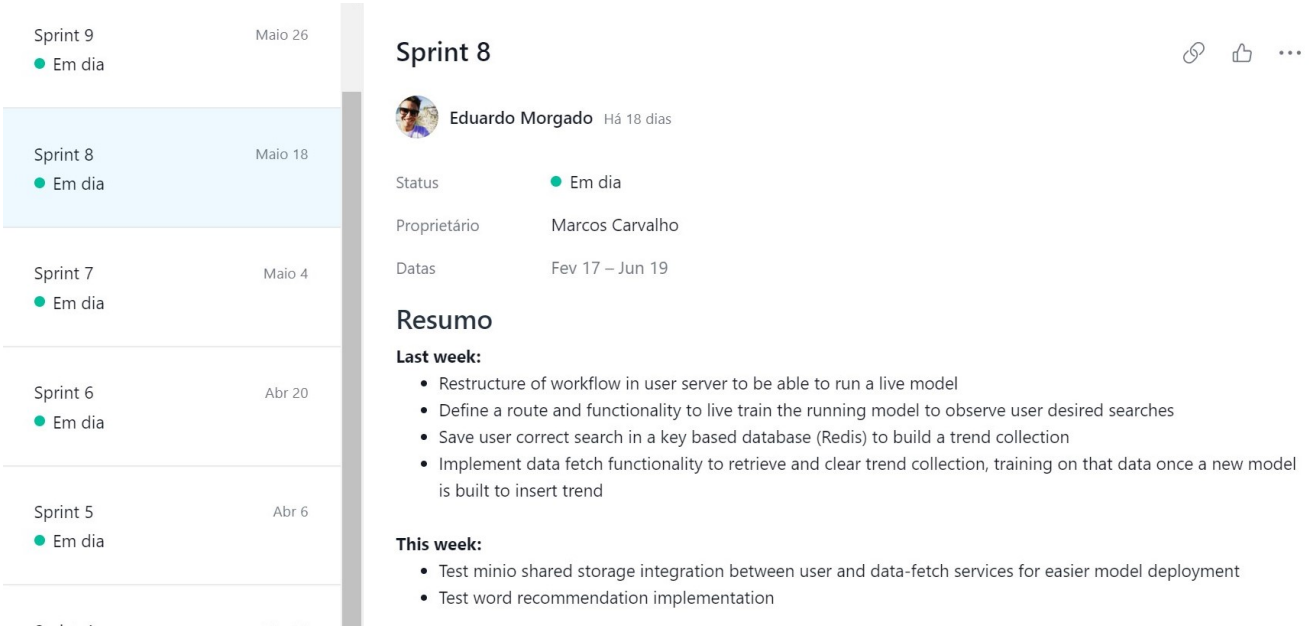
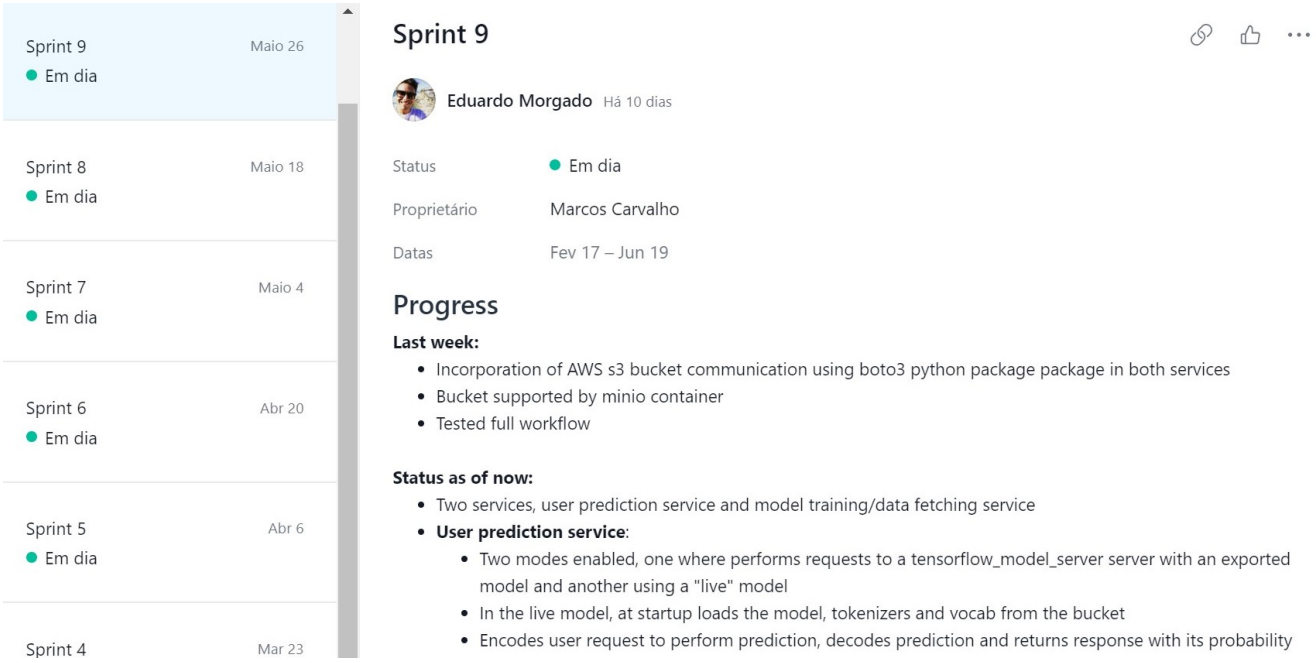


Figure 19: Sprint 8 - 18/5/2020



- Encodes user request to perform prediction, decodes prediction and returns response with its probability values
- Supports and endpoint to provide user correct/desired searches where for each search it will train the live model granting it real time trend predictions and stores the trend to a Redis database where it will be used for the new model trained

- **Data fetch service:**
  - Periodically check the database for new entries, if new data/games are found since the last check, it will retrieve the new content and generate a new model from all the updated database
  - If no new data is found it will jump to the retrain process
  - The present model (new if new data is found) is then trained for trends, fetching the trend data stored in the redis database
  - After training processes end the model is then deployed, stored in a bucket
  - The service will then send a signal to the user prediction service to update its model, loading it and all necessary files from the bucket

This week:

- After testing the workflow locally we will now move to a server test
- After the server testing were we will analyse the accuracy of predictions and the time required to train, we will then move to integration with the WTP service

Figure 20: Sprint 9 - 26/5/2020

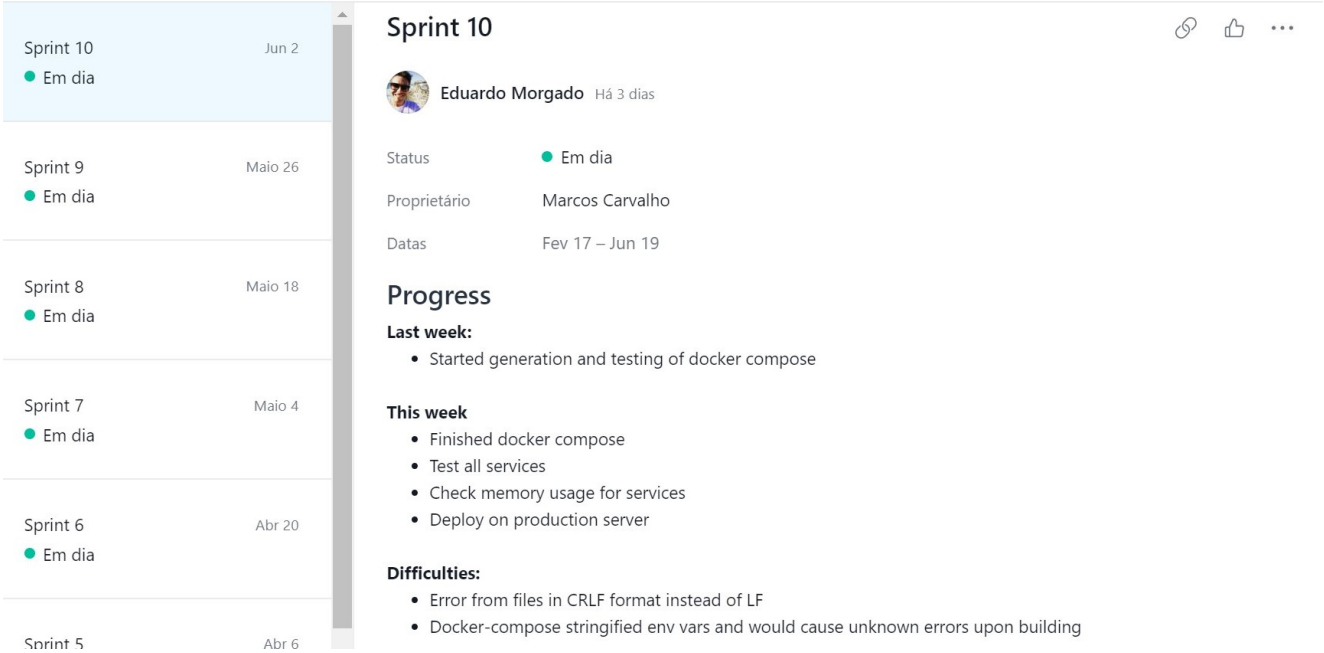


Figure 21: Sprint 10 - 2/6/2020

A.4 Tasks

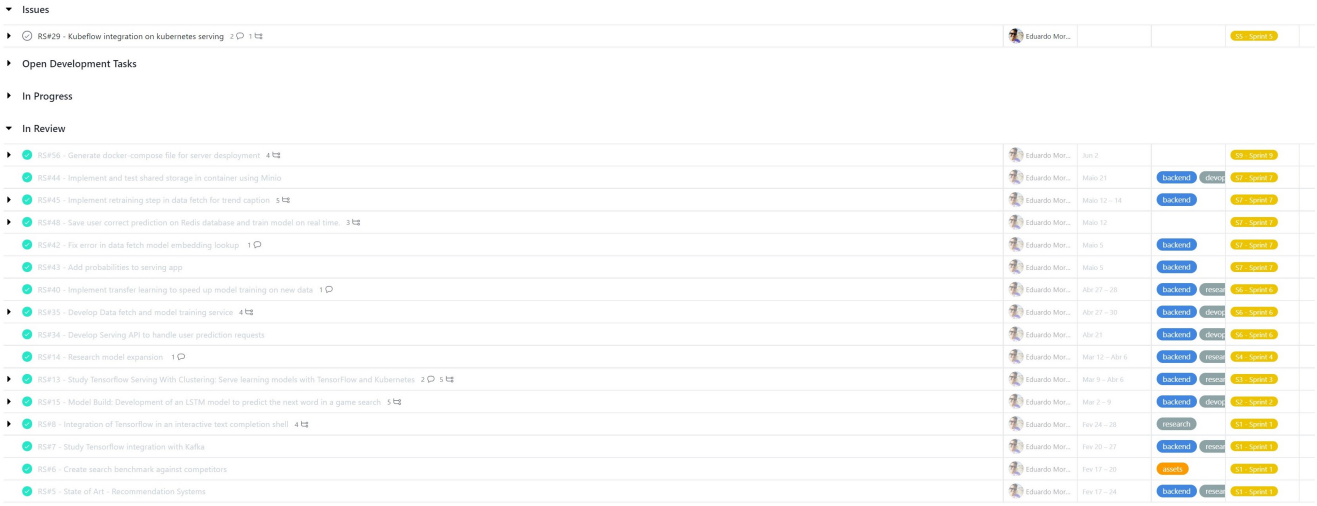


Figure 22: Tasks - Complete list

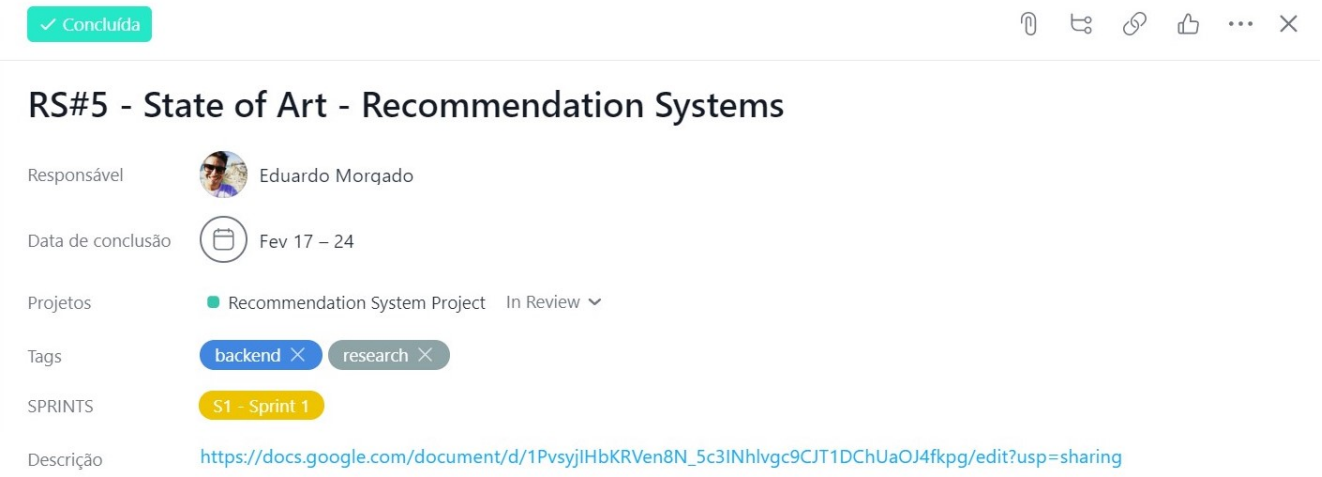


Figure 23: Task 1 - from 17/2/2020 to 24/2/2020

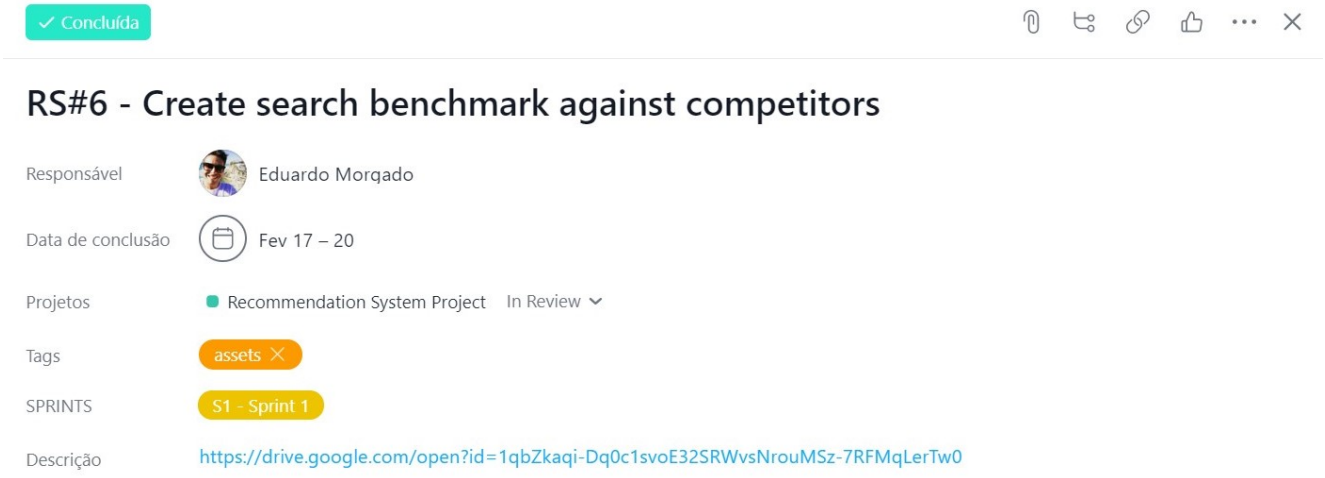


Figure 24: Task 2 - from 17/2/2020 to 20/2/2020

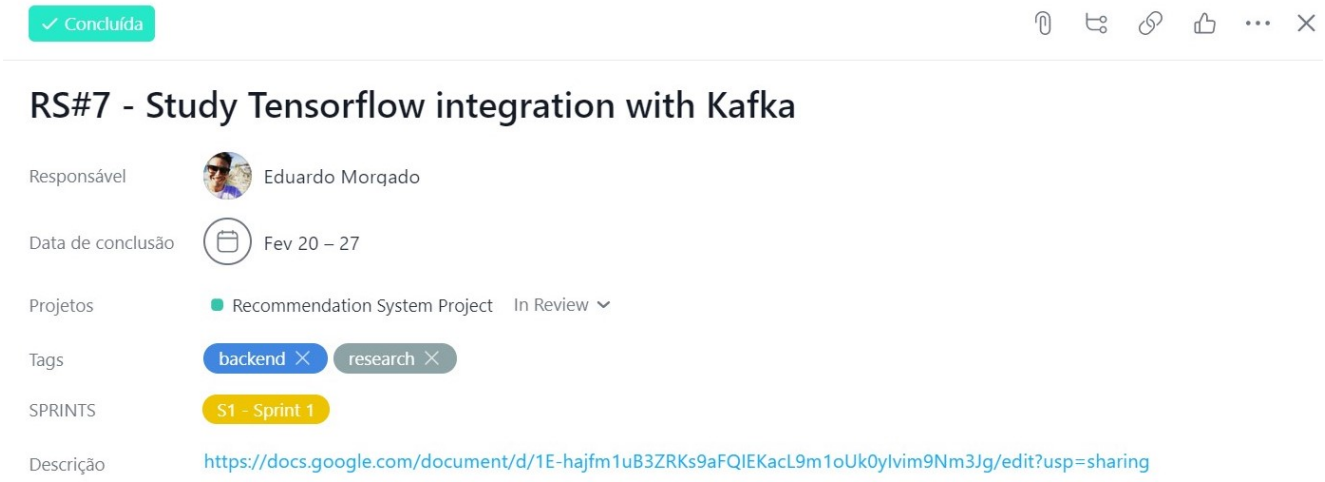


Figure 25: Task 3 - from 20/2/2020 to 27/2/2020

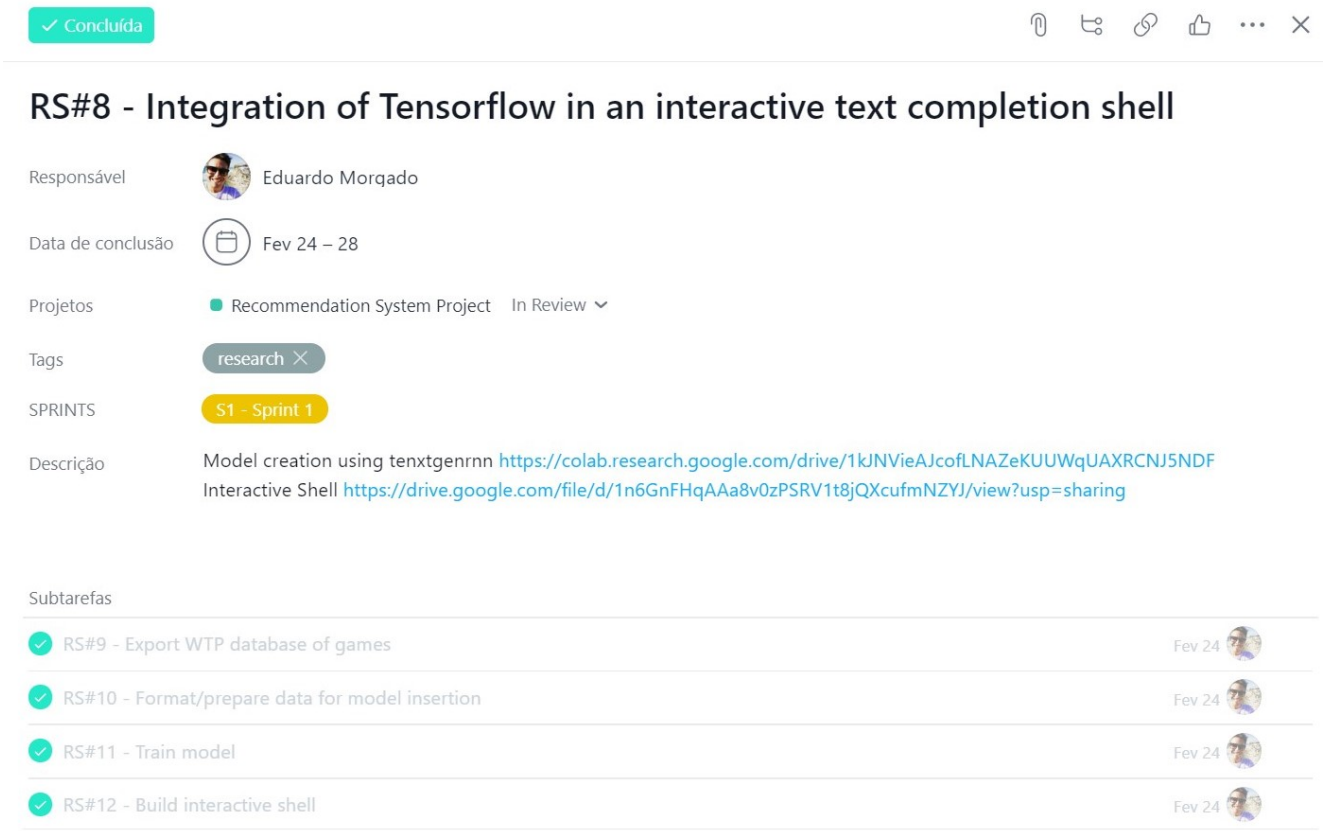


Figure 26: Task 4 - from 24/2/2020 to 28/2/2020

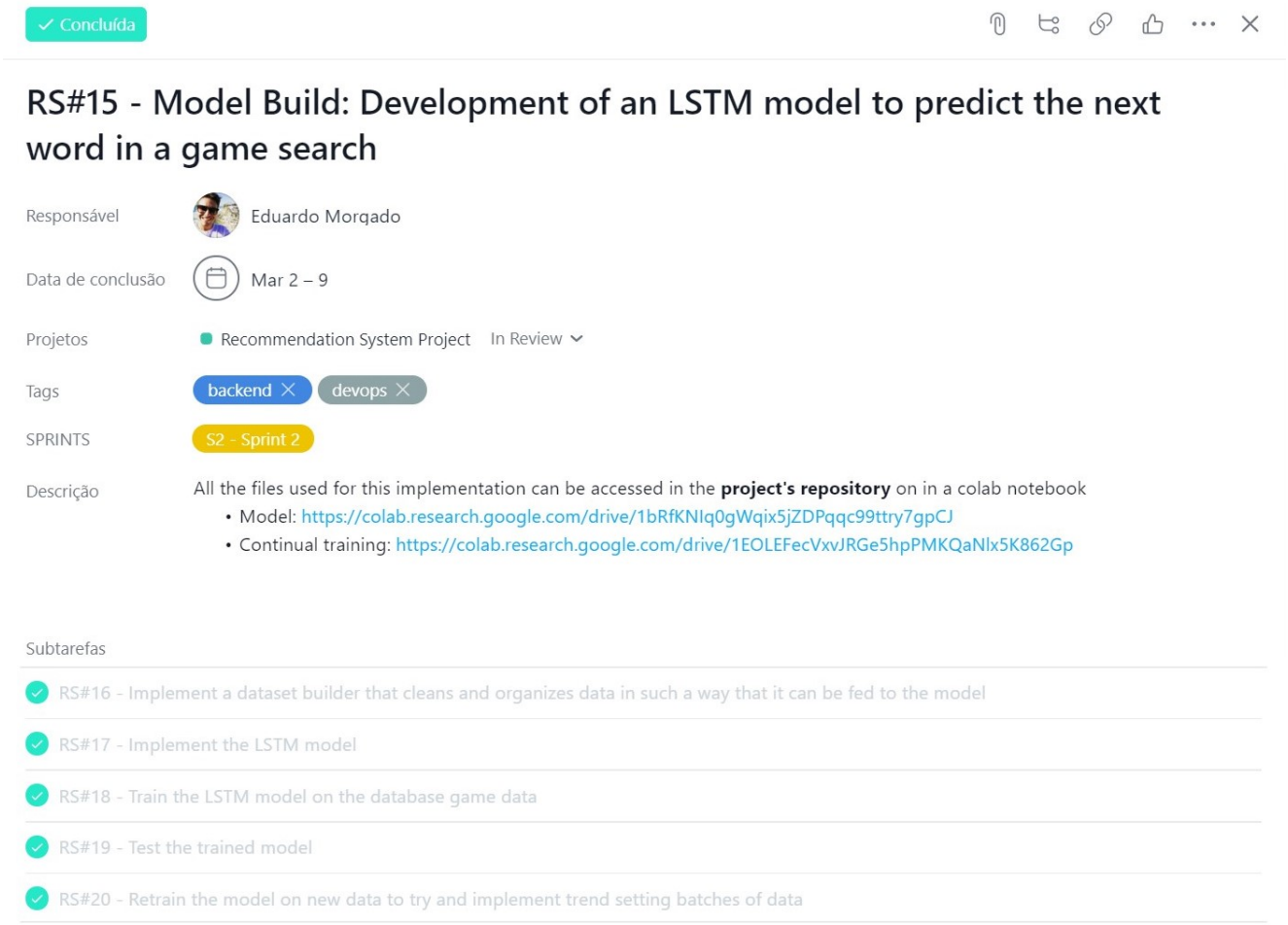


Figure 27: Task 5 - from 2/3/2020 to 9/3/2020

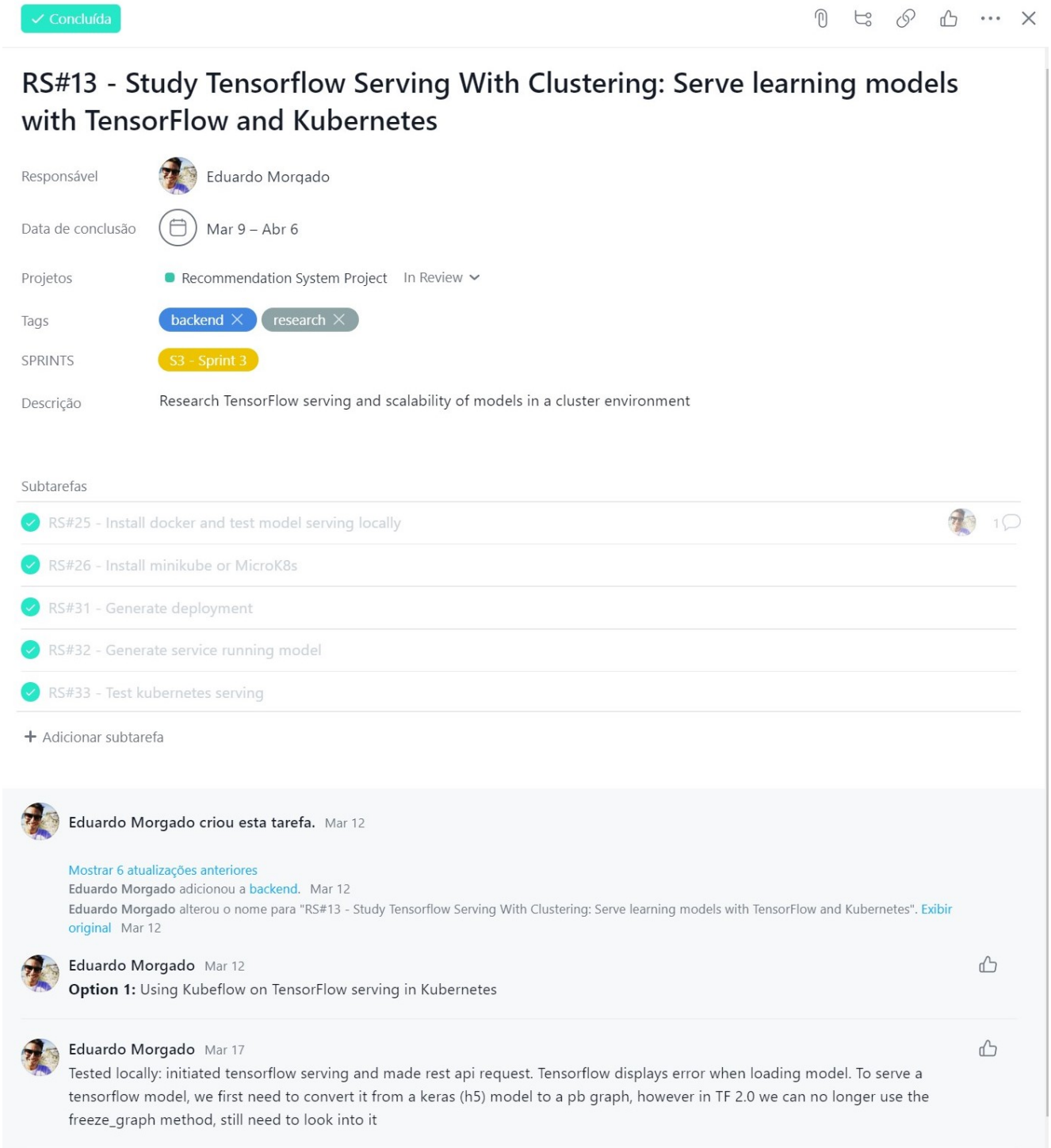


Figure 28: Task 6 - from 9/3/2020 to 6/4/2020

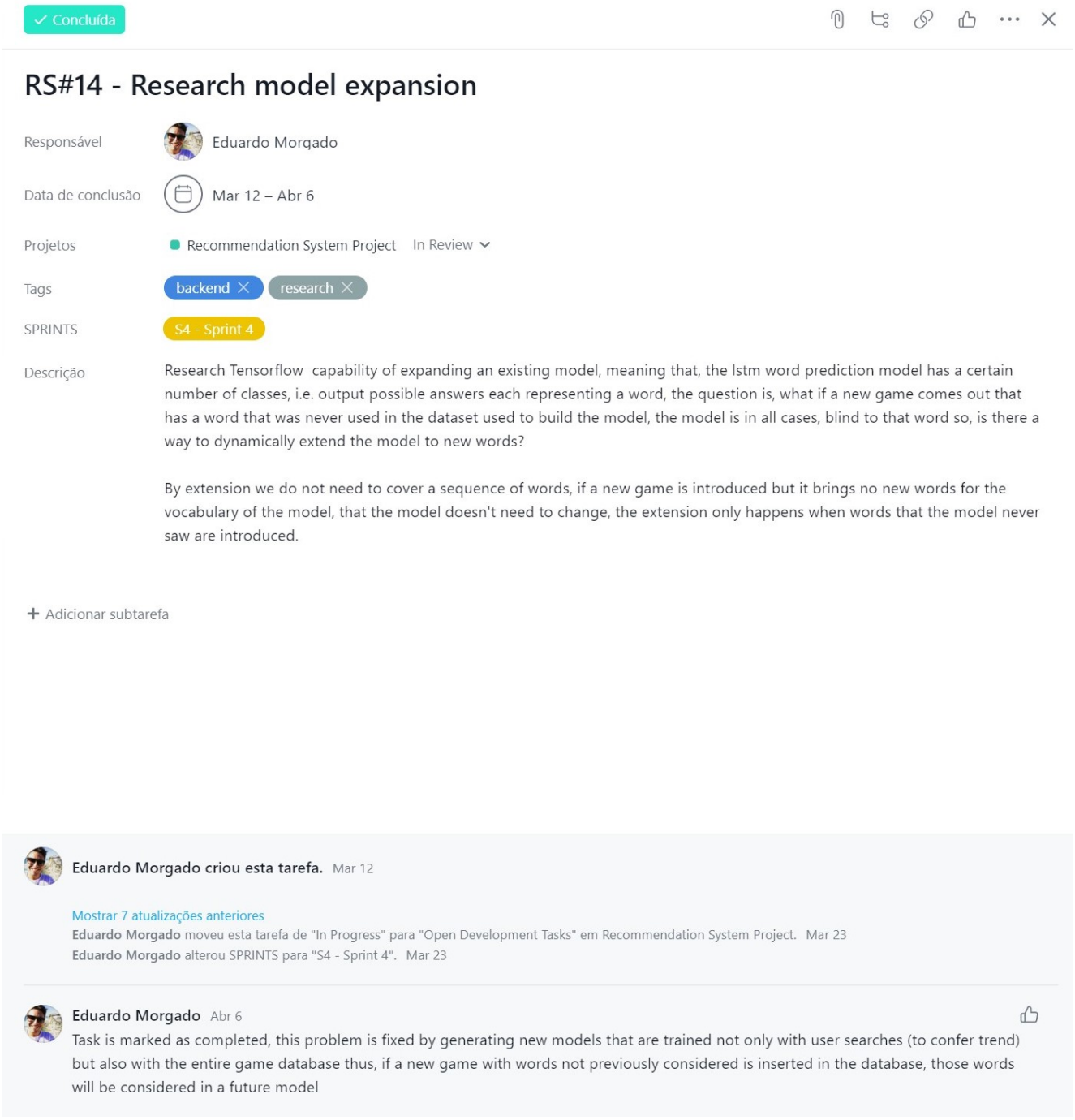


Figure 29: Task 7 - from 12/3/2020 to 6/4/2020

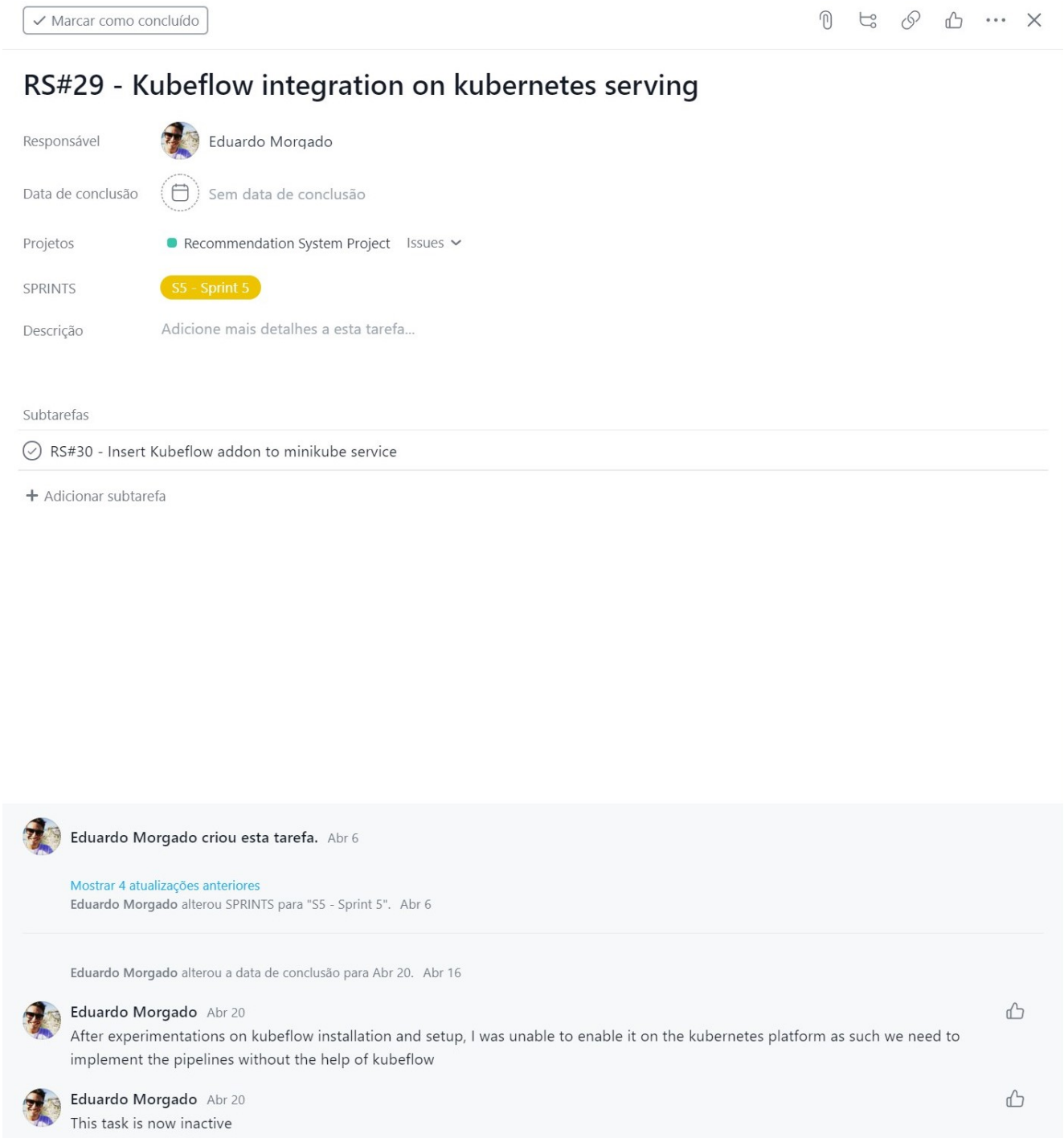


Figure 30: Task 8 - from 6/4/2020 to 20/4/2020, inactive due to issue

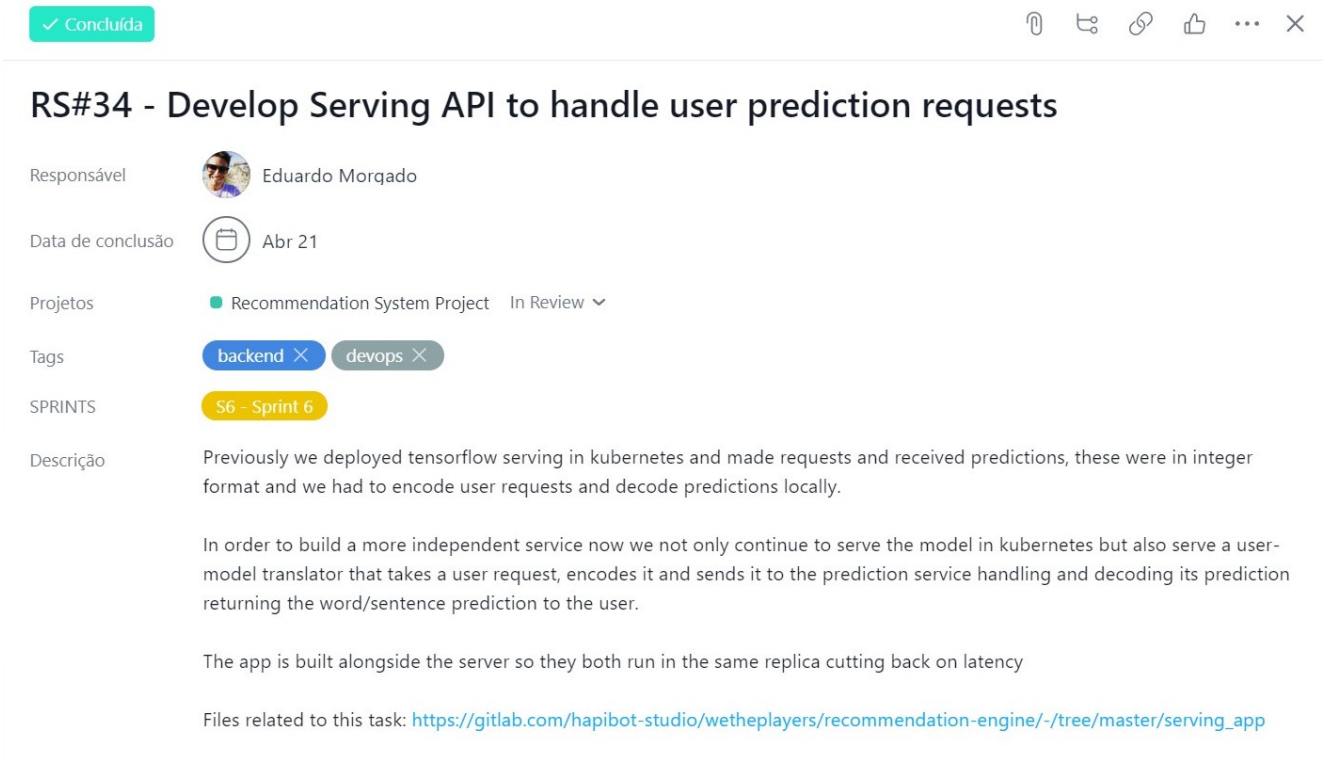


Figure 31: Task 9 - 21/4/2020

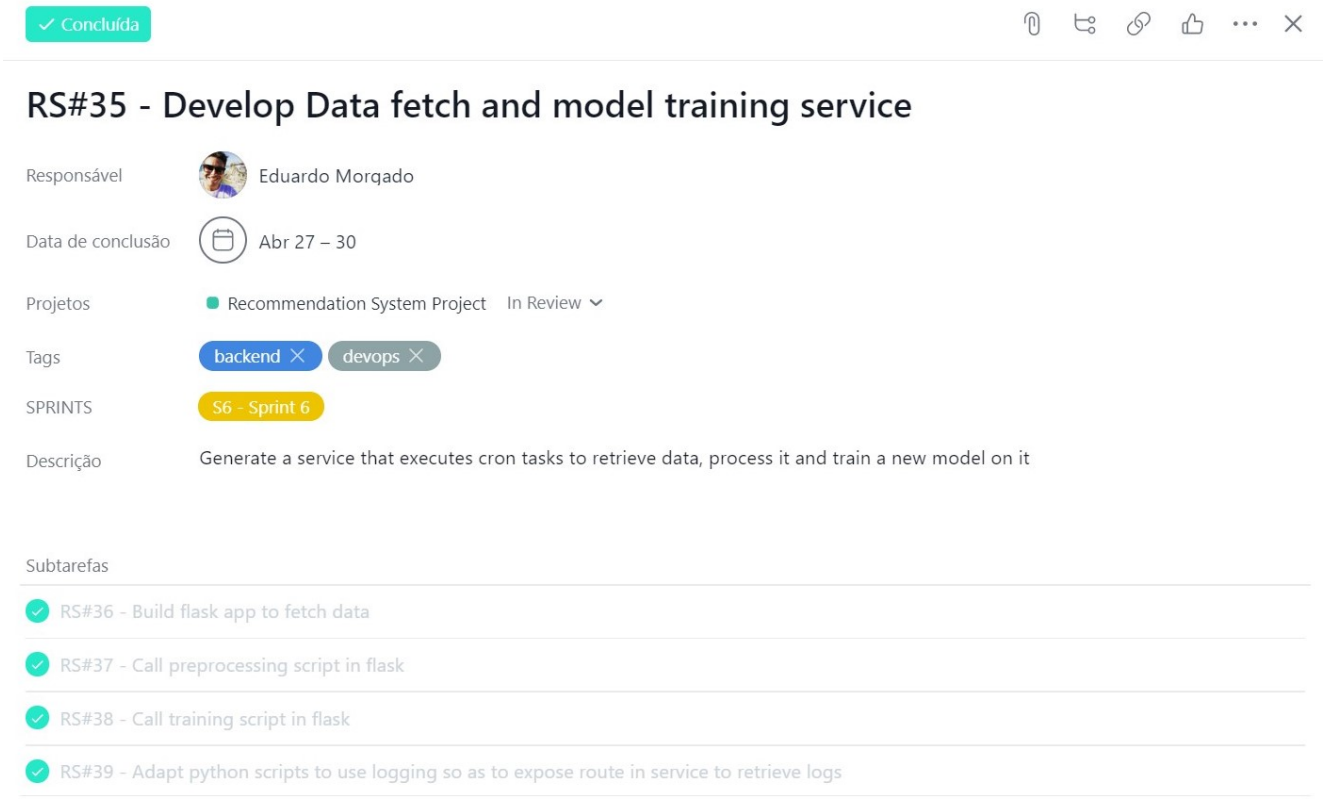


Figure 32: Task 10 - from 27/4/2020 to 30/4/2020

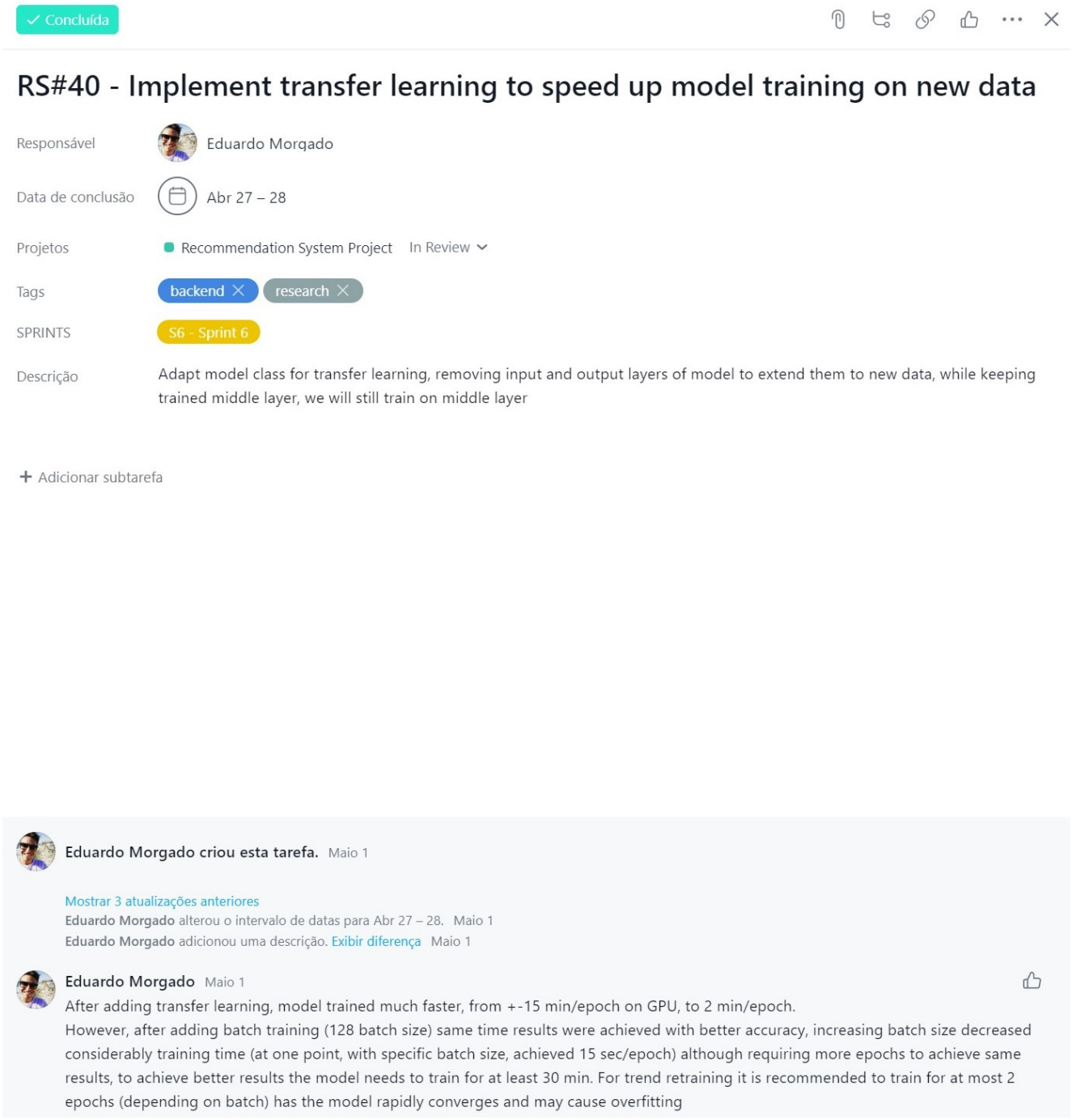


Figure 33: Task 11 - from 27/4/2020 to 28/4/2020

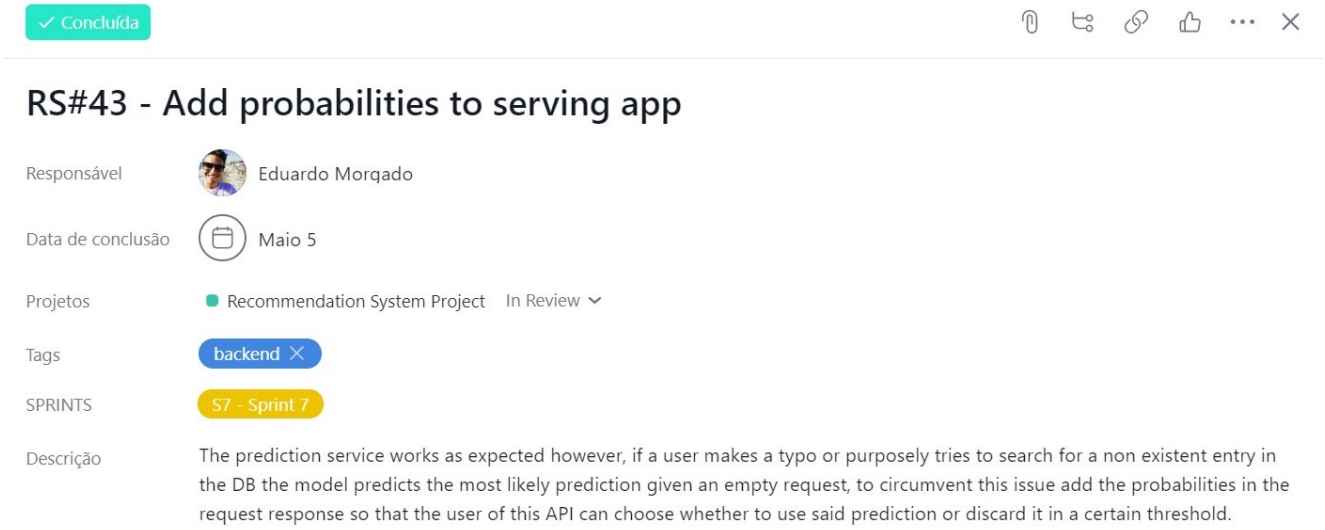


Figure 34: Task 12 - 5/5/2020

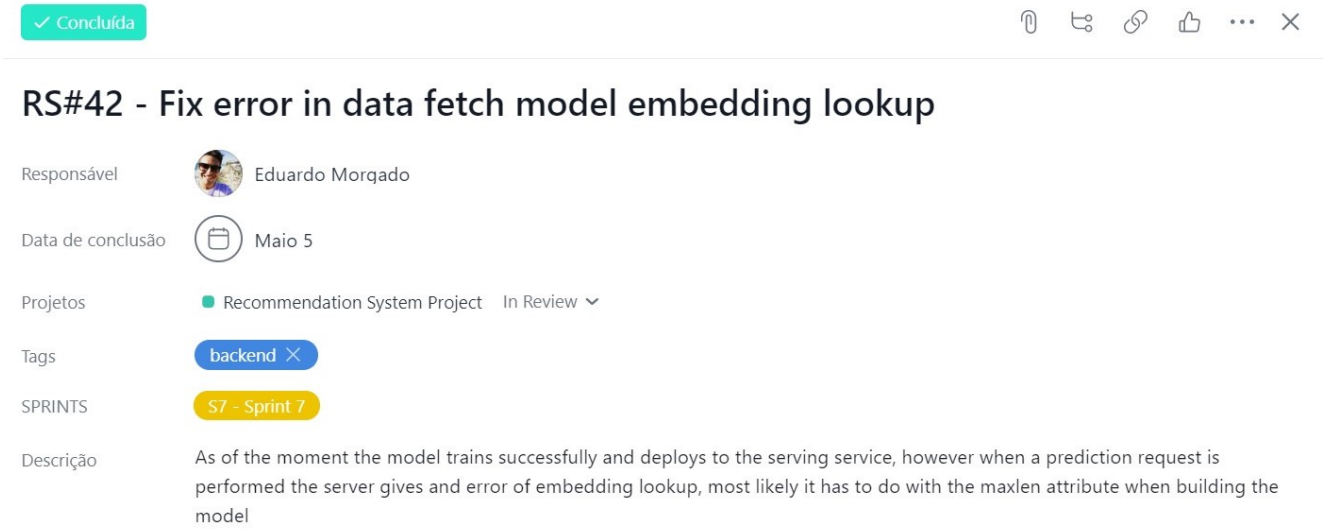


Figure 35: Task 13 - 5/5/2020

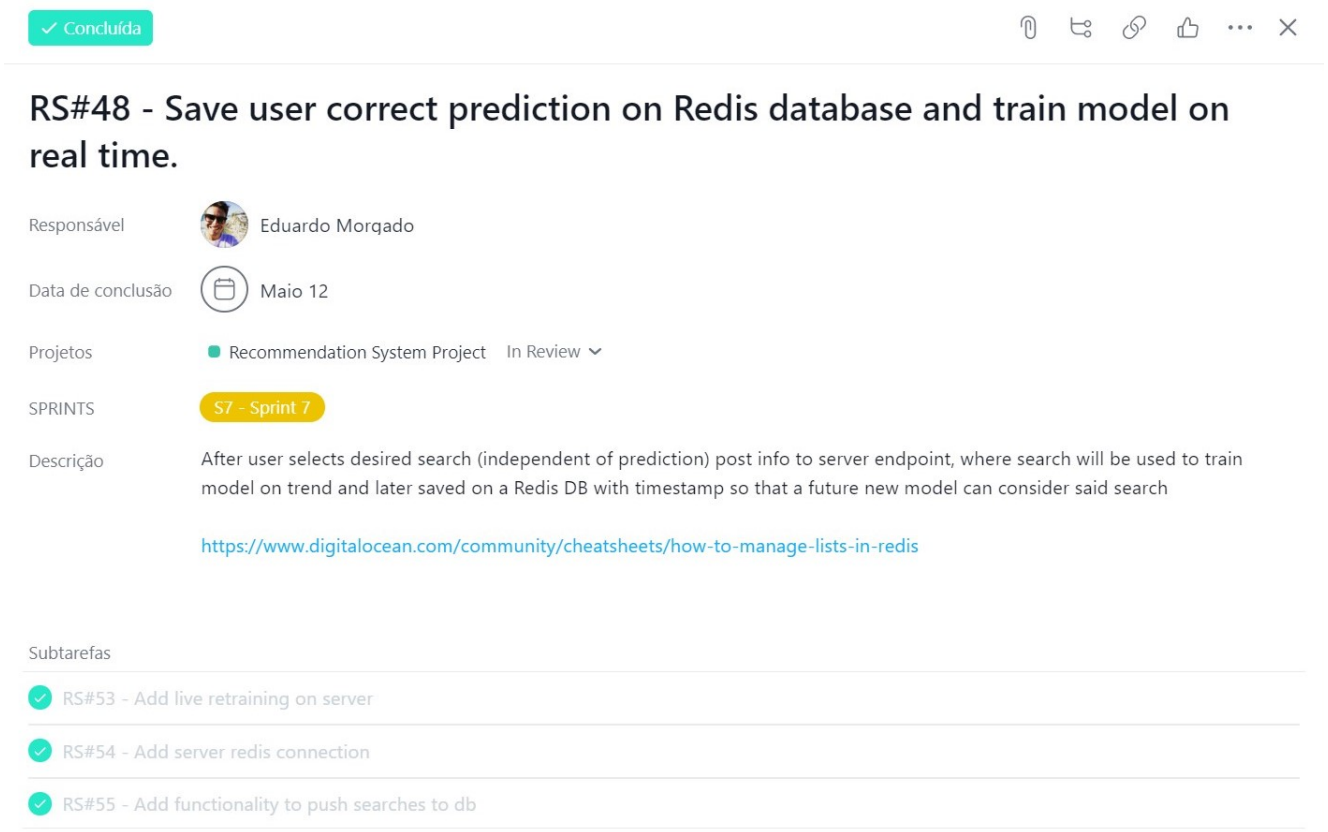


Figure 36: Task 14 - 12/5/2020

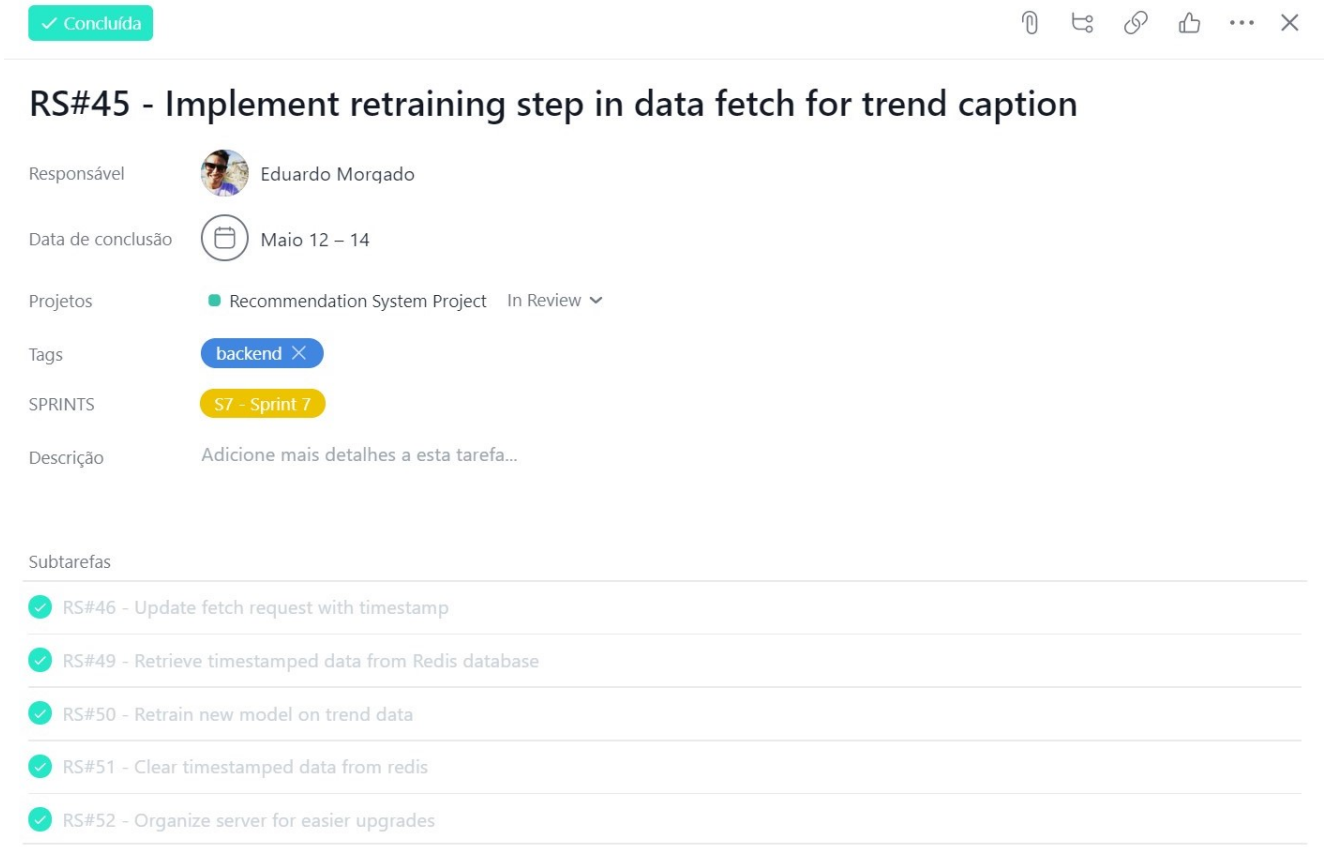


Figure 37: Task 15 - from 12/5/2020 to 14/5/2020

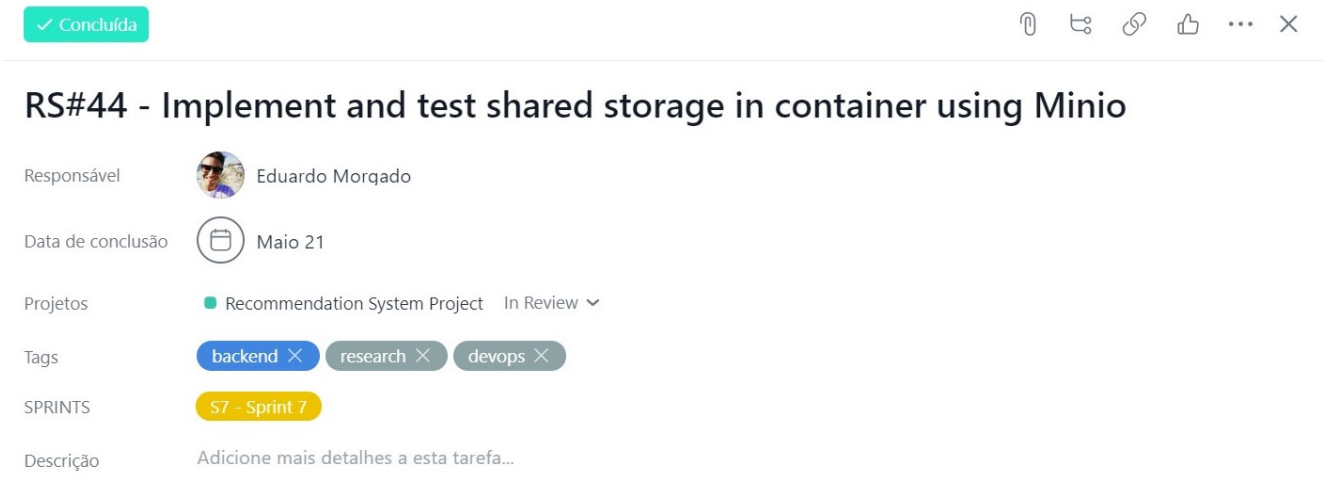


Figure 38: Task 16 - 21/5/2020

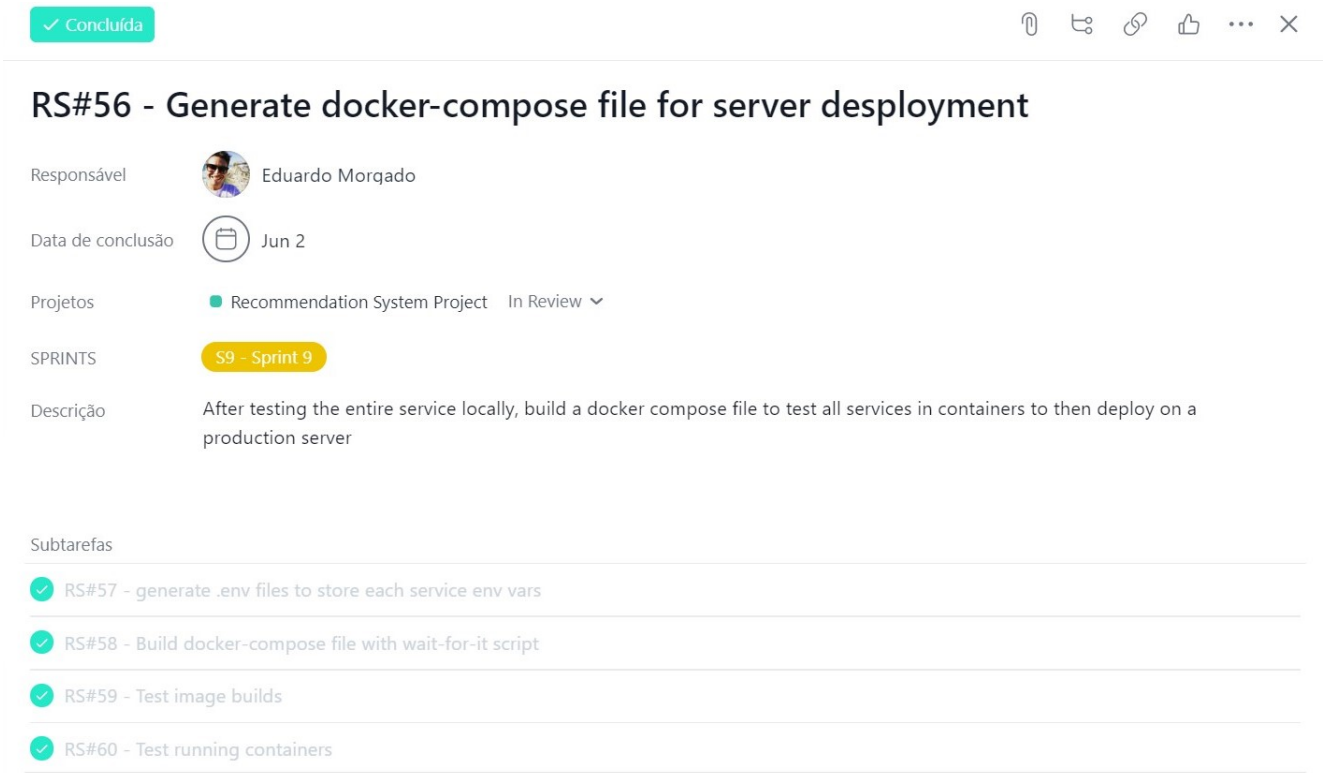


Figure 39: Task 17 - 2/6/2020

A.5 Diagrams

A.5.1 Development scheme

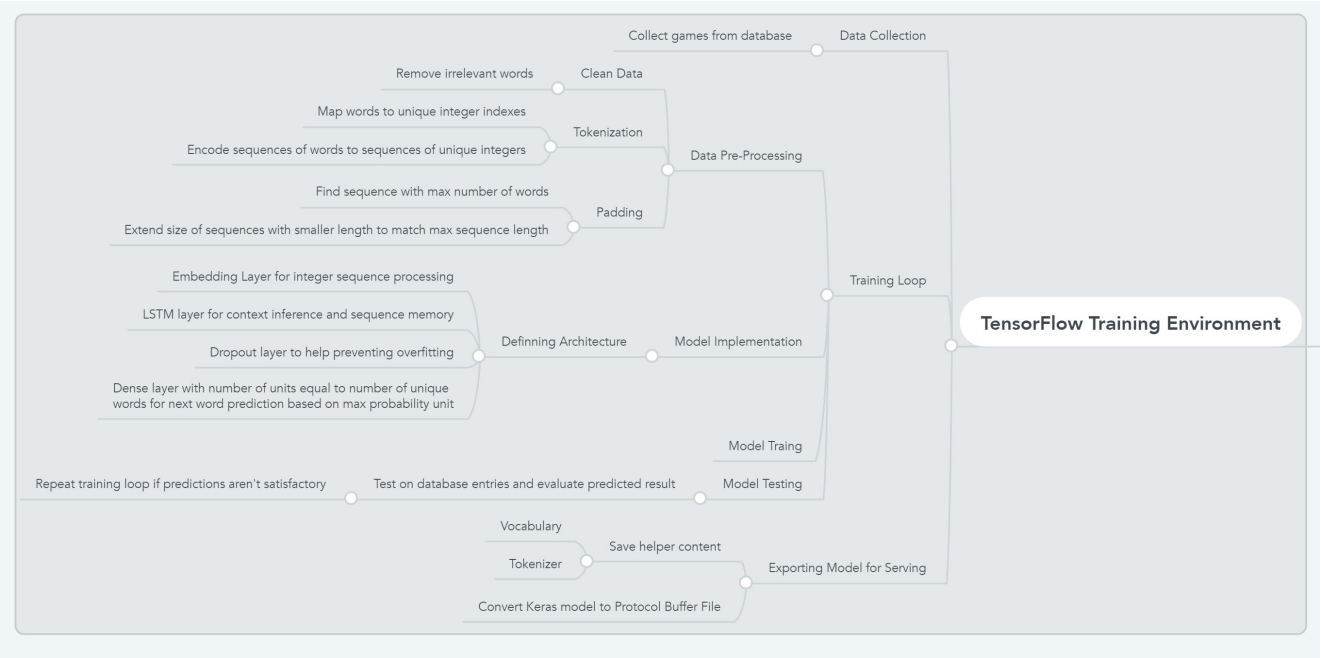


Figure 40: Scheme 1 - training environment

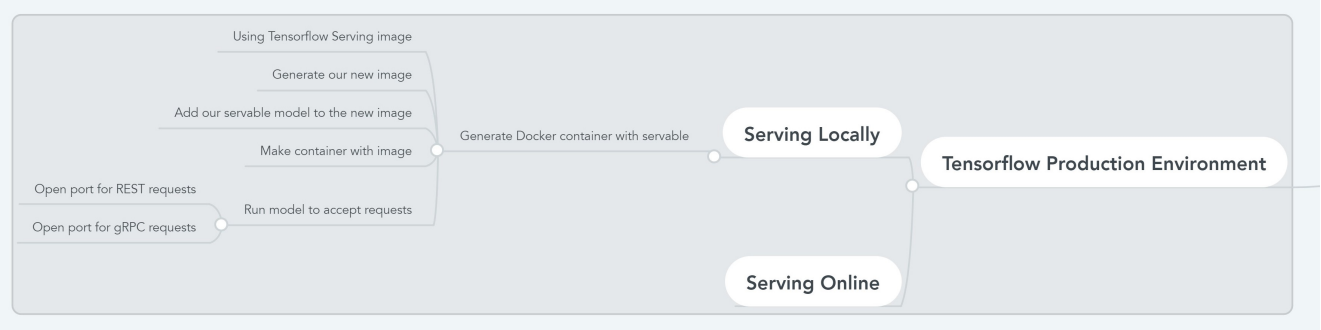


Figure 41: Scheme 2 - local serving in production

Figure 42: Scheme 2 - online scalable serving in production

A.5.2 Pipeline Design

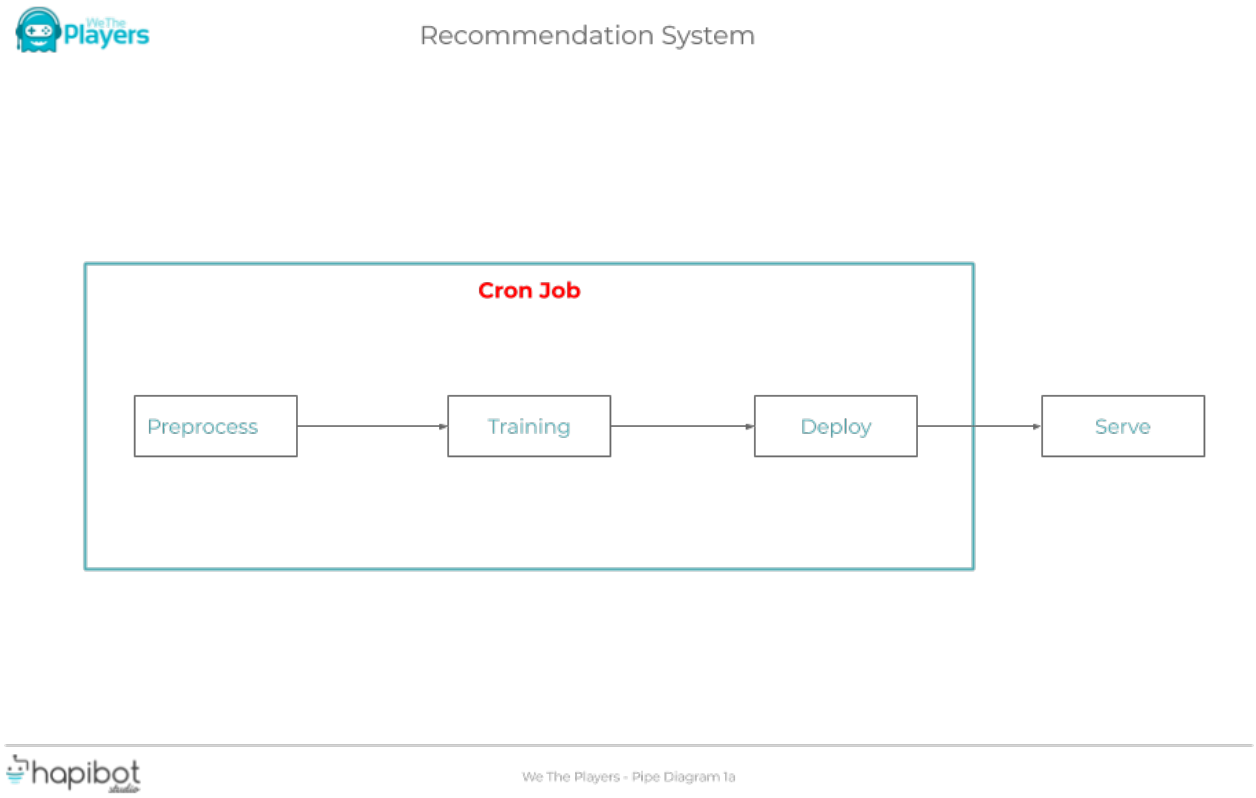


Figure 43: Pipeline general design

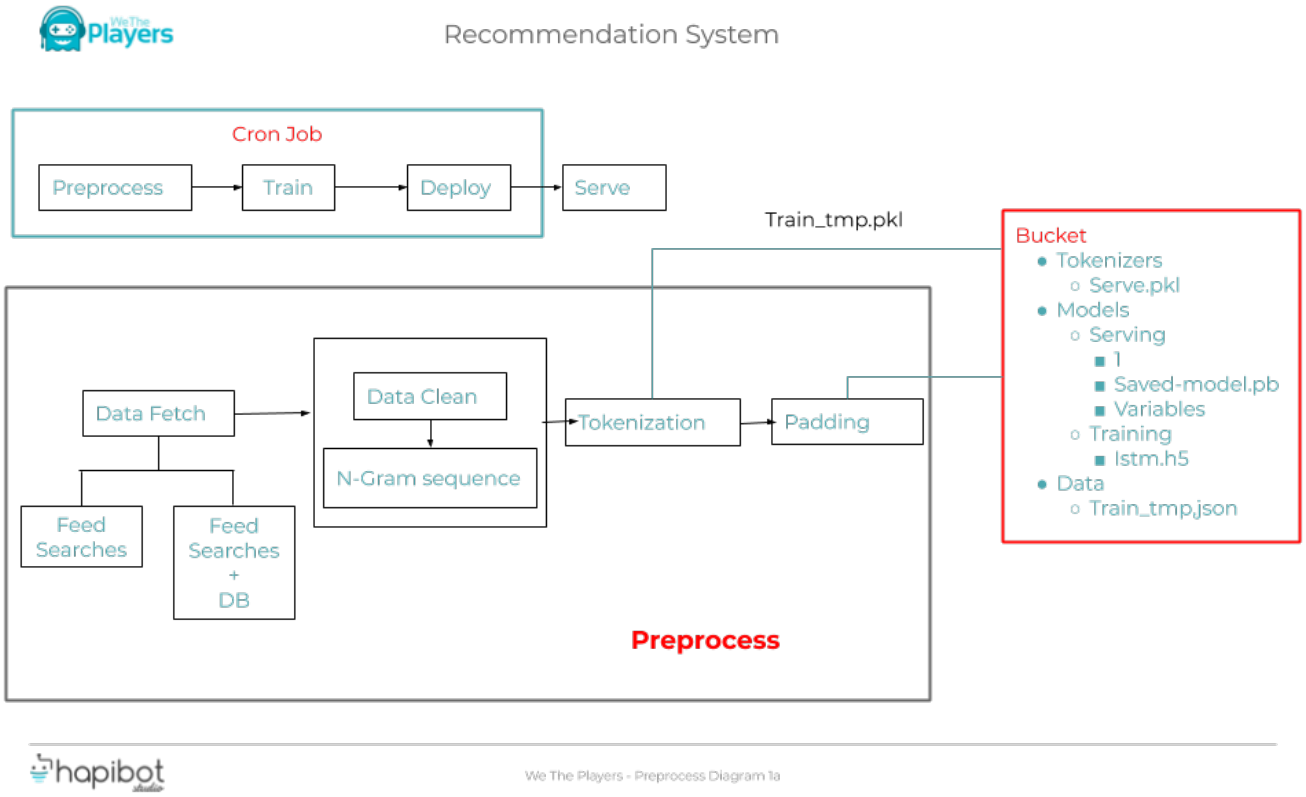


Figure 44: Pipeline 1 - preprocessing stage

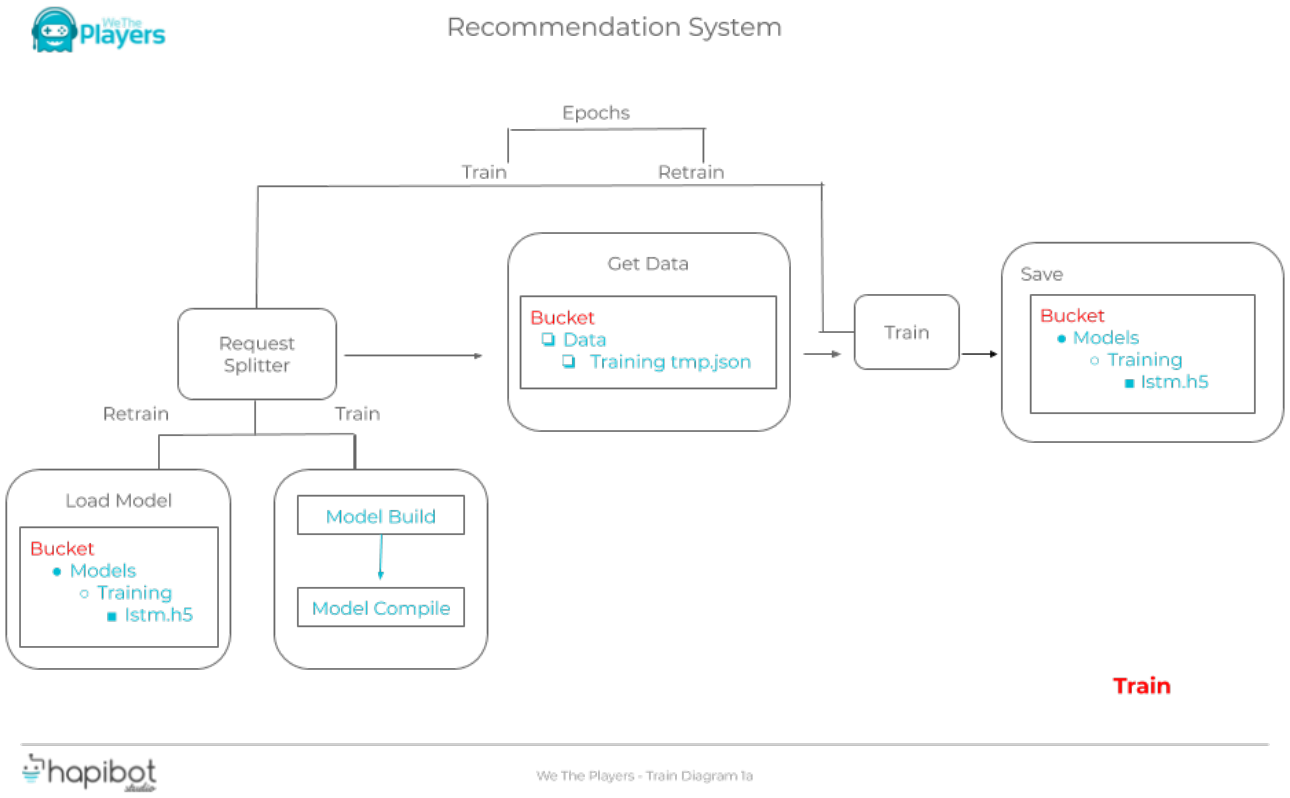


Figure 45: Pipeline 2 - training stage

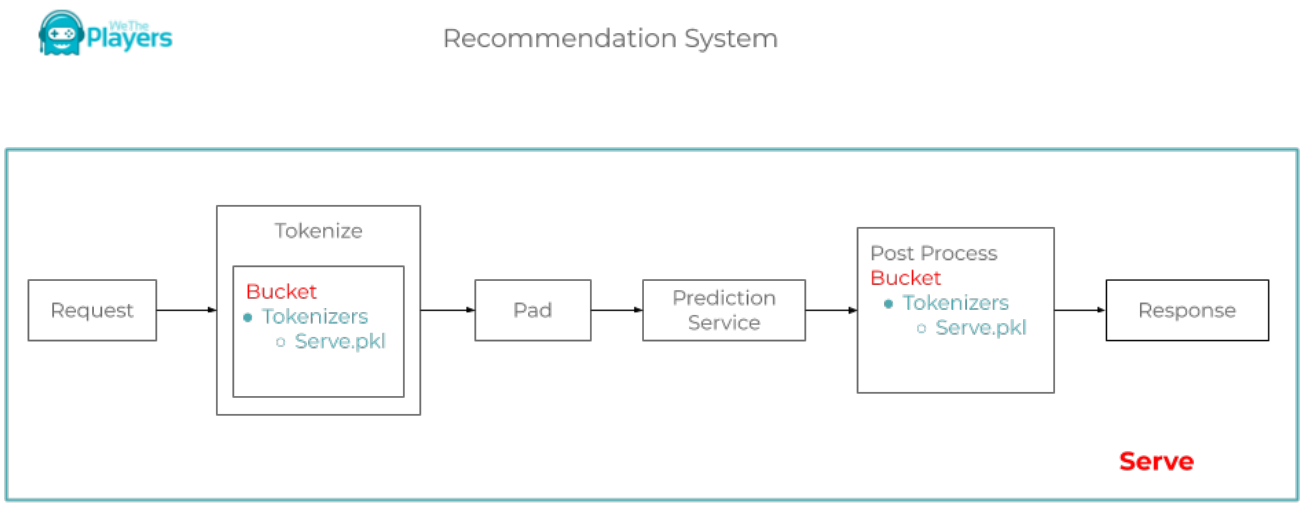


Figure 46: Pipeline 3 - serving stage

A.5.3 Kubernetes Design

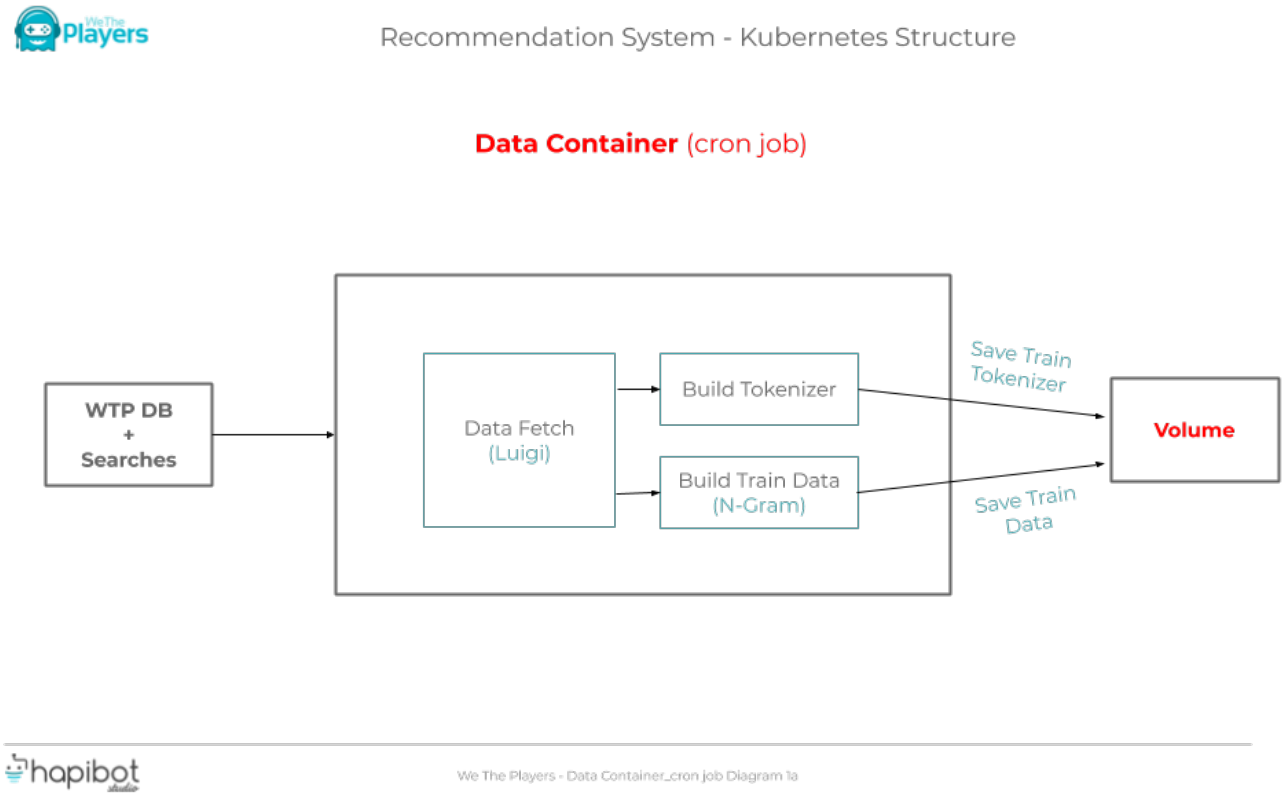


Figure 47: Kubernetes 1 - data fetch container

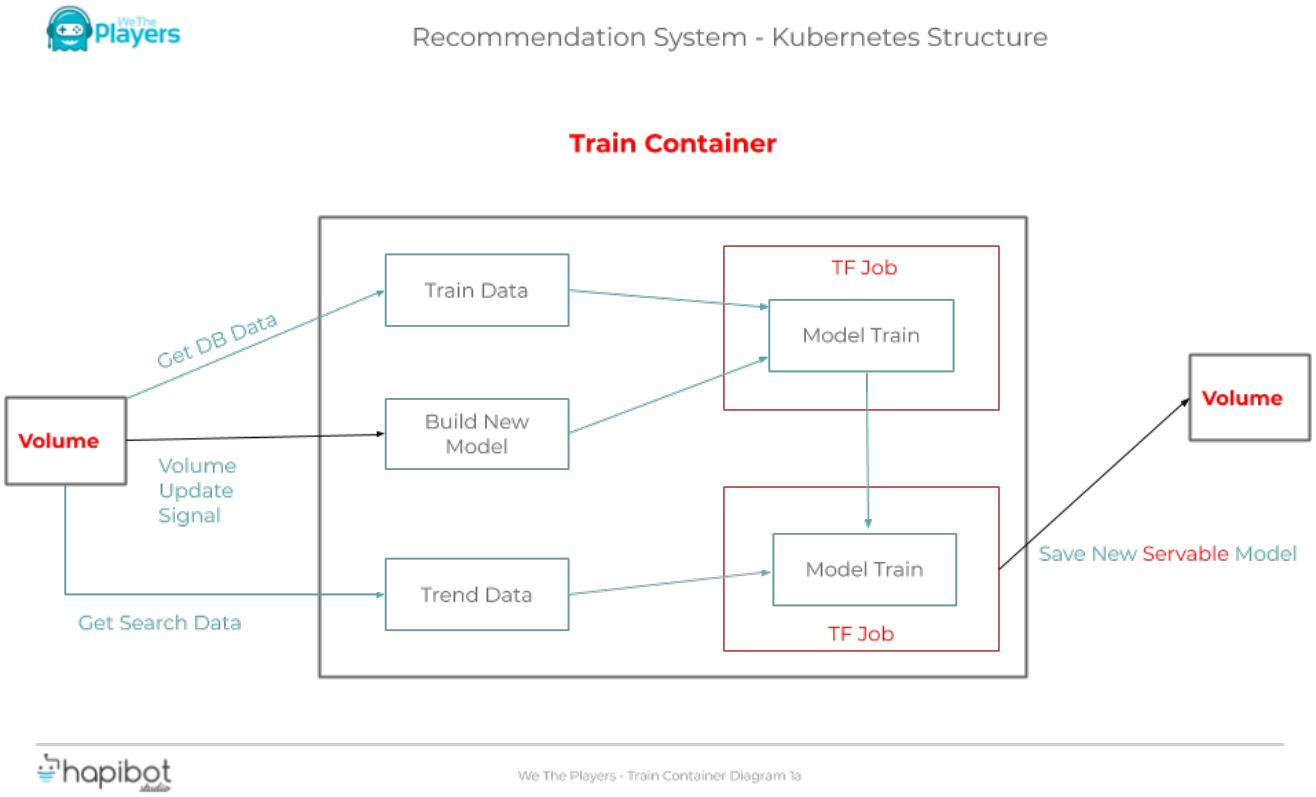


Figure 48: Kubernetes 2 - training container

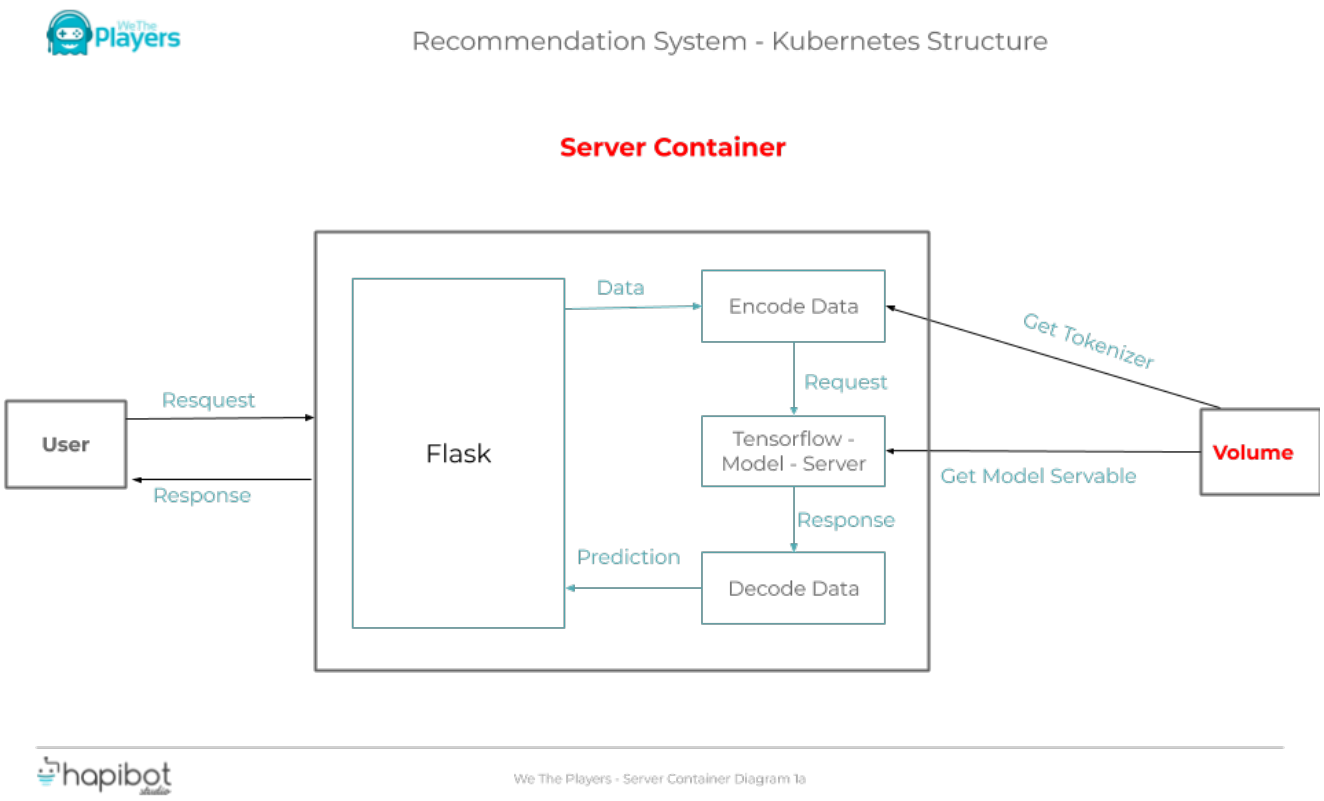


Figure 49: Kubernetes 3 - serving container

A.5.4 Final Product Design

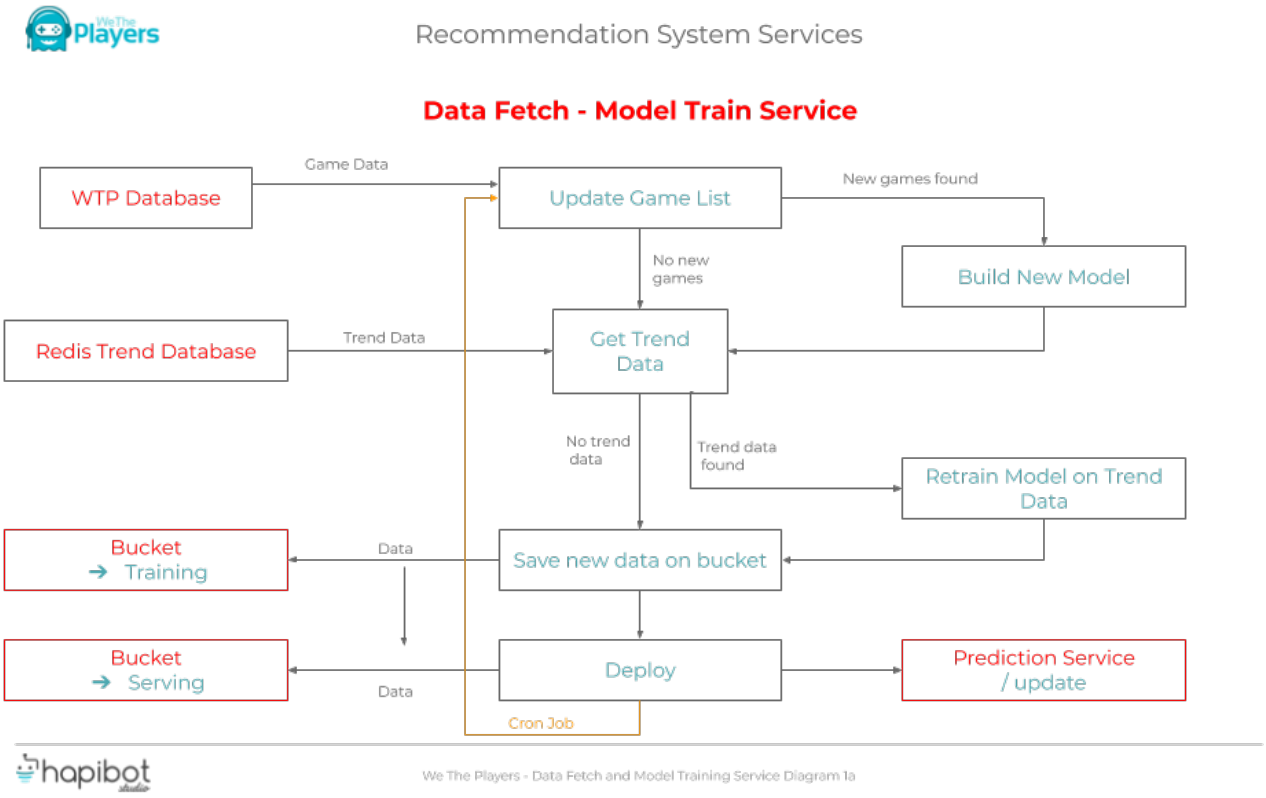


Figure 50: Data fetch and model training service

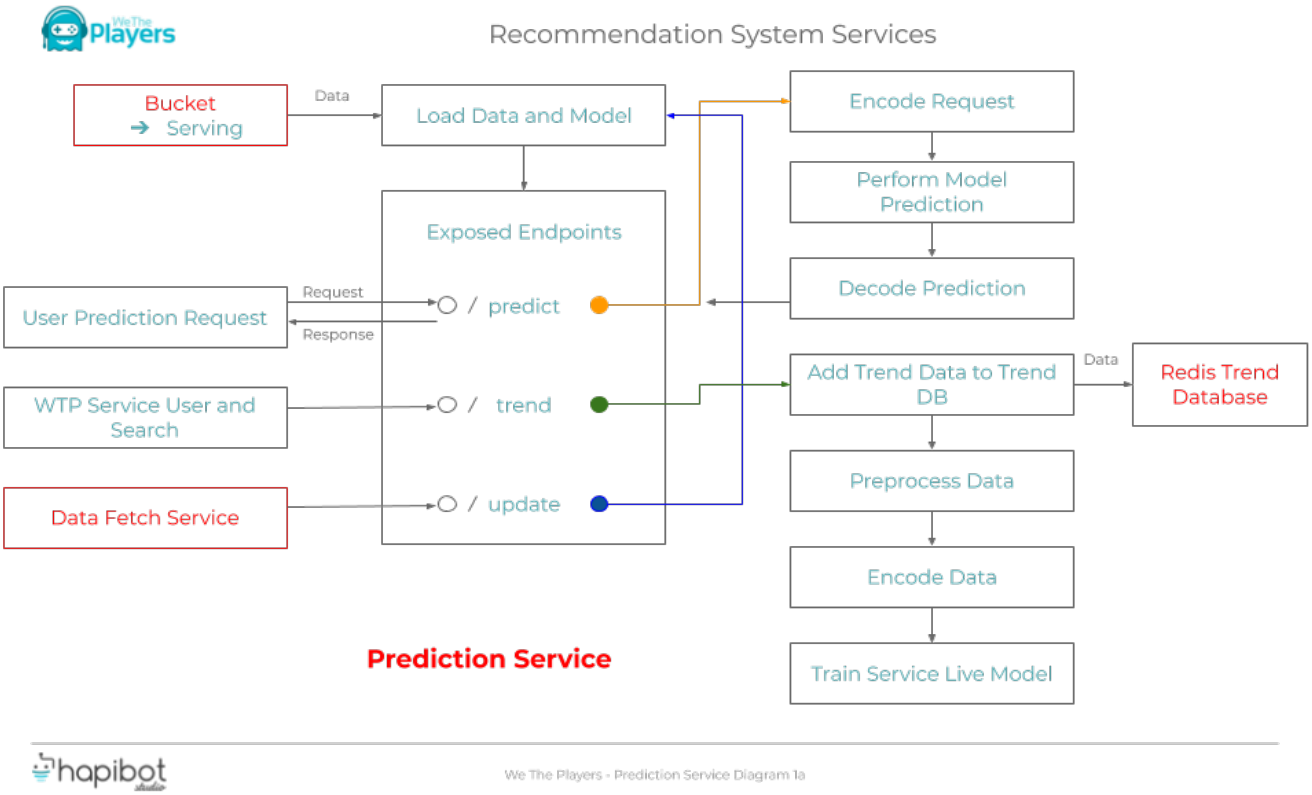


Figure 51: User prediction and trend caption service

B Code Snippets

B.1 Kubernetes configuration

Code Snipet 1: Example of YAML Kubernetes configuration file for a persistent volume and a persistent volume claim

```
apiVersion: v1
kind: PersistentVolume
metadata:
    name: storage-pv
    labels:
        type: local
spec:
    storageClassName: manual
    capacity:
        storage: 5Gi
    accessModes:
        - ReadWriteMany
    hostPath:
        path: "/mnt/data"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
```

```
    name: storage-pv-claim
    labels:
        app: storage
spec:
    storageClassName: manual
    accessModes:
        - ReadWriteMany
    resources:
        requests:
            storage: 5Gi
---
apiVersion: v1
kind: Pod
metadata:
    name: storage-access
spec:
    containers:
    - name: storage-files
      image: hapibot/server-files
      imagePullPolicy: Never
      volumeMounts:
      - name: storage-data
        mountPath: /mnt/data/serving
    volumes:
    - name: storage-data
      persistentVolumeClaim:
        claimName: storage-pv-claim
```

Code Snipet 2: Example of YAML Kubernetes configuration file for a model server

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tf-server
spec:
  replicas: 3
  selector:
    matchLabels:
      app: tf-server
  template:
    metadata:
      labels:
        app: tf-server
    spec:
      containers:
      - name: server-tf
        image: hapibot/server-tf:latest
        imagePullPolicy: Never
        ports:
        - name: tf-server
          containerPort: 8500
        env:
```

```
          - name: PORT
            value: "8500"
          - name: MODEL_NAME
            value: "lstm"
          - name: MODEL_PATH
            value: "/serving/tf-server/models"
          volumeMounts:
          - mountPath: "/serving/tf-server"
            name: tf-server-data
        restartPolicy: Always
        volumes:
        - name: tf-server-data
          persistentVolumeClaim:
              claimName: storage-pv-claim
---
apiVersion: v1
kind: Service
metadata:
    name: tf-server
    labels:
      service: tf-server
spec:
    selector:
        app: tf-server
        type: LoadBalancer
    type: ClusterIP
    ports:
    - name: tf-server
      port: 8500
      targetPort: 8500
```

B.2 Sequence preprocessing

Code Snipet 3: Example of Sequence tokenization

```
Sequences: [
    "Example sequence of tokenization",
    "Padding      of   sequence",
    "Numerical  vectors"
]

Tokenization map:{
    "example": 1,
    "sequence": 2,
    "of": 3,
    "tokenization": 4,
    "padding": 5,
    "numerical": 6,
    "vectors": 7
}

Sequences after preprocessing before tokenization:[
```

```
    "example sequence of tokenization",
    "padding of sequence",
    "numerical vectors"
]

Sequences after tokenization:[
    [1,2,3,4],
    [5,3,2],
    [6,7]
]

Max sequence length = 4

Ngram sequences:{
    x:[
        [1],
        [1,2],
        [1,2,3],
        [5],
        [5,3],
        [6]
    ],
    y:[
        [2],
        [3],
        [4],
        [3],
        [2],
        [7]
    ]
}

Sequences at end of processing (tokenization, ngram, padding):{
    x:[
        [0,0,0,1],
        [0,0,1,2],
        [0,1,2,3],
        [0,0,0,5],
        [0,0,5,3],
        [0,0,0,6]
    ],
    y:[
        [2],
        [3],
        [4],
        [3],
        [2],
        [7]
    ]
}
```

B.3 Docker compose

Code Snipet 4: Example for Docker Compose file

```
version: '3'
services:
    redis:
        image: "redis:alpine"
        container_name: wtp_search_prediction_trend_redis
        command: redis-server --requirepass redis
        restart: always
        ports:
            - "6379:6379"
        volumes:
            - $PWD/storage/redis:/data
    bucket:
        image: minio/minio
        container_name: wtp_search_prediction_bucket
        command: ["server", "/data"]
        restart: always
        environment:
            MINIO_ACCESS_KEY: key
            MINIO_SECRET_KEY: secret_key
        ports:
            - "9000:9000"
        volumes:
            - $PWD/storage/minio:/data
    training_studio:
        container_name: wtp_search_prediction_training_studio
        build:
            context: ./data_fetch_train
            dockerfile: Dockerfile
        restart: always
        env_file: ./data_fetch_train/.env
        environment:
            - HOSTNAME=redis
            - DB_PORT=6379
            - DB_PASS=redis
            - BUCKET_ENDPOINT=http://bucket:9000
            - BUCKET_ACCESS_KEY=key
            - BUCKET_SECRET_KEY=secret_key
            - SERVE_BUCKET_NAME=serving
        depends_on:
            - bucket
            - redis
        ports:
            - "5001:5001"
        volumes:
            - './wait-for-it.sh:/training/wait-for-it.sh'
        entrypoint: ["/training/wait-for-it.sh", "bucket:9000", "--"]
        command: ["python", "app.py"]
```

```
user_prediction_api:
    container_name: wtp_search_prediction_user_prediction_api
    build:
        context: ./serving_app
        dockerfile: Dockerfile
    restart: always
    env_file: ./serving_app/.env
    environment:
        - HOSTNAME=redis
        - DB_PORT=6379
        - DB_PASS=redis
        - BUCKET_ENDPOINT=http://bucket:9000
        - BUCKET_ACCESS_KEY=key
        - BUCKET_SECRET_KEY=secret_key
        - SERVE_BUCKET_NAME=serving
    depends_on:
        - bucket
        - redis
    ports:
        - "5000:5000"
    volumes:
        - './wait-for-it.sh:/serving/wait-for-it.sh'
    entrypoint: ["/serving/wait-for-it.sh", "bucket:9000", "--"]
    command: ["python", "app.py"]
```